Last compiled May 12, 2026 at 12:08 Noon.
© Kent Quanrud, 2026.

Fundamental Algorithms, CS381, Spring 2026

Instructors:

- *Mike Atallah* (`matallah@...`).
  Lectures: Tuesday and Thursday, 10:30–11:45AM, SMTH 108.
  Office hours: Thursday, after lecture, in LWSN 2116D.
- *Kent Quanrud* (`krq@...`).
  Lectures: Tuesday and Thursday, 1:30–2:45PM, BHEE 170.
  Office hours: Tuesday, after lecture, in LWSN 1211.

Graduate TAs:	Email (`@purdue.edu`)	Office hours (LWSN Commons)
Kyle Asbury	`asbury7`	Thursday, 1:30–2:30PM
Maksim Lavrenko	`mlavrenk`	Monday, 3:30–4:30PM
Qi Ling	`ling102`	Thursday, 9:30–10:30AM
Ethan Mader	`emader`	Thursday, 4:30–5:30PM
Han Qin	`qin184`	Tuesday, 2:30–3:30PM
Haoyu Song	`song522`	Tuesday, 1:30–2:30PM
Borna Tavasoli	`btavasol`	Thursday, 6–7PM
Justin Zhang	`zhan3554`	Wednesday, 12:30–1:30PM, Thursday, 3:30–4:30PM

Undergraduate TAs:	Email (`@purdue.edu`)	Office hours (LWSN Commons)
Mukul Agarwal	`agarw396`	Thursday, 2:30–3:30PM
Wilbert Chu	`chu285`	Monday, 2:00-3:00PM
Sharanya Devgon	`sdevgon`	Monday, 12:30–1:30PM
Jimmy Dinh	`dinh16`	Thursday, 11:30AM–12:30PM
Corbet Elkins	`cdelkins`	Thursday, 5–6PM
Aditya Gandhi	`gandh105`	Monday, 2:30–3:30PM
Pratyanch Jain	`jain617`	Thursday, 10:30–11:30AM
Amy Kang	`kang568`	Wednesday, 7–8PM
Maanas Karwa	`mkarwa`	Thursday, 4–5PM
Leo Lee	`lee4247`	Wednesday, 10:30–11:30AM
Matthew Li	`li4691`	Wednesday, 4:30–5:30PM
Philip Liu	`liu3688`	Monday, 4:30–5:30PM
Segyul Park	`park1362`	Wednesday, 2:30–3:30PM

Aakash Patel	`pate2011`	Wednesday, 3:30–4:30PM
Mridu Prashanth	`mprasha`	Thursday, 3–4PM
Angela Qian	`qian220`	Thursday, 12:30–1:30PM
Alexandre Sauquet	`asauquet`	Tuesday, 4:30–5:30PM
Diya Singh	`singh959`	Tuesday, 10:30–11:30AM
Sage Stefonic	`sstefoni`	Monday, 11:30AM–12:30PM
Kovid Tandon	`tandon34`	Tuesday, 3:30–4:30PM
Madhav Valiyaparambil	`mvaliyap`	Wednesday, 5:30–6:30PM
Nathan Vijitbenjaronk	`wvijitbe`	Thursday, 9:30–10:30AM
Aaryan Wadhwani	`wadhwan2`	Wednesday, 12:30–1:30PM

Collectively, we have office hours in LWSN Commons on:

- Monday, 11:30AM–1:30PM and 2:00–5:30PM
- Tuesday, 10:30–11:30AM and 1:30–5:30PM
- Wednesday, 10:30–11:30AM, 12:30–1:30PM, 2:30–6:30PM, and 7–8PM
- Thursday, 9:30AM–7PM

PSO	Day and time	Room	TAs
P01	Tuesday, 3:30–4:20PM	ARMS B071	Aditya, Justin, Sage, Segyul
P02	Tuesday, 4:30–5:20PM	ARMS B071	Kovid, Maksim, Mukul, Pratyanch
P03	Wednesday, 10:30–11:20AM	WALC B091	Alexandre, Kyle, Nathan
P04	Wednesday, 11:30–12:20PM	WALC B091	Aaryan, Madhav, Matthew, Song
P08	Wednesday, 2:30–3:20PM	WALC 3122	Aakash, Han, Mridu, Philip
P07	Wednesday, 3:30–4:20PM	WALC 3122	Angela, Ethan, Maanas, Sharanya
P05	Thursday, 10:30–11:20AM	WALC B091	Amy, Jimmy, Leo, Qi
P06	Thursday, 11:30–12:20PM	WALC B091	Borna, Corbet, Diya

Please see the syllabus (Page 726) for more information.

Homework

- Homework 0 due January 22.
- Homework 1 due January 29.
- Homework 2 due February 5.
- Homework 3 due February 12.
- Homework 4 due February 26.
- Homework 5 due March 5.

- Homework 6 due March 12.
- Homework 7 due March 26.
- Homework 8 due April 9.
- Homework 9 due April 16.
- Homework 10 due April 23.

Class schedule[1]

1. *January 13.* Searching and sorting, including binary search, merge sort, and lower bounds for sorting (Chapter 1).
2. *January 15.* Induction and recursion (Chapter 2).
3. *January 20.* Selection and closest pair in $\mathbb{R}^2$ (Chapter 3).
4. *January 22.* Fast integer multiplication and Fourier transform (Chapter 4).
5. *January 27.* Edit distance and matrix chain multiplication (Chapter 5).
6. *January 29.* Depth-first search and strongly connected components (Chapter 6).
7. *February 3.* Breadth-first search and single-source shortest paths (Chapter 7).
8. *February 5.* Negative edge weights and all-pairs shortest paths (Chapter 8).
9. *February 10.* Boolean satisfiability, CNF satisfiability, and reduction from Boolean to CNF SAT (Chapter 9).
10. *February 12.* 2-SAT and Max 2-SAT (Chapter 10).

($\star$) *Wednesday, February 18, 8–10PM at BHEE 129 & 170:* Midterm 1. (Scrambled problems; see also Chapter G.)

11. *February 24.* Subset sum (Chapter 11).
12. *February 26.* Hamiltonian path on graphs and on DAG's (Chapter 12).
13. *March 3.* Independent set, coloring, and other optimization problems on graphs and special cases of graphs (Chapter 13).
14. *March 5.* NP-Completeness, and big-picture summary (Chapter 14).
15. *March 10.* Spanning trees. (Chapter 16).
16. *March 12.* Disjoint paths and min-cut (Chapter 17).

($\star$) Spring Break

[1] All future dates are tentative. Lecture materials and video recordings can be found at the end of each chapter.

17. *March 24.* Network flow. (Chapter 18).
18. *March 26.* Applications of disjoint paths and network flow (Chapter 19). (Chapter 22).

($\star$) *Wednesday, April 1, 8–10PM at WTHR 104 & 200:* Midterm 2. (Scrambled problems; see also Chapter G.)

19. *April 7.* Lazy data structures and amortized analysis (Chapter 22).
20. *April 9.* Randomized searching and sorting (Chapter 25).
21. *April 14.* Priority queues and disjoint union (Chapter 23).
22. *April 16.* Hashing and heavy hitters (Chapter 27).
23. *April 21.* Randomized rounding (Chapter 26).
24. *April 23.* The push-relabel algorithm for flow (as another example of amortized analysis, Chapter 24).
25. *April 28.* Matchings (Chapter 21).
26. *April 30.* Review.

($\star$) *Saturday, May 2.* Indy Mini.[2]

($\star$) *Tuesday, May 5, 1–3PM, at LILY 1105 & MTHW 210:* Final. (All the exercises in random order. See Chapter G for practice problems from other textbooks.)

[2]TA Justin ran a 17:32 5K! TA Mridu ran the 5k too!

Contents

Basic information 1

Class schedule 3

I Lecture notes 15

1 Searching and sorting 16
- 1.1 Binary search . . . 16
- 1.2 Sorting . . . 19
- 1.3 Lower bounds for sorting . . . 26
- 1.4 Further sorting algorithms . . . 28
- 1.5 Additional notes and references . . . 30
- 1.6 Exercises . . . 31

2 Induction and recursion 39
- 2.1 One step at a time . . . 39
- 2.2 Towers of Hanoi . . . 40
- 2.3 Greatest common divisor . . . 42
- 2.4 Bipartite matchings and vertex cover . . . 44
- 2.5 Induction on planar graphs . . . 47
- 2.6 Additional notes and materials . . . 51
- 2.7 Exercises . . . 52

3 Selection and closest pair in the plane 61
- 3.1 Selection . . . 61
- 3.2 Minimum distance in $\mathbb{R}^2$ . . . 70
- 3.3 Additional notes and references . . . 74
- 3.4 Exercises . . . 75

4 Fast multiplication and Fourier transform **81**
4.1 Multiplication . . . 81
4.2 Matrix multiplication . . . 84
4.3 Fourier transforms . . . 85
4.4 Additional notes and references . . . 92
4.5 Exercises . . . 93

5 Sequential optimization with caching **96**
5.1 Edit distance . . . 96
5.2 Caching, a.k.a. dynamic programming: a meta-algorithm . . . 101
5.3 Sequential matrix multiplication . . . 104
5.4 TEX . . . 107
5.5 Additional notes and references . . . 108
5.6 Exercises . . . 109

6 Searching and sorting graphs **122**
6.1 Directed acyclic graphs and topological sort . . . 124
6.2 Depth-first search . . . 126
6.3 Post-DFS Order . . . 129
6.4 Strongly-connected components . . . 130
6.5 Additional notes and materials . . . 135
6.6 Exercises . . . 135

7 Shortest paths **141**
7.1 Distances . . . 141
7.2 Distances in a DAG . . . 142
7.3 BFS for unweighted shortest paths . . . 144
7.4 Dijkstra's algorithm for weighted shortest paths . . . 146
7.5 Additional notes and materials . . . 150
7.6 Exercises . . . 151

8 Negative edge lengths and all-pairs shortest paths **160**
8.1 Negative edges, shortest walks, and shortest paths . . . 160
8.2 Bellman-Ford: distances with negative edges . . . 161
8.3 All pairs shortest paths and the doubling trick . . . 164
8.4 Floyd-Warshall . . . 166
8.5 Additional notes and references . . . 167
8.6 Exercises . . . 168

9 SAT: a different kind of search problem **171**
9.1 Boolean formulas . . . 171
9.2 CNF . . . 174
9.3 Circuits . . . 177
9.4 Searching for answers . . . 179
9.5 Additional results . . . 180
9.6 Additional notes and references . . . 182
9.7 Exercises . . . 183
Appendix 9.A Proof of Theorem 9.1 . . . 186

10 2-SAT and Max 2-SAT **189**
10.1 2-SAT . . . 189
10.2 Max 2-SAT . . . 192
10.3 Additional notes and materials . . . 196
10.4 Exercises . . . 197

11 Subset-Sum, and the secret language of numbers **203**
11.1 Algorithms for subset sum . . . 203
11.2 Packing in the complexity . . . 205
11.3 Perfect is the enemy of good . . . 209
11.4 Additional notes and references . . . 210
11.5 Exercises . . . 211

12 The longest path **217**
12.1 The longest path in a DAG . . . 217
12.2 The longest path in general graphs . . . 219
12.3 Additional notes and materials . . . 222
12.4 Exercises . . . 222

13 Optimization in graphs **233**
13.1 Independent set . . . 233
13.2 Graph coloring . . . 242
13.3 Dominating set . . . 246
13.4 Additional notes and materials . . . 248
13.5 Exercises . . . 249

14 NP-Completeness **257**
14.1 Logical circuits . . . 257
14.2 Every program is a circuit . . . 259

14.3 Cook-Levin theorem . . . 261
14.4 NP-Completeness . . . 263
14.5 Recap . . . 264
14.6 Additional notes and materials . . . 265
14.7 Exercises . . . 266

15 Polynomial lower bounds **268**
15.1 Disjoint sets . . . 268
15.2 Furthest pair and diameter . . . 270
15.3 Closest pair . . . 271
15.4 Longest common subsequence . . . 271
15.5 Additional notes and materials . . . 278
15.6 Exercises . . . 279

16 Spanning trees **280**
16.1 Unweighted graphs . . . 281
16.2 Weighted graphs . . . 284
16.3 Steiner trees . . . 292
16.4 Additional notes and materials . . . 294
16.5 Exercises . . . 295

17 Packing and covering paths **300**
17.1 Duality . . . 301
17.2 Augmenting paths . . . 304
17.3 Multicommodity flow . . . 307
17.4 Additional notes and materials . . . 308
17.5 Exercises . . . 309

18 Network flow and linear programs **312**
18.1 Edge capacities, fractional path packings, and capacitated cuts . . . 312
18.2 Network flow . . . 314
18.3 Algorithms for maximum flow . . . 317
18.4 Minimum cost metrics, linear programming and LP duality . . . 320
18.5 Additional notes and references . . . 324
18.6 Exercises . . . 325

19 Applications of flows and cuts **327**
19.1 The problems . . . 327
19.2 Reductions to flows and cuts . . . 333

19.3 Additional notes and references . . . 340
19.4 Exercises . . . 341

20 Trees, flows, and cuts **347**
20.1 Global minimum cut . . . 347
20.2 Sparsification and faster augmenting paths . . . 353
20.3 Linear time greedy forest packings . . . 356
20.4 Notes and references . . . 359
Appendix 20.A Computing MA orderings in linear time in unweighted graphs 359

21 Matchings and vertex covers **361**
21.1 Packing and covering edges . . . 361
21.2 Vertex cover . . . 363
21.3 General matching . . . 364
21.4 Additional notes and materials . . . 371
21.5 Exercises . . . 372

22 Lazy data structures **374**
22.1 Binary counters . . . 374
22.2 Amortized analysis . . . 375
22.3 Array Lists . . . 379
22.4 Lazy search trees . . . 382
22.5 Immutable data structures . . . 386
22.6 Additional notes and references . . . 388
22.7 Exercises . . . 389

23 Priority queues and disjoint union **392**
23.1 Priority queues . . . 392
23.2 Disjoint union . . . 396
23.3 Additional notes and materials . . . 401
23.4 Exercises . . . 402

24 Push-relabel **404**
24.1 Push-Relabel Max Flow . . . 404
24.2 Top-down push-relabel . . . 410
24.3 Accelerating push-relabel with data structures . . . 412
24.4 Scaling push-relabel . . . 414
24.5 Additional notes and references . . . 415
24.6 Exercises . . . 416

25 Randomized searching and sorting **419**
25.1 First randomized algorithms . . . 419
25.2 Basic probability . . . 422
25.3 Randomized approximations for SAT . . . 429
25.4 Randomized sorting . . . 433
25.5 Additional notes and materials . . . 436
25.6 Exercises . . . 438

26 Randomized Rounding **450**
26.1 SAT . . . 452
26.2 Set cover . . . 456
26.3 Covering integer programs . . . 458
26.4 Additional notes and materials . . . 461
26.5 Exercises . . . 462
Appendix 26.A Proof of the multiplicative Chernoff bound . . . 465

27 Hashing and heavy hitters **467**
27.1 Introduction . . . 467
27.2 Hashing . . . 470
27.3 Hash tables with chaining . . . 471
27.4 Using hashing to approximate frequencies . . . 473
27.5 Amplification . . . 475
27.6 Extensions . . . 476
27.7 Takeaways . . . 477
27.8 Additional notes and references . . . 478
27.9 Exercises . . . 479

28 Online algorithms **487**
28.1 Caching . . . 487
28.2 Buy-or-rent . . . 491
28.3 Additional notes and materials . . . 495
28.4 Exercises . . . 495

29 Hash tables and linear probing **497**
29.1 Dictionaries . . . 497
29.2 Hash tables with chaining . . . 501
29.3 Linear probing . . . 502
29.4 4-wise independence . . . 508
29.5 Takeaways . . . 511

29.6 Additional notes and materials . . . 511
29.7 Exercises . . . 512

30 Fun with trees: splay trees, Euler trees, and dynamic connectivity 513
30.1 Splay trees . . . 513
30.2 Dyamic trees, subtree queries, and Euler trees . . . 517
30.3 Dynamic Connectivity . . . 521
30.4 Additional notes and references . . . 525
30.5 Exercises . . . 526
Appendix 30.A Amortized analysis of the remaining splay tree operations . . . 526

31 Range Queries and Dynamic Trees 529
31.1 Range queries . . . 529
31.2 Range queries on paths in a tree . . . 530
31.3 Dynamic trees . . . 531
31.4 Roots, re-rooting, and an application to dynamic disjoint union . . . 537

32 Dynamic Connectivity and MST's 539
32.1 Range queries . . . 539
32.2 Euler trees . . . 540
32.3 Dynamic Connectivity . . . 543
32.4 Dynamic MST . . . 547
32.5 Exercises . . . 548
32.6 Additional notes and references . . . 548

33 Convex minimization 549
33.1 Preliminaries . . . 549
33.2 Convex functions . . . 550
33.3 Gradient descent for smooth functions . . . 553
33.4 Gradient descent for smooth and convex functions . . . 555
33.5 Exercises . . . 557

34 Greedy Approximations and LP Duality 559
34.1 Maximum coverage . . . 559
34.2 Linear programs and LP duality . . . 561
34.3 An algorithmic proof of LP duality . . . 563

35 Random Sums and Graphs **569**
35.1 Random sums . . . 569
35.2 Random graphs . . . 571
35.3 A gap in component size . . . 574
35.4 Galton-Watson process with general branching factors . . . 577
35.5 Additional notes and materials . . . 581
35.6 Exercises . . . 581

A Additional exercises **583**

B Basic data structures **584**
B.1 Stacks . . . 584
B.2 Heaps . . . 585

C Mathematical Background **586**
C.1 Estimating sums (quickly) . . . 586

D Some NP-Complete Problems **590**

Bibliography **593**

II Course materials **607**

E Homework assignments **608**
Homework 0 . . . 609
Homework 1 . . . 610
Homework 2 . . . 611
Homework 3 . . . 612
Homework 4 . . . 613
Homework 5 . . . 614
Homework 6 . . . 615
Homework 7 . . . 616
Homework 8 . . . 617
Homework 9 . . . 618
Homework 10 . . . 619

F Selected solutions **620**
F.1 Homework 0 . . . 620
F.2 Homework 1 . . . 631

F.3 Homework 2 . . . 641
F.4 Homework 3 . . . 651
F.5 Midterm 1 . . . 658
F.6 Homework 4 . . . 662
F.7 Homework 5 . . . 669
F.8 Homework 6 . . . 675
F.9 Homework 7 . . . 680
F.10 Midterm 2 . . . 683
F.11 Homework 8 . . . 692
F.12 Homework 9 . . . 697
F.13 Homework 10 . . . 705
F.14 Final . . . 711

G Practice problems for exams **717**
G.1 Mapping to other textbooks . . . 717
G.2 Problems from [Eri19] . . . 718

H Aggregate statistics **721**
H.1 Homework . . . 721
H.2 Midterm 1 . . . 722
H.3 Midterm 2 . . . 723
H.4 Final . . . 724
H.5 Total . . . 725

I Syllabus, policies, and procedures **726**
I.1 Textbooks . . . 726
I.2 Correspondence . . . 727
I.3 Grading . . . 728
I.4 Exams . . . 728
I.5 Homework . . . 728
I.6 Academic integrity . . . 731
I.7 Posting Class Material . . . 731
I.8 Purdue's Honor Pledge . . . 732
I.9 Grief Absence Policy . . . 732
I.10 Conduct and Courtesy . . . 732
I.11 Students with Disabilities . . . 732
I.12 Emergencies . . . 733
I.13 Violent Behavior Policy . . . 733
I.14 Mental Health and Wellness . . . 734

I.15 Health in general . . . 735
I.16 Basic needs and security . . . 735
I.17 Nondiscrimination . . . 735
I.18 Privacy . . . 736
I.19 Netiquette . . . 736
I.20 Changes to the syllabus . . . 737

J PSO **738**
J.1 Week 0 . . . 739
J.2 Week 1 . . . 742
J.3 Week 2 . . . 747
J.4 Week 3 . . . 751
J.5 Week 4 . . . 754
J.6 Week 5 . . . 755
J.7 Week 6 . . . 758
J.8 Week 7 . . . 760
J.9 Week 8 . . . 762
J.10 Week 9 . . . 764
J.11 Week 10 . . . 766
J.12 Week 11 . . . 770
J.13 Week 12 . . . 772
J.14 Week 13 . . . 775
J.15 Week 14 . . . 778

Part I

Lecture notes

Chapter 1

Searching and sorting

1.1 You already know a lot about theoretical computer science

I am not sure if the following story is true, but it makes a basic point. Bill is an electrical engineer working in chip verification: that is, he looks for bugs in computer chips. Nevermind the details. At some kind of team meeting at his work, a co-worker proudly boasts that they can count the number of remaining bugs on two hands — that is, at most 10. To which Bill replies:

> "You mean, 1024?!"

He might have been off by 1, but the point is clear: your ten fingers make ten *binary digits*, a.k.a., *bits*. We usually count starting from two closed fists — 0 — and extend one finger at a time until our hands are open and we've counted to 10. But there are many more *combinations* of open and closed fingers: 2^{10} combinations, to be exact. You could probably train yourself to finger-count all the way to 1023 in binary.

This is a trick that we all know very well. It is implicit every time we use decimal digits, instead of "caveman" style tallies. Below are two different ways to express the number 23:

$$||||||||||||||||||||||| = 23.$$

Clearly the right-hand side (RHS) is a much more efficient representation than the left-hand side (LHS).[1] The RHS leverages the mathematical fact that the number of strings of k characters from an alphabet of size 10 (namely, $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$) is 10^k. *Duh.* But the k in the top-right corner of "10^k" is extremely important. It indicates an *exponential growth* in the number of possible combinations, as a function of the number of digits. We see such an exponential growth for any alphabet size (e.g., the set of bits $\{0, 1\}$ has size 2) — except, of course, for alphabets of size 1.

[1] In fact, a previous version of this note mistakenly had 25 tallies on the left, and nobody noticed.

Over/under. Consider the following simple game we played in elementary school called "over-under". The teacher declares to the class that he or she has secretly picked a number between 1 and 100. The game then proceeds in rounds. Each round, a student from the class guesses a number between 1 and 100. The teacher then replies that the guess is either equal, over, or under the secret number. The game goes back and forth, with the students guessing a number and the teacher giving the over/under, until the students guess the secret number. The goal of the students is to guess the number in as few rounds as possible.

Of course it takes at most 100 guesses by looping through the numbers from 1 to 100. This approach is slow and boring. There is a better strategy that is pretty obvious, even to children. *Guess* 50. Not because 50 represents an average in the range, but because it is the *median*. The teacher replies either "over" or "under". In the first case, you have restricted the range of possibilities to the numbers 1 through 49. In the second, the remaining range is 51 through 100. In either case, you have reduced the problem from 1 of 100 possibilities, to 1 of 50. Next round, you should guess 25 or 75, and so forth.

How many rounds are required? By picking the median of the remaining range, each round reduces the number of possibilities by half. Thus the number of rounds is the number of "halvings" required to reduce the number of possibilities from 100 to 1. There is a mathematical term for this and it is called the *logarithm base* 2 of 100, denoted $\log_2(x)$. For a base $b > 1$, $\log_b(x)$ is the function defined by

$$b^{\log_b(x)} = x.$$

That is, the number of times we multiply by b (starting from 1) to get x. For us, we have

$$\log_2(100) = 6.643856...$$

This implies that 6 rounds are enough.[2] The real point is that by repeatedly dividing the space in half, we only need $\log(n)$ guesses instead of n, to find a number.

This is our first algorithm; it is called *binary search*. Binary search doesn't "guess" the secret number; it deduces it. Formally, `binary-search` finds an element in a sorted array of n elements in $O(\log n)$ time. See Fig. 1.1 on the following page for pseudocode.

[2]Why 6 instead of 7? Certainly 7 rounds are enough. 6 rounds are enough to reduce to (strictly) less than 2 choices (because $\log_2(50) = \log_2(100) - 1 < 6$), and the only integer less than 2 is 1! So 6 actually suffices. That said, we normally don't care about a difference of one step — what's one more step for a computer?

`binary-search(`$A[1..n]$`, `x`)`

/ Given an array of n comparable elements $A[1..n]$ in sorted order, and another comparable element x, returns an index i such that $A[i] = x$, or* `nil` *if no such index exists.* */*

1. If $n = 0$ return `nil`.
2. $m \leftarrow \lceil (n+1)/2 \rceil$.
3. If $A[m] = x$ then return m.
4. If $A[m] > x$ then return `binary-search(`$A[1..m-1]$`,`x`)`.
5. If $A[m] < x$ then return $m +$ `binary-search(`$A[m+1..n]$`,`x`)`.

Figure 1.1: A recursive implementation of binary search.

Exponential and logarithmic. We now have two mathematical functions that we all have understood implicitly since early childhood: the exponential function, 2^x, and the logarithm, $\log_2(x)$. The former represents combinatorial explosion; the latter represents the awesome efficiency of binary search. Algorithm design is basically about appreciating and exploiting the difference between these two functions. As simple as that may sound, it is a concept that we do not totally understand.

There is a world of difference between the exponential function and the logarithm. In fact "a world" may be understating it. Consider the number of atoms in the universe. This is called the *Eddington number*. The current estimate on the Eddington number is about 10^{80}, which would be painful to write out in full. Now, the *logarithm base 2* of the Eddington number is

$$\log_2\left(10^{80}\right) = 80 \log_2(10) \approx 265.75.$$

265.75 is a large number but not astronomical. (You can even count it on two hands!) Now suppose we take the logarithm *again*. We have

$$\log_2 \log_2(\#\text{ atoms in the universe}) \approx \log_2(265.75) \approx 8.$$

8! Just 8! You could have finger-counted to eight the old-fashioned way.

So the logarithm rapidly takes astronomical numbers to very modest ones. 265.75 seems small given the scale of the universe. You could play the over/under game for a secret number between 1 and the Eddington number, and in a medium (if boring) amount of time, win! In many ways, binary search is the ideal algorithm. We will encounter many problems that we wish were as easy as over/under.

1.2 Sorting

This brings us to the topic of sorting. Sorting is maybe the most useful thing a computer can do. Once n items are put in sorted order, we can locate any one of them in $O(\log n)$ time by binary search (i.e., the over-under game).

Human-sort. How do we (as humans) sort? Take for example the following 10 numbers, in arbitrary order.

$$47,\ 7,\ 82,\ 31,\ 87,\ 63,\ 19,\ 94,\ 86,\ 43.$$

Suppose we want to sort these numbers in increasing order. The most natural way to go about this is to build out the sorted list from beginning to end, one at a time. First, we need to find the first number. Scanning the list above, we see that it is 7. We write down 7, and mark it off in the input list to remind ourselves that we've already placed 7 in the output list. Below we start our output list on the left and keep track of the marks on the right.

7

47, ~~7~~, 82, 31, 87, 63, 19, 94, 86, 43

The next step, naturally, is to find the second smallest number. We scan the input list looking for the smallest number not yet crossed off: 19.

7, 19

47, ~~7~~, 82, 31, 87, 63, ~~19~~, 94, 86, 43

We continue in this fashion, identifying the third number, then the fourth, then the fifth... Eventually, we will finish the list. The last few lines of our transcript are

$$\vdots$$

7, 19, 31, 43, 47, 63, 82, 86, 87

~~47~~, ~~7~~, ~~82~~, ~~31~~, ~~87~~, ~~63~~, ~~19~~, 94, ~~86~~, ~~43~~

7, 19, 31, 43, 47, 63, 82, 86, 87, 94

~~47~~, ~~7~~, ~~82~~, ~~31~~, ~~87~~, ~~63~~, ~~19~~, ~~94~~, ~~86~~, ~~43~~

So that's how we naturally tend to sort. Let's call this algorithm "`human-sort`".[3] It is intuitive and simple. But is it a *good algorithm*? This question hardly matters for sorting ten numbers. But the point of computers is that they can be automated on huge inputs. When analyzing the running time, we want to understand how the

[3]This algorithm is more commonly called `selection-sort`.

algorithm *scales* with the input size. So we treat the input size as a *variable*. Let n denote the number of numbers in the input. (Above, $n = 10$.) How long does the algorithm take, as a function of n?

Our algorithm `human-sort` has n iterations, where in the ith iteration (out of n), it identifies the ith smallest number. To find this number, it scans the entire list of n numbers, having to do a few elementary operations for each (like checking for a mark, or comparing to the smallest number found so far). So the running time is roughly

$$(n \text{ iterations}) \times (\text{scanning } n \text{ items}) \approx n^2 \text{ operations overall.}$$

Is it exactly n^2 operations? The question is not well-posed, because we did not formally define what an "operation" is. But we clearly understand that as we scan the list, we need a constant amount of time to process each item. Maybe it is 1 operation, or maybe it is 10, or maybe it is 100. We won't worry about the exact count, in part because an exact accounting depends on the language and the hardware (which is always improving), and because this level of detail does not really inform high-level algorithm design.

As far we are concerned, there is some constant $C > 0$, *independent of* n, such that `human-sort` takes at most Cn^2 steps / units of time, at least for n sufficiently large. The point is not on C, but rather on the polynomial n^2. This emphasis is codified in a notation called "big-O". For a function $f : \mathbb{N} \to \mathbb{N}$, we say that an algorithm takes $O(f(n))$ time if there are constants $C > 0$ and $N > 0$ such that for all inputs of size $n \geq N$, the algorithm takes at most $Cf(n)$ "steps". Again we won't specify exactly what a step is. You and I can both generally agree on what is and isn't a single step; where there's disagreement, it will only affect the constant C. So we can generally agree on a $O(f(n))$ running time without worrying about what language we're programming in or what computer we're running it on. More broadly, we can begin to develop a *general theory* of algorithms.

In big-O notation, `human-sort` takes $O(n^2)$ time. The hidden constant C may depend on low-level details, but we all agree that *asymptotically* the algorithm grows like n^2. We call this a *quadratic* running time, since n^2 is a quadratic function of n.

Checking a sort. Can we do better? That is, can we sort numbers faster than $O(n^2)$ time? "Better?" is always the question in algorithm design. Let us remind ourselves that many of us have used `human-sort` all our lives without questioning it. $O(n^2)$ is also a natural running time, since it is the time required to compare all pairs of numbers. So when we are asking for a faster algorithm, we are really asking if it is possible to sort all numbers without directly comparing every pair of numbers.

Let us ask a related question from a different direction: verification. Given a list of n numbers, how long does it take to *verify* that it is sorted? (We pause to let the

reader consider this.)

A list of n numbers $x_1, \ldots, x_n$ is sorted in increasing order if and only if $x_i \leq x_{i+1}$ for every index $i = 1, \ldots, n-1$. To verify a list is sorted, we only need to compare every consecutive pair of numbers, x_i and x_{i+1}, and certify that $x_i \leq x_{i+1}$. So the answer is linear time: $O(n)$.

Now we have a gap. We can sort with `human-sort` in $O(n^2)$ time. We can verify that n numbers are sorted in $O(n)$ time. Can we sort as fast we can verify a list is sorted?

As one more baby step before unveiling our next algorithm, consider the following special case of sorting. Suppose our list of n numbers is *almost* sorted in the following sense. Assume n is even (for simplicity), and suppose the first half and second half of the list are both sorted amongst themselves. That is, out of n numbers $x_1, \ldots, x_n$, we are promised that

$$x_1 < x_2 < \cdots < x_{n/2} \text{ and that } x_{n/2+1} < x_{n/2+2} < \cdots < x_n.$$

We are *not* promised that $x_i < x_j$ when $i \leq n/2 < j$. For example, for the list from our discussion on `human-sort` (Page 19), suppose the first and second halves were sorted as

7, 31, 47, 82, 87 and 19, 43 63, 86, 94.

Given a "half"-sorted list in the above sense, how long does it take to produce a fully sorted list? Again we pause and encourage the reader to solve this.

Merge-sort. The reader might have discovered the following procedure to combine the two sorted lists into a single sorted list. We build the output list one by one, starting from the beginning. What is the smallest number in our two lists? We know automatically that it is the first number in one of the two sublists, so we only have to check these two numbers to identify it. This is already faster than `human-sort`, where we had to scan all of the items to find the first item. Next, we need the second smallest number. Now, depending on whether the smallest number came from the first or second list, the second smallest number overall will be either the first number in one list or the second number in the other. To expedite this process, we can keep track of how many numbers have been taken from the top of each list. Then in each

`merge`$(A[1..m], B[1..n])$

/ Given two sorted lists A and B, produces the combined lists in sorted order. */*

1. Allocate a new array $C[1..m+n]$, let $i = 1$, and let $j = 1$.
2. While $i \leq m$ and $j \leq n$:
   A. If $A[i] \leq B[j]$ *// Take from A.*
      1. $C[i+j-1] \leftarrow A[i]$ and $i \leftarrow i+1$.
   B. Else ($B[j] < A[i]$): *// Take from B.*
      1. $C[i+j-1] \leftarrow B[j]$ and $j \leftarrow j+1$.
3. While $i \leq m$:
   A. $C[i+j-1] \leftarrow A[i]$ and $i \leftarrow i+1$.
4. While $j \leq n$:
   A. $C[i+j-1] \leftarrow B_2[j]$ and $j \leftarrow j+1$.
5. Return C.

Figure 1.2: Merging two sorted lists into a single sorted list in linear time.

step we only need to check the two tops of the remaining parts of the lists for the next smallest number. Continuing in this fashion, we will eventually "zip up" the two sorted lists into a single, globally sorted list. This process takes a constant amount of time to produce each number in the output, so the overall running time for n numbers is $O(n)$.

We call this subroutine `merge`. Pseudocode is given in Fig. 1.2 above. Formally, we have the following guarantee.

Lemma 1.1. *Given two sorted lists $A[1..m]$ and $B[1..n]$, `merge`(A,B) returns the list containing all the elements of A and B in sorted order in $O(m+n)$ time.*

The `merge` subroutine gives rise to a sorting algorithm called `merge-sort`. `merge-sort` is very simple from a top-down perspective: given a list of unsorted numbers, divide the list into two halves. Sort the first half and sort the second half separately to produce two sorted lists, each containing half the numbers. Then merge them together to produce one sorted list. But wait – how do we sort the two halves? *By recursion:* we call `merge-sort` on each of the two sublists.

When designing a recursive algorithm such as `merge-sort` it is critical – both for ease of implementation, and proving correctness – that we declare what the algorithm does, as a sort of binding contract. We call this the *recursive specification*. For this instance, we define:

> `merge-sort`($A[1..n]$): Given an array $A[1..n]$ of comparable elements, returns an array consisting of the elements of A sorted in increasing order.

Here the input and the output (and properties thereof) are made explicitly clear.

A specification makes implementation easier by clarifying our intent. It also solidifies what we can assume from `merge-sort` when making recursive calls to smaller inputs. Indeed, the recursive specification maps directly to the *inductive hypothesis* when proving correctness.

Pseudocode for `merge-sort` is given in Fig. 1.3 on the following page. The task is now to analyze the algorithm. A full analysis of `merge-sort` should address two aspects.

1. Correctness: prove that `merge-sort` returns a sorted list.
2. Running time: identify a function $f(n)$, as small as possible, for which we can show that `merge-sort` takes $O(f(n))$ time on an input of n numbers.

We first prove the correctness – that `merge-sort` indeed sorts its input correctly. We prove this by induction on the number of numbers in the input, n.

In the base case, $n \leq 1$, the input is already sorted, and the algorithm simply returns it.

In the general case $n > 1$, we *assume by induction on* n that for all $k < n$, `merge-sort` correctly sorts any input of size k. For $n \geq 2$, the algorithm splits the input into two lists of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ and recursively calls `merge-sort` on these inputs. $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ are both strictly less than n because $n \geq 2$. By induction, both of these recursive calls correctly sorts their respective halves of the input. The algorithm then combines the two sorted sublists by merging. We have already discussed the correctness of merging above. Thus `merge-sort` correctly outputs the n numbers in sorted order.

Next we analyze the running time. The first step is to model the running time mathematically. Let $T(n)$ denote the amount of time taken by `merge-sort` on an input of size n. We can define $T(n)$ recursively as follows.

$$\begin{aligned} T(0) = T(1) &\leq C, \\ T(n) &\leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + Cn, \end{aligned}$$

for some constant $C > 0$. Let us point out that when n is even, the second line above simplifies to the cleaner form

$$T(n) \leq 2T(n/2) + Cn.$$

```
merge-sort(A[1..n])
/* In the base case, the list is so short there is nothing to do. */
1. If n ≤ 1 then return A.
/* In the general case, we divide the input in half and recursively sort each half. */
2. Let m = ⌊n/2⌋.
3. B_1[1..m] ← merge-sort(A[1..m]).
4. B_2[1..n − m] ← merge-sort(A[m + 1..n]).
/* Now we merge the two sorted halves in linear time. */
5. return merge(B_1, B_2).
```

Figure 1.3: The `merge-sort` algorithm.

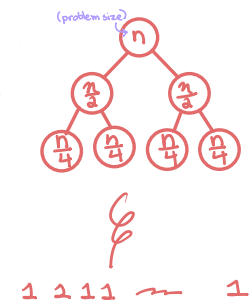
Morally, when doing an asymptotic analysis in n, we shouldn't fret over ± 1 associated with rounding $n/2$ up and down. Let us assume for the moment that the second recursion (where $T(n) \leq 2T(n/2) + Cn$) is simply a valid recursion for $T(n)$. We will justify this later. Alternatively, the reader can assume that n is a power of two, where the math is automatically cleaner. This assumption can also be justified formally.

Recursion trees. To analyze `merge-sort`, and in particular the recursion

$$T(0) = T(1) \leq C$$
$$T(n) = 2T(n/2) + Cn$$

for some absolute constant $C > 0$, let us briefly describe a useful tool called *recursion trees*. Recursion trees help us visualize the recursive subproblems in a tree-like structure. The idea is to create a tree where each node corresponds to a subproblem. One node is a child of another if the first node is a subproblem of the second.

For `merge-sort`, the root of the tree corresponds to our initial call to `merge-sort`$(A[1..n])$. The root has two children, corresponding to the subproblems for the recursive calls to `merge-sort`$(A[1..\lfloor \frac{n}{2} \rfloor])$ and `merge-sort`$(A[\lfloor \frac{n}{2} \rfloor + 1..n])$. When doing an analysis like this, I literally draw the first few layers of the tree. I annotate each node with the size of the input, as well as the amount of time spent on each subproblem, excluding recursive calls. (E.g., the root will be labeled with size n and work Cn.)

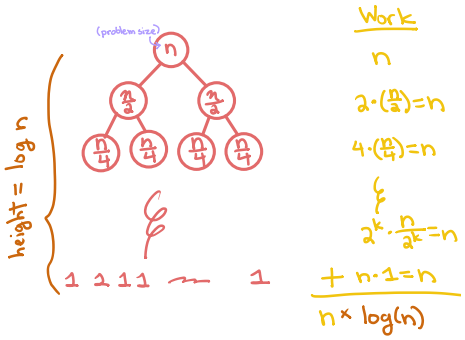


Having visualized the computation in this tree, we calculate the total running time as follows. For $i \in \mathbb{Z}_{\geq 0}$, let the *ith level* of the tree refer to the subproblems that are i

generations away from the root. The root is the only node in level 0, its two children form level 1, its four grandchildren form level 2, and so forth.

We divide the total computation into levels. We first upper bound the total amount of work at each level i, excluding recursive calls (which are accounted for in lower levels of the tree). Then we upper bound the number of levels. Finally, the overall running time is bounded above by the product of the work-per-level and the total number of levels.

Fix a level i. Label each subproblem with the size of the subproblem, and then the running time spent on that subproblem. In the ith level, we have 2^i subproblems whose total sizes added up to n. The total time spent on each subproblem, excluding recursive calls, is linear in the input size. Thus we spend $O(n)$ time on subproblems in level i, excluding recursive calls.



Next we identify the number of levels. Each successive level divides the problem size by 2. This means the leaves — corresponding to problems of size 1 — are at level (at most) $O(\log n)$, hence the tree has height $O(\log n)$ in all.

Now we combine everything together to get the overall running time. We have

$$(O(\log n) \text{ levels}) \times (O(n) \text{ time per level}) = O(n \log n) \text{ total time.}$$

$O(n \log n)$ is a sizeable improvement over the $O(n^2)$ time required by `human-sort`.

On the rounding error. We simplified our discussion by assuming the running time could be modeled by the recursion

$$T(n) \leq 2T(n/2) + Cn,$$

for an absolute constant C.

This can be justified by the following trick. Let us redefine $T(n)$ as the maximum amount of time taken by `merge-sort` on any input of size *less than or equal to* n. Note that by definition $T(n)$ is increasing.[4] Then we analyze $T(n)$ for the simpler case where n is a power of two, as above.

Now consider any other value n that is not a power of two. Then the value $n' = 2^{\lceil \log_2 n \rceil}$ is a power of 2, and $n \leq n' < 2n$. We can use $T(n')$ to upper bound

[4] Of course this seems true already just based on the `merge-sort` code, but it seems annoying to prove this formally.

$T(n)$: $T(n) \leq T(n') \leq O(n' \log n') = O(n \log n)$. Since n' is within a constant factor of n, $T(n')$ is within a constant factor of $T(n)$, and lazily upperbounding by $T(n')$ is not so bad.

A different way to justify the rounding error is as follows. We first write

$$T(n) \leq 2T(n/2 + 1) + Cn$$

Then we can define $U(n) = T(n + 2)$. Then we have

$$\begin{aligned} U(n) &= T(n+2) \\ &\leq T(\lfloor (n+2)/2 \rfloor) + T(\lceil (n+2)/2 \rceil) + C(n+2) \\ &\leq 2T(n/2+2) + C(n+2) \\ &= 2U(n/2) + C'n \end{aligned}$$

for $C' = (C + 2)$. So we could apply our cleaner analysis directly to $U(n)$ instead of $T(n)$, which in turn bounds $T(n)$.

Guess and check. The recursion tree makes it easy to (literally) see the $O(n \log n)$ running time. Alternatively, we could have guessed that the running time was at most $Dn \log n$ for some constant D, and then verified directly that it satisfies the recursion by induction on n.

1.3 Lower bounds for sorting

We now adopt a different perspective and consider *lower* bounds for sorting. That is, we want to identify some function $f(n)$ for which we can prove that *any* sorting algorithm takes *at least* $\Omega(f(n))$ time, in the worst case, on an input of size n.[5]

Lower bounds might seem less intuitive than upper bounds. It is clear how to prove an upper bound $O(f(n))$ for a given problem (such as sorting): we provide an algorithm along with a proof that the algorithm is correct and runs in $O(f(n))$ time in the worst-case. *Any single* correct algorithm and proof would suffice. For lower bounds, we need to argue that *every* correct algorithm – including all those we have never even imagined – must have at least a certain worst-case running time. Our reasoning must generalize beyond any specific algorithm, which is typically a more abstract discussion than when analyzing a single concrete algorithm.

We will obtain a lower bound specifically in the *comparison model*. The comparison model assumes the input is a list of distinct elements of some definite order, where

[5] "big-Ω", $\Omega(\cdots)$, is the lower bound counterpart of $O(\cdots)$. Some quantity x is $\Omega(f(n))$ if there exists constants $C, N > 0$ such that $x \geq Cf(n)$ for $n \geq N$.

the only information we can gain about these elements is by pairwise comparisons. Formally, we define a *comparison query* as a query where we take two elements x and y and find out if either $x < y$ or $y < x$. Rather than try to lower bound the running time directly, we will lower bound the number of comparison queries required by any sorting algorithm. This translates to a lower bound on the running time under the reasonable assumption that each comparison takes at least constant time.

Now, fix any sorting algorithm. Suppose the sorting algorithm uses only k comparisons in the worst case, and then (allegedly) outputs the sorted list. Consider the *transcript* of outputs of these comparisons, where for each comparison we record whether the first or second element in the query was larger. Observe that the only external information the algorithm takes in is the results of these queries. In particular, the first comparison query is predetermined. The second comparison query can only depend on the outcome of the first comparison. In general the ith comparison depends only on the outcomes of the preceding $i - 1$ comparisons. Ultimately, the algorithm outputs some reordered list as a function of *only* the outputs of the k comparison queries. Each query has 2 possible outcomes. Therefore the algorithm can output at most 2^k different lists.

Meanwhile, there are $n!$ different ways to order n elements (a.k.a. *permutations*). So to be able to produce all $n!$ possible permutations, we must have $2^k \geq n!$. That is,

$$k \geq \log_2(n!).$$

What is $\log_2(n!)$? The exact value of $n!$ is difficult to handle. However, we are only interested in a constant factor estimate. Moreover, a very loose estimate of $n!$ will suffice because we will take the log of that estimate anyway. To this end, observe that

$$n! = n(n-1)(n-2)\cdots 1 \geq (n/2)^{n/2},$$

hence

$$\log(n!) \geq \log\Big((n/2)^{n/2}\Big) = (n/2)(\log(n) - 1) = \Omega(n \log n).$$

Thus, up to constants, any algorithm must make at least $n \log n$ comparisons (just like `merge-sort`!) simply to be able to output all possible permutations.

Theorem 1.2. *Any deterministic comparison based sorting algorithm requires at least* $\log(n!) = \Omega(n \log n)$ *comparisons in the worst case.*

The theorem brings some closure to our discussion: `merge-sort` is essentially best possible. This conclusion – where we have an upper bound that is *tight* (i.e., best

possible) with the lower bound – is the ideal for any problem we are interested in. Most problems are not understood this well.

While we often focus on algorithms, lower bounds like Theorem 1.2 are important too. We will see some problems that are so tricky we haven't found good algorithms despite many years of research – but instead we have (conditional) lower bounds that suggest these problems *should be hard*.

We sometimes think of upper bounds and lower bounds as competing in an adversarial relationships – and certainly they bound one another mathematically – but in another sense they are two sides of the same coin. Neither the $O(n \log n)$ upper bound nor the $\Omega(n \log n)$ lower bound are as compelling taken alone as they are together, where they *mutually certify* one another to be tight (up to constants).

Now, every theorem has its terms and conditions and so we remind ourselves of exactly what we proved. The theorem states that when the elements can only be compared pairwise, any correct sorting algorithm must make at least $\Omega(n \log n)$ pairwise comparisons. This does not limit related tasks such as finding the median element (which we discuss later) or sorting only the first k elements (exercise 1.9). Another situation the theorem does not cover is where we are sorting integers represented by bit strings of a fixed length. One might try to take advantage of the bits and somehow sort the numbers without actually having to compare the numbers pairwise. (The `radix-sort` algorithm takes this approach.)

1.4 Further sorting algorithms

`merge-sort` is clean and serves as a nice introduction to divide-and-conquer and for solving recurrences. It is also optimal in the comparison model. However it is not the only $O(n \log n)$ time algorithm and here we list a few more approaches.

Sorting with heaps. Recall that the *heap* data structure (cf. Section B.2) maintains a collection of comparable items with direct access to the minimum one. More precisely, the method `insert` will insert an element, and `remove-min` will remove the minimum element, both in $O(\log n)$ time, where n is the number of items in the heap. Note that a heap does not necessarily sort all the elements; it merely promises to keep track of the smallest one. Still we can use a heap to sort the items, as follows.

`heap-sort`$(x_1, \ldots, x_n)$

1. Initialize an empty heap H.
2. Insert each element x_i into H.
3. Repeatedly call `remove-min` on H until the heap is empty, and return the elements in the order they are output by H.

Given the running times for the heap operations, it is easy to analyze the sorting algorithm above. Each element takes $O(\log n)$ to insert. Each element removed takes $O(\log n)$ time. This gives the following bound.

Theorem 1.3. `heap-sort` *sorts n items in $O(n \log n)$ time.*

`heap-sort` is our first example of using a data structure to facilitate faster running times.

Quicksort. The algorithm `quick-sort` is very simple and effective in practice, but also *randomized*. `quick-sort` is (like `merge-sort`) a divide and conquer algorithm, but divides the data in a different and *randomized* way. Given n unsorted items, it selects one item x *uniformly at random*. This item x is called the *pivot*. It then divides the remaining items into two (unsorted) sets: the set of items smaller than x, and the set of items bigger than x. (It takes $O(n)$ comparisons to do this.) We sort each set separately by recursive calls to `quick-sort`, and concatenate the returned lists with x in between. See Fig. 1.4 on the next page for pseudocode.

Intuitively, when selecting a random pivot x_i, we are hoping that x_i is close to the median. Then A and B will roughly divide the input in half. If indeed x_i always divided the input in half (or close to half), then a $O(n \log n)$ running time follows from essentially the same analysis as `merge-sort`. Of course we may be unlucky and in the worst case `quick-sort` could run in $O(n^2)$ time. But this is the wrong way to look at it.

When analyzing randomized algorithms, we are more interested in bounding the *average running time* with respect to *any* input. Here "average" is meant exclusively to the randomization within the algorithm. (We are not averaging over a distribution of inputs.) Here, one can show that `quick-sort` runs in $O(n \log n)$ time on average (for any list of n items).

Theorem 1.4. `quick-sort` *sorts n items in $O(n \log n)$ randomized time in expectation.*

`quick-sort` is appealing because it is very simple to implement; this simplicity appears to translate well to practice. Indeed it is the default sorting subroutine in `unix`. The only tricky part is in the analysis, because the analysis is *probabilistic*. Actually the analysis itself is not too complicated, were it not for that probability theory is new (and appears strange) for many students.

We briefly sketch the high-level ideas here since it is interesting; we will do the analysis again much more carefully when we study randomized algorithms in detail in Chapter 25.

`quick-sort`$(x_1, \ldots, x_n)$

/* *We assume the elements are distinct for simplicity.* */

1. If $n = 0$ return the empty list and if $n = 1$ return the list consisting of x_1.
2. Otherwise randomly sample x_i uniformly at random.
3. Let $A = \{x_j : x_j < x_i\}$ and let $B = \{x_j : x_j > x_i\}$. // $n - 1$ *comparisons.*
4. Recursively sort A and sort B, and combine the sorted sublists with x_i inserted in between. Return the combined list.

Figure 1.4: The `quick-sort` algorithm.

Clearly the running time is proportional to the total number of comparisons. Let y_i refer to the ith largest element. For $i < j$, what are the odds that y_i is compared to y_j? It is the probability that y_i or y_j gets randomly selected as a pivot before any of the in-between elements, $y_{i+1}, \ldots, y_{j-1}$. Among the elements $y_i, \ldots, y_j$, all are equally likely to be selected first. Thus the probability of comparing y_i to y_j is $2/(j - i + 1)$. Now, the key observation is that

> *by* linearity of expectation *(which is introduced in Chapter 25), the expected number of comparisons equals the sum, over all $i < j$, of the probability that y_i is compared to y_j.*

Now the sum of $1/(j - i + 1)$, over all $1 \leq i < j \leq n$, comes out to $O(n \log n)$.

One can show further that `quick-sort` runs in $O(n \log n)$ time with exceedingly high probability; in particular, the $O(n^2)$ running time will almost never occur (for large n).

Etcetera. There are many more possible sorting algorithms as well as heuristics[6] and hybrid approaches that help performance in practice. For example, the Python `sort` subroutine checks the input for large *runs* (consecutive subsequences that are already sorted) that can effectively be skipped; in particular, it takes $O(n)$ time if the list is already sorted.

1.5 Additional notes and references

`merge-sort` was invented by Jon von Neumann in 1945 [Knu73]. The topics discussed here can also be found in [Eri19, §1.4–1.7], [DPV08, §2.3], [KT06, §5.1–5.2], [Blu11b,

[6] Here we use "heuristics" to refer to any techniques that (may be very useful but) do not theoretically improve the worst-case running times.

Chap. 2 and 5], and [Knu73]. Our discussion assumed some comfort with the subtle and important notion of *induction*. We discuss induction at greater length in Chapter 2.

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- (Re-)Recorded video lecture.

1.6 Exercises

Exercise 1.1. Use recursion trees to solve the following recurrences.

1. $T(n) = 3T(n/3) + 6n$ and $T(1) = 2$.
2. $T(n) = 2T(n/3) + 4n$ and $T(1) = 7$.
3. $T(n) = 4T(n/3) + n$ and $T(1) = 11$.

Exercise 1.2. Recall that binary search can find one element out of a sorted list of elements in $O(\log n)$ time. (Assuming the elements are arranged in, say, an array, so that we can access the ith element in constant time.) Here we will try to obtain a matching lower bound.

The general problem gives as input a list of n elements $(x_1, \ldots, x_n)$ in sorted order and with constant time access for any index, as well as an additional element k that acts as a key. For simplicity we may assume that k is promised to be one of the elements. The goal is to identify an index i such that $x_i = k$. We restrict ourselves to a comparison-only model where we may ask comparison queries such as "is $k < x_i$?", "is $k \leq x_i$?", "is $k = x_i$?", and so forth. Prove a lower bound of $\Omega(\log n)$ comparison queries (in the worst case) for any search algorithm that correctly finds k in the sorted list $(x_1, \ldots, x_n)$.

Exercise 1.3. Problems 1–5 describe a search problem on a nonempty array $A[1..n]$ of comparable elements.[7] For each of these problems, design a recursive algorithm that solves the search problem as fast as possible.[8] Briefly analyze the running time of each algorithm.[9]

1. Let $A[1..n]$ be in a *rotated* sorted order. That is, there exists an index $i \in [n]$, called the *rotation index*, such that the concatenation of $A[i..n]$ and $A[1..i-1]$ is sorted in increasing order. (One can think of $A[1..n]$ as being sorted initially, and then rotated cyclically to the right by $i - 1$ slots). The goal is to compute the unknown rotation index i. For simplicity, you may assume all the elements are distinct.

2. We say that $A[1..n]$ is a *mountain* if there exists an index $i \in [n]$, called the *peak*, such that $A[1..i]$ is sorted in increasing order and $A[i+1..n]$ is sorted in decreasing order. Given a mountain $A[1..n]$, find the peak. For simplicity, you may assume all elements are distinct.

3. We say an index $i \in [n]$ is a *local minimum* if either (a) $i = n = 1$, (b) $i = 1$ and $A[1] \leq A[2]$; (c) $i = n$ and $A[n] \leq A[n-1]$; or (d) $1 < i < n$, $A[i-1] \geq A[i]$ and $A[i] \leq A[i+1]$. The goal is to find a local minimum from an array $A[1..n]$.

4. Let $A[1..n]$ be an array of integers such that $A[1] = 0$ and $A[n]$ is odd. Call a consecutive pair of indices $(i, i+1)$ a *gap*. We say that a gap is *even* if the difference $A[i+1] - A[i]$ is even and *odd* if the difference is odd. Observe that when $A[1] = 0$ and $A[n]$ is odd, there must be an odd gap. Given an array of integers $A[1..n]$ such that $A[1] = 0$ and $A[n]$ is odd, find an odd gap.

[7] Given two elements x and y, you can query if $x < y$, $x > y$, or $x = y$.

[8] This includes a recursive spec, and pseudocode implementing the recursive spec.

[9] For the sake of brevity we do not ask for an explicit analysis of correctness. We will treat your recursive spec as the induction hypothesis of a proof by induction, and fill in the rest.

5. Let $A[1..n]$ be an array of integers such that $A[1] = 1$ and $A[n] = n$. For $i \in \{1, \ldots, n-1\}$, we say that there is a *jump* at index i if $A[i+1] \geq A[i] + 1$. Observe that since

$$\sum_{i=1}^{n-1} (A[i+1] - A[i]) = A[n] - A[1] \geq n - 1,$$

there must be a jump at least one index i. (If there was no jump, then the sum on the left would less than $n-1$, a contradiction.)

Design and analyze an algorithm that, given an array $A[1..n]$ such that $A[1] = 1$, $A[n] = n$, and $n \geq 2$, finds an index i for which there is a jump.

6. Prove that each of the problems above have a lower bound of $\Omega(\log n)$ queries in the comparison model.

7. (Extra credit / just for fun) Invent your own fun and interesting search problem on an array of comparable elements.

Exercise 1.4. Let $A[1..n]$ be an array of integers in the set $\{1..n\}$, sorted in increasing order. (There may be repeats.) A *fixed point* is an index $i \in [n]$ such that $A[i] = i$.

1. (2 pt.) Prove the following by induction on $j - i$: *for all pairs of indices i, j such that $1 \leq i \leq j \leq n$, $i \leq A[i]$, and $A[j] \leq j$, there is a fixed point k in the range $i \leq k \leq j$.*

   This claim shows in particular that A has a fixed point.

2. (2 pt.) Moving onto developing algorithms, give a recursive spec for the problem of finding a fixed point in A.[10]

3. (2 pt.) Give a recursive algorithm (the faster the better) implementing your recursive spec.

4. (2 pt.) Prove your algorithm is correct by induction. (There may or may not be some redundancy in your argument with part 1. You may also use the mathematical fact established in part 1.)

5. (2 pt.) Analyze the running time of your algorithm.

[10] There may be more than one fixed point; any fixed point is fine.

Exercise 1.5. Recall that given a sorted array $A[1...n]$ of comparable elements, we can find an element x (i.e., identify an index i such that $A[i] = x$, or declare that x does not exist) in $O(\log n)$ time with binary search. Here we develop an extension that in some sense allows us to do binary search on infinite length arrays. (A situation which does arise naturally.)

For simplicity, we assume the elements in A are distinct. For fixed A, and an element x, we say that x has *rank k in A* if x would be the kth smallest element if it were an element in A. (If x is already in A, then this is the unique index k such that $A[k] = x$.)

The goal is design and analyze an algorithm with the same interface as binary search: given access to a sorted array $A[1..n]$, and an element x either (a) returns the unique index i such that $A[i] = x$, or (b) declares that x is not in A. This time, the running time should take $O(\log k)$ time, where k is the rank of x.

Note that $O(\log k) \leq O(\log n)$, but it may be significantly faster when n is much larger than k, or even infinite.

Exercise 1.6. Let $A[1..m]$ and $B[1..n]$ be two sorted arrays of comparable elements. Given an integer $k \in \{1, \ldots, m + n\}$, the goal is to find the kth largest element among the combined elements of A and B. For simplicity you make assume that all the elements are distinct.[11]

1. Define a recursive spec for a recursive algorithm to solve this problem.
2. Show that if either m or n is ≤ 5 then we can find the kth element in $O(1)$-time by direct means. (This is to help declutter your implementation and avoid fencepost errors.)
3. Give a recursive algorithm implementing your recursive spec. You may simply refer to the previous part for your base case.
4. Prove your algorithm is correct by induction.
5. Analyze the running time of your algorithm.

Exercise 1.7. Let $A[1..n] \in \mathbb{R}^n$ be an array of n numbers. An *inversion* is a pair of numbers out of increasing order; more precisely, a pair of indices $i, j \in [n]$ such that $i < j$ and $A[i] > A[j]$. Design and analyze an algorithm (as fast as possible) for counting the number of inversions in A.

[11] By assuming standard data structures keeping track of the offset and length, you can get (a pointer to) the subarray $A[i..j]$ of an array $A[1..m]$ in $O(1)$ time.

Exercise 1.8. Two points $(a, b), (c, d) \in \mathbb{R}^2$ are *comparable* if either (a) $a \leq c$ and $b \leq d$, or (b) $c \leq a$ and $d \leq b$.

Given a set of n points $(a_1, b_1), \ldots, (a_n, b_n) \in \mathbb{R}^2$, consider the problem of computing the number of pairs of points that are comparable. You may assume for simplicity that all points are distinct. Design and analyze an algorithm (as fast as possible) for counting the number of comparable pairs of points.[12]

Exercise 1.9. Let $A[1..n] \in \mathbb{R}^n$ be an array of n comparable elements. Let $k \in \mathbb{N}$ be an input parameter where $k \leq n$. (We are particularly interested in the case where k is much smaller than n). Consider the problem where we want to obtained a sorted list of the k smallest elements.

1. Design and analyze an algorithm (as fast as possible) that returns the k smallest numbers in A in sorted order.

2. Prove a lower bound (large as possible, up to constants) on the number of comparisons required by any correct algorithm.

Food for thought: How does your lower bound compare to your upper bound obtained in exercise 1.9?

Exercise 1.10. Let $G = (V, E)$ be a directed graph with m edges and n vertices. Recall that for $s, t \in V$, s can *reach* t if there is a path from s to t. There are many efficient algorithms to figure out if s can reach t, that run in $O(m + n)$ time but also use $O(n)$ space.

Suppose we want to develop an algorithm that uses polylogarithmic space (but may take more than polynomial time). Consider the following recursive specification:

> `reach(`s`,`t`,`k`)`: Given two vertices $s, t \in V$ and $k \in \mathbb{N}$, returns true if s can reach t in at most k edges.

Design and analyze a recursive implementation of `reach` that runs in subexponential time and uses polylogarithmic space.

Exercise 1.11. Let T be a tree on n vertices. A *centroid* of T is a vertex v whose removal breaks T into subtrees of at most $n/2$ vertices. A tree can have up to two centroids.

[12] *Hint:* As a warmup, try to instead compute the number of *incomparable* pairs of points. (A pair of points is incomparable if is not comparable. E.g., (x_1, y_1) and (x_2, y_2) are incomparable if $x_1 < x_2$ and $y_1 > y_2$.) You may want to do exercise 1.7 before this one.

1. Give a linear time algorithm taking as input a tree and returning a centroid. Your proof of correctness doubles as a proof that every tree has a centroid.

Centroids can be used as a sort of "pivot" in divide-and-conquer algorithms over trees. Design and analyze an algorithm for each of the following problems, leveraging centroids and recursion.[13]

2. Design and analyze an algorithm that computes, for a given $k \in \mathbb{N}$, the number of pairs of vertices that are k edges apart in T.

3. Let each edge by labeled by a fixed-length bit strength (of the same length). Assuming we can compute the XOR of two bit strings in $O(1)$ time, design and analyze an algorithm that, given a string x, computes the number of paths in T where the XOR over the edges in the path equals x.

4. Under the same setting as the previous problem, design and analyze an algorithm that maximizes the XOR over its labels (where each bit string is interpreted as an integer encoded in binary).

For the following, let the edges of T be labeled by "lengths" $\ell(e)$. The distance between two vertices is the sum of edge lengths over the unique path between the two vertices in T.

5. Design and analyze an algorithm that computes, for each vertex v, the sum of distances to all other vertices.

6. Design and analyze an algorithm that computes the diameter of T; i.e., the pair of vertices of maximum distance in T.

7. Define the *radius* of T as the minimum, over all vertices $v \in T$, of the maximum distance from v to any other vertex. The vertex v minimizing the maximum distance is called the "1-median" Design and analyze an algorithm computing the radius of T and the 1-median.

8. Design and analyze an algorithm that, given values $a, b \in \mathbb{R}$ with $a \leq b$, counts the number of pairs with distance in the interval $[a, b]$

9. Design and analyze an algorithm that counts the number of ordered pairs (u, v) where the edge weights along the (u, v)-path are (strictly) monotonically increasing.

[13] A couple of these problems can be solved by other means; for pedagogical reasons, we still encourage you to solve them via centroids.

10. Design and analyze a data structure that, after preprocessing T, can answer "distance queries" that take as input two vertices s, t and output the distance between s and t.

11. Suppose that the edges have both lengths $\ell(e) \in \mathbb{R}$ and costs $c(e) \in \mathbb{R}$. Design and analyze an algorithm that, given $C, D \in \mathbb{R}$, counts the number of pairs of vertices u, v with distance at most D and where the cost of the (u, v)-path is at most C.

Exercise 1.12. Peter Venkman, Ray Stantz, and Egon Spengler were three professors at Columbia researching paranormal activity, broadly construed. After their first encounter with a ghost at the New York Public Library, the university dean dismisses the credibility of their paranormal-focused research and fires them. The trio responded by establishing "Ghostbusters", a paranormal investigation and elimination service operating out of a disused firehouse. They develop high-tech nuclear-powered equipment to capture and contain ghosts. Business was initially slow but after a ghost epidemic they suddenly need to scale up their operation. Peter, Ray and Egon need to build a ghostbuster army.

After placing some television ads, and thanks in part to their catchy theme song, they were able to assemble n recruits to their ghostbuster army. Now, to be in the ghostbuster army, you can't be afraid of no ghosts, and you definitely can't be afraid of the dark. To test the latter, Peter, Ray and Egon put all n recruits in a room and turned off the lights. One of the recruits screamed. They turned the lights back on. Apparently one of the recruits is afraid of the dark.


Peter, Egon, and Ray

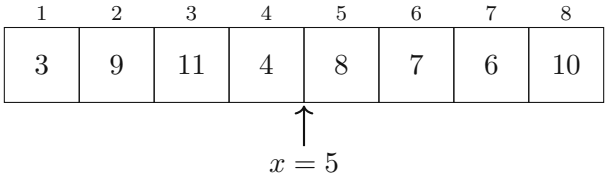
The ghostbusters need to figure out who in the army is afraid of the dark. Nobody wants to admit they are afraid of the dark. But we can test if someone is afraid of the dark by putting them in a room, and turning off the lights. A person screams if and only if they are afraid of the dark.

Of course we can identify the scaredy-cat by testing each recruit one by one. This can take a long time. The ghostbusters have other options: they can place a bunch of recruits in the dark, and by turning off the lights, figure out if the scaredy-cat is among them.

Let's say that it takes "one round" to put a set of recruits in the room, turn off the lights, and see if any of the recruits are scared of the dark. Design and analyze an algorithm that identifies a recruit that's scared of the dark in the minimum number of rounds.

Exercise 1.13. Let $A[1..n]$ be an array where each entry is colored `red` or `blue`, such that $A[1]$ and $A[n]$ have different colors. (This also implies that $n > 1$.) We say that a pair of consecutive indices $(i, i+1)$ is *colorful* if $A[i]$ and $A[i+1]$ have different colors. Design and analyze an algorithm (the faster the better) that computes a colorful consecutive pair of indices $(i, i+1)$ in A, or `nil` if there is no colorful pair of indices.

Exercise 1.14. Let $A[1..n]$ be an array of numbers, and x a number, such that $A[1] \leq x < A[n]$. (This also implies that $n > 1$.) We say that an index i "brackets" x if $A[i] \leq x < A[i+1]$. In the array below, the index $i = 4$ brackets $x = 5$ since $A[4] = 4 \leq 5 < 8 = A[5]$.

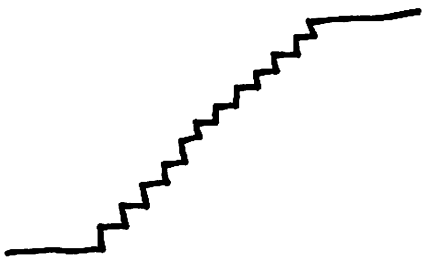


Consider the problem of computing either an index i that brackets x, or `nil` if no such index exists. Design and analyze an algorithm for this problem.

Chapter 2

Induction and recursion

2.1 One step at a time



Imagine a long staircase with, say, 1000 stairs. Someone asks you, how do I get from the bottom to the top? And you might respond:

> *Let's take it one step at a time. Say you're at a particular step. Now put one foot on the next step, and use that leg to push your body up and put the other foot on that step, so now your entire mass is on the next step. You can use either foot for the first step. Now if you keep repeating this for each step you will eventually get to the top.*

The key part is the beginning: *Let's take it one step at a time. Say you're at a particular step.* Immediately you recognize that each step is identical, and it suffices to give instructions for a generic step. It would be excessive to say, "For the first step, do bla bla bla. Then for the second step, do bla bla bla. For the third step, do bla bla bla, ..." and so forth, giving detailed instructions for every step.

Can we prove this for any staircase?

Claim: *For any $n \in \mathbb{N}$, the instructions above will scale a staircase of n steps.*

The argument must work for all n at once — not a separate proof for every value of n, getting longer and longer as n grows. But if we assumed the claim for $n-1$

steps, the extension to n steps would be easy. *Induction* lets us make exactly this assumption, for any n.

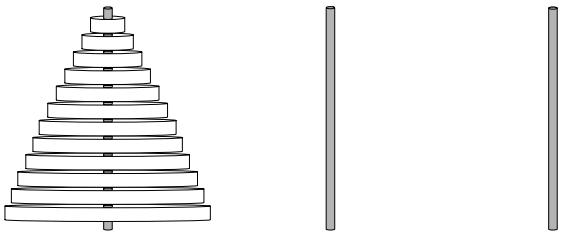
The one exception is $n = 1$, where there is no "$n - 1$" to build on. This is the *base case*, proved directly.

Base case: $n = 1$. Follow the instructions once and climb one stair. The staircase is complete.

General case: $n > 1$. Follow the instructions and climb one stair. The remaining staircase has $n-1$ steps. *By induction on* n, the instructions work for any $(n-1)$-step staircase. So the instructions work for n steps as well.

This may be obvious for staircases. Less so for the problems ahead.

2.2 Towers of Hanoi



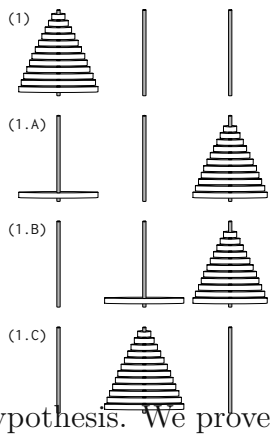
In the legend of the towers of Hanoi, there are three posts, which we call post A, post B, and post C. On post A, there is a stack of n rings of increasing radii from top to bottom. (In the story, $n = 64$.) We are allowed to move one ring at a time from the top of one post to the top of another post. Our only constraint is that each post must remain in order: a ring cannot be placed on top of another ring with smaller radius.

Can you move all the rings from post A to post B? (Try it.)

Algorithm. The first step is a *recursive spec*:

> `Towers-of-Hanoi(`n`,`A`,`B`,`C`)`: Assuming the n smallest rings are on post A, moves the top n rings from post A to post B (while respecting the ordering rules of the towers of Hanoi).

We now implement `Towers-of-Hanoi` (abbr. `ToH`). In the base case, $n = 0$, and there are no rings to move. So `ToH(0,`A`,`B`,`C`)` does not do anything.



Let $n > 0$. When implementing the algorithm recursively, we assume `ToH` fulfills its specification for any smaller value of n.

Now, to move the top n rings from A to B, we will need to move the nth ring (from the top) from A to B. To do that we need to move the top $n-1$ rings out of the way, from A to C. How do we do that? Easy: `ToH`$(n-1, A, C, B)$! Then move the nth ring from A to B, and call `ToH`$(n-1, C, B, A)$ to move the rest to B.

That's the entire algorithm. The general case is three lines. See Fig. 2.1 on the following page for pseudocode.

Proof of correctness. The recursive spec *is* the inductive hypothesis. We prove `Towers-of-Hanoi` fulfills its spec by induction on n.

The base case, $n = 0$, is immediate, as there are no rings to move, and the algorithm does nothing.

Let $n > 0$. When we first call `ToH`$(n-1, A, C, B)$, the $n-1$ smallest rings are on post A, as required. By induction on n, `ToH`$(n-1, A, C, B)$ moves the top $n-1$ rings from A to C. Then we move the top ring from A to B, putting the nth smallest ring on B. Before we call `ToH`$(n-1, C, B, A)$, the $n-1$ smallest rings are on post C, as required. By induction on n, `ToH`$(n-1, C, B, A)$ moves the $n-1$ smallest rings from C to B, where they sit on top of the nth smallest ring. At termination, the n smallest rings are now on post B, as desired. This completes the proof.

Optimal towers of Hanoi. We have *an* algorithm. Is it the *best* algorithm? How do you solve the Towers of Hanoi with the *minimum* number of moves? (Try it.)

We don't need a new algorithm. The `Towers-of-Hanoi` algorithm already makes the minimum number of moves. We just strengthen the recursive spec to prove it.

> `Towers-of-Hanoi`(n, A, B, C): Assuming the smallest n rings are on post A, moves the top n rings from post A to post B *with the minimum number of ring moves.*

Let T_n be the number of ring moves made by `ToH` on n rings. We claim that T_n is optimal for all n, by induction on n.

The base case is $n = 0$. Here $T_0 = 0$ which is clearly optimal.

Suppose $n > 0$, and we are moving n rings from post A to post B. `ToH` makes $T_n = 2T_{n-1} + 1$ ring moves; we want to show that this is optimal.

```
Towers-of-Hanoi(n, A, B, C)
/* Assuming the n smallest rings are on post A, move the top n rings from post A to post B. */
1. If n > 0:
   A. Towers-of-Hanoi(n − 1, A, C, B).
   B. Move the top ring from A to B.
   C. Towers-of-Hanoi(n − 1, C, B, A).
```

Figure 2.1: Recursively solving the towers of Hanoi problem.

The nth ring can be moved from A to B only if the top $n-1$ rings are on post C. Thus any algorithm must (at least) move the top $n-1$ rings from A to C, move the nth ring from A to B, and move the $n-1$ rings back from C to A. By induction on n, T_{n-1} equals the minimum number of ring moves to move $n-1$ rings from one post to another. So any algorithm must make at least $2T_{n-1}+1 = T_n$ ring moves. Thus T_n is the minimum number of ring moves.

2.3 Greatest common divisor

For two integers $x, y > 0$, the *greatest common divisor* (GCD) of x and y is the largest positive integer that divides both x and y. How do you compute the GCD?

Suppose $x \geq y$. If q divides x and y, then q also divides $x - y$. Conversely if q divides y and $x - y$, then q also divides $x = y + (x - y)$. Thus the greatest common divisor of x and y is the greatest common divisor of y and $x - y$.

Now how do we compute the greatest common divisor of y and $x - y$? Recurse!

First, the recursive spec. We assume $x \geq y$ and $x > 0$ for convenience.

> `GCD(x,y)`: Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.

Then the implementation.

Lemma 2.1. *Given two nonnegative integers $x, y \geq 0$ with $x \geq y$ and $x > 0$,* `GCD(x,y)` *of Fig. 2.2 computes the greatest common divisor of x and y.*

`GCD(`x, y`)`

/* *Given two positive integers $x, y \in \mathbb{Z}_{\geq 0}$ with $x \geq y$ and $x > 0$, returns the greatest common divisor of x and y.* */

1. If $y = 0$ then return x.
2. Return `GCD(`$\max\{x - y, y\}, \min\{x - y, y\}$`)`.

Figure 2.2: Euclid's algorithm for computing the greatest common divisor.

Proof. We prove the claim by induction on $x + y \in \mathbb{N}$.

In the base case $x + y = 1$. Then we must have $x = 1$ and $y = 0$, and the greatest common divisor is 1. This matches the output of `GCD(1,0)`.

Suppose $x + y > 1$. If $y = 0$ then x is the greatest common divisor of x and y, matching the output of `GCD(`x, y`)`.

Now suppose $y > 0$. Let $x' = \max\{x - y, y\}$ and $y' = \min\{x - y, y\}$. The algorithm returns `GCD(`x', y'`)`. As argued above, the greatest common divisor of x and y is the greatest common divisor of x' and y'. Since $x' + y' < x + y$, by induction, `GCD(`x', y'`)` returns the greatest common divisor of x' and y', which is also the greatest common divisor of x and y, as desired. □

Running time analysis. What is the running time of `GCD(`x, y`)`? Each recursive call decreases $x + y$ by at least 1, and the minimum value of $x + y$ is 1. So the running time is $O(x + y)$ (treating subtraction as a constant time operation).

Is $O(x + y)$ a good running time? It looks like a linear running time. However, it takes $O(\log x)$ bits to express x and $O(\log y)$ bits to express y, so the input size is really $O(\log(x) + \log(y)) = O(\log(x + y))$. In fact, $O(x + y)$ is *exponential* in the bit complexity of x and y.

A faster algorithm. To speed up `GCD(`x, y`)`, consider a bad case that realizes the $O(x + y)$ running time. For example, suppose $y = 2$, and x is a very large odd number; say, $x = 1001$. The algorithm repeatedly subtracts 2 from x until x becomes 1, taking $O(x/2)$ iterations in all.

In that example, $1 = x \mod y$. For general $x \geq y > 0$, the algorithm repeatedly subtracts y from x until we reach $x \mod y$.

Observe that if $r = x \mod y$, then q divides both x and y if and only if (abbr. iff) q divides r and y. This follows from the fact that $x = ky + r$ for some integer k such that $ky \leq y$. Thus r and y have the same greatest common divisor as x and y. This gives the following faster implementation for `GCD(`x, y`)` (as specified above):

```
GCD(x,y)
/* Given two positive integers x, y ∈ Z_{≥0} with x ≥ y and
   x > 0, returns the greatest common divisor of x and y. */
1. If y = 0 then return x.
2. Return GCD(y, x mod y).
```

Figure 2.3: A faster algorithm for computing the greatest common divisor.

First, is the new algorithm correct?

Lemma 2.2. *Given two nonnegative integers $x, y \geq 0$ with $x \geq y$ and $x > 0$,* `GCD(`x`,`y`)` *of Fig. 2.3 computes the greatest common divisor of x and y.*

Proof. We prove the claim by induction on $y \in \mathbb{N}$.

If $y = 0$ then x is the greatest common divisor of x and y, matching the output of `GCD(`x`,`y`)`.

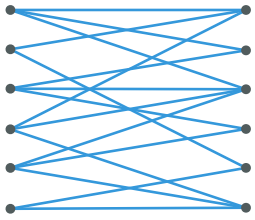
In the general case $y > 0$. Let $r = x \mod y$. The algorithm returns `GCD(`y`,`r`)`. As argued above, the greatest common divisor of x and y is the greatest common divisor of y and r. We also have $r < y$. By induction on y, `GCD(`y`,`r`)` returns the greatest common divisor of y and r, hence the greatest common divisor of x and y. □

Now for the running time analysis. If $y = 0$ then of course the running time is constant. Let $x \geq y > 0$, and let $r = x \mod y$. If $r \leq y/2$, then this iteration divides y in half. If $r \geq y/2$, then in the next iteration, $y \mod r \leq y/2$. Thus every two iterations divides the y parameter by a factor of 2. Thus the algorithm runs in $O(\log y)$ iterations.

If we treat "$x \mod y$" as a constant time operation, then the algorithm takes $O(\log y)$ time. Otherwise, $x \mod y$ can be computed in $O(\log(x)\log(y))$ time by long division. This gives a $O\Big(\log(x)\log^2(y)\Big)$ running time overall. Either way, we now have a running time that is polynomial in the input size, rather than exponential.

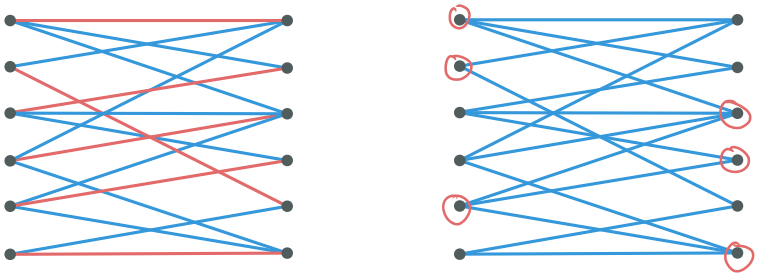
2.4 Bipartite matchings and vertex cover

A graph $G = (V, E)$ consists of points V, called *vertices*, and point-to-point connections E, called *edges*. In undirected graphs, edges model symmetric connections between the vertices, and are represented by unordered pairs of vertices like $e = \{u, v\}$. In directed graphs, edges are "directed" from one vertex (the tail) to another (the head), and are represented as ordered pairs (u, v).



Let $G = (L \cup R, E)$ be an undirected bipartite graph. "Bipartite" means that the vertices are partitioned into two sets, L (for "left") and R (for "right"), and every edge e has exactly one endpoint in L and one endpoint in R.

A *matching* is a set of edges $M \subseteq E$ whose endpoints are disjoint. (M "matches" vertices on the left to vertices on the right.) A *vertex cover* is a pair of vertex sets (S, T), where $S \subseteq L$ and $T \subseteq R$, such that every edge has an endpoint in S or T (or both).



Consider the following two problems:

1. *Maximum matching*: compute the maximum cardinality matching M.
2. *Minimum vertex cover*: compute the minimum cardinality vertex cover (S, T).

For any matching M, and any vertex cover (S, T), each edge $e \in M$ has at least one endpoint in $S \cup T$. Moreover, since M is a matching, each of these endpoints are disjoint. Thus

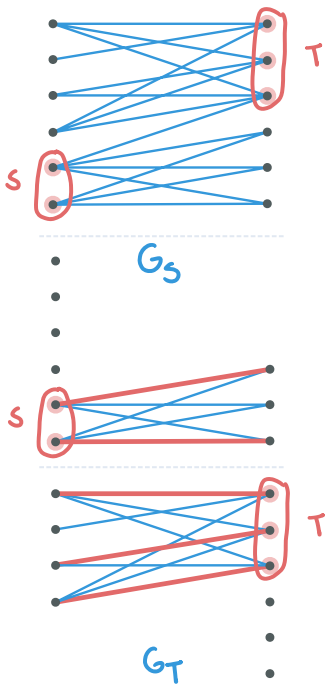
$$|M| = \sum_{e \in M} 1 \leq \sum_{e \in M} |e \cap (S \cup T)| \leq |S \cup T|.$$

This is *weak duality*: any matching is at most any vertex cover. Does equality hold? That is, is the maximum matching always equal to the minimum vertex cover?

Theorem 2.3 (Kőnig's theorem)**.** *In any bipartite graph G, the maximum size of any matching equals the minimum size vertex cover.*

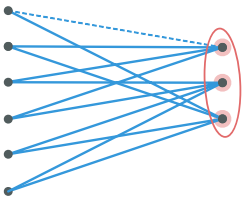
Proof We define the "size" of a graph G, denoted $|G|$, as the number of vertices plus the number of edges, $|G| = |V| + |E|$. We prove the claim by induction on the size of the graph. In the base case, a graph of size 0 has no vertices or edges, and the claim is immediate.

Consider a bipartite graph $G = (L \cup R, E)$ of nonzero size N, and assume by induction that the claim holds for all smaller graphs. Let k be the size of the minimum vertex cover in G; we claim there exists a matching of size k.



Suppose there exists a vertex cover (S, T), of size $|S| + |T| = k$, such that neither S nor T is empty. Let G_S be the graph obtained by removing T and all the edges incident to T. S is a vertex cover in G_S. Moreover, any vertex cover (A, B) (where $A \subseteq L$ and $B \subseteq R \setminus T$) in G_S gives a vertex cover $(A, B \cup T)$ in G. Since (S, T) is the minimum vertex cover in G, we have $|A| + |B| \geq |S|$ for any vertex cover in G_S. That is, $(S, \emptyset)$ is a minimum vertex cover in G_S. Since G_S is smaller than G, by induction, there exists a matching M_S in G_S of size $|S|$. Note that M_S is covered by S and does not have any edges incident to T.

Symmetrically, there exists a matching M_T of size $|T|$ that is covered by T and does not have any edges incident to S. Then $M = M_S \cup M_T$ gives a matching in G size k, as desired.



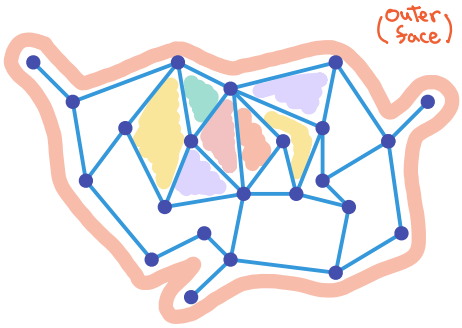
Otherwise, suppose every vertex cover (S, T) of size k has either S or T empty. Let $e = \{u, v\} \in E$ (where $u \in L$ and $v \in R$) be any edge, and let $G' = (V, E')$, where $E' = E - e$, be the graph obtained by removing e. If G' has a vertex cover (A, B) of size $|A| + |B| < k$, then both $(A + u, B)$ and $(A, B + v)$ are vertex covers of G of size at most k, and at least one of these two vertex covers has both parts nonempty – a contradiction. Thus every vertex cover in G' also has size at least k, and in particular, (S, T) is a minimum vertex cover of G'. By induction on the size of the graph, there exists a matching M in G' of size k; clearly, M is the desired matching in G of size k. □

Matching and vertex cover are an instance of *packing and covering duality*. A maximum matching and minimum vertex cover of the same size *mutually certify* each other's optimality. Duality will be a theme later in the book.

2.5 Induction on planar graphs

Working again in undirected graphs, we now allow the endpoints of an edge e to be equal; then e is called a *self-loop* or just a *loop*. Graphs are drawn as dots (vertices) connected by lines (edges).

A *planar graph* is a graph that can be drawn in the plane so that the edges intersect only at their endpoints. Such a drawing is called an *embedding* of the graph in the plane. The graph on the right is an example of a planar graph.



Not all graphs are planar. For example, the complete graph on 5 vertices (with all $\binom{5}{2}$ distinct edges) is not a planar graph.

Let G be a planar graph and fix an embedding of G in the plane. The vertices and edges of a planar graph divide the plane into regions, called *faces*. The unbounded region is called the *outer face*; the vertices and edges bounding the outer face are called the *outer boundary* of G.

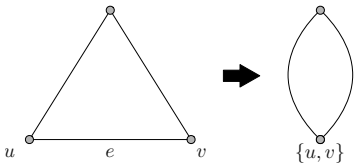
Planar graphs are very special. Consider *Euler's formula*.

Theorem 2.4 (Euler's formula)**.** *For any connected planar graph G with m edges, n vertices, and p faces,*

$$n - m + p = 2.$$

Proof. We prove the claim by induction on the number of vertices, n. In the base case, $n = 1$. Then every edge is a self-loop, enclosing an interior face, so $p = m + 1$. Then $n - m + p = n + 1 = 2$, as desired.

Suppose $n > 1$. Since the graph is connected, G has at least one edge $e = \{u, v\}$. Let H be the graph obtained by contracting e to a single vertex. This has the effect of decreasing the number of edges and the number of vertices by 1. The number of faces remains the same. By induction on the number of vertices, Euler's formula holds for H, which gives



$$(n - 1) - (m - 1) + p = 2.$$

The LHS simplifies to $n - m + p$, as desired. □

Corollary 2.5. *For any simple planar graph (with no self-loops or parallel edges), with m edges and $n \geq 3$ vertices, $m \leq 3n - 6$.*

Proof. Let p be the number of faces. Every face is bounded by at least three edges, and each edge is on the boundary of at most 2 faces. Thus $3p \leq 2m$. Plugging this into Euler's formula gives

$$n - m + (2/3)m \geq 2.$$

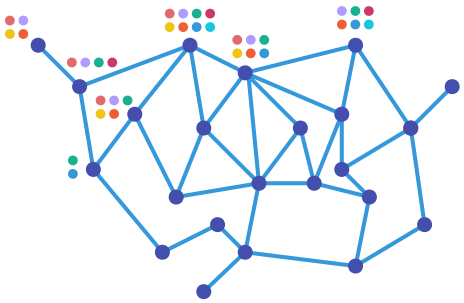
Rearranging, we have $m \leq 3n - 6$ □

List-coloring planar graphs with 6 colors. Fix a graph $G = (V, E)$. For $k \in \mathbb{N}$, a *k-coloring* is an assignment $\pi : V \to [k]$ such that for any edge $e = \{u, v\} \in E$, $C(u) \neq C(v)$.[1] We can think of $[k]$ as representing k different colors, and $\pi : V \to [k]$ as "coloring" each vertex one of the k colors. Then π is a feasible k-coloring if there is no monochromatic edge; i.e., the endpoints of every edge have different colors.

Given a graph $G = (V, E)$ and $k \in \mathbb{N}$, is there a k-coloring of G?

The famous *four color theorem* states that any planar graph can be colored with 4 colors. The theorem is notorious for its long history of failed proofs. The theorem was finally proven by [AH77], but with a catch: Appel and Haken needed the help of a computer to enumerate a large number of special cases. Consequently the proof was largely incomprehensible. It remains open to obtain a proof without the help of a computer.

A related problem is *list coloring*. Here each vertex v is associated with a set of colors $C(v) \subset \mathbb{N}$. The goal is to color the vertices $\pi : V \to \mathbb{N}$ so that (a) $\pi(v) \in C(v)$ for all v, and (b) there is no monochromatic edge. A graph $G = (V, E)$ is said to be *k-list colorable* if for any family of lists $\{C(v) : v \in V\}$ such that $|C(v)| \geq k$ for all v, there exists a feasible list-coloring of G.



Theorem 2.6. *Every planar graph is 6-list colorable.*

Proof. We may assume the graph is simple, since self-loops and parallel edges have no effect on coloring.

In any planar graph with at least 3 vertices, by Euler's formula, the average degree is bounded above by

$$\frac{2m}{n} \leq \frac{6n - 12}{n} < 6.$$

Since the average degree is less than 6, the minimum degree is less than 6; that is, at most 5.

Now, consider any planar graph G. If G has fewer than 3 vertices then certainly it can be colored with 6 colors. Suppose G has at least 3 vertices. Then there is a vertex v with degree ≤ 5. Let H be the graph obtained by removing v and all incident edges. H has one fewer vertex than G. By induction on the number of vertices, there is a 6-coloring of H. At most 5 of the 6 colors are taken by neighbors of v. Thus

[1] Here $[k] \stackrel{\text{def}}{=} \{1, \ldots, k\}$.

the 6-coloring on H can be extended to a 6-coloring on G by assigning v an available color. □

One can convert the proof into a linear time algorithm; see exercise 2.4.

List-coloring planar graphs with 5 colors. The proof, due to [Tho94], strengthens the induction hypothesis.

Theorem 2.7. *All planar graphs are 5-list colorable.*

Theorem 2.7 follows from the following lemma by arbitrarily coloring any two consecutive vertices along the outer boundary.

Lemma 2.8. *Let $G = (V, E)$ be a planar graph, and let $B = (V_B, E_B)$ be the boundary of G.*

(a) Two vertices $x, y \in V_B$ adjacent in B are already colored α and β, respectively.

(b) $|C(v)| \geq 3$ for all $v \in V_B$.

(c) $|C(v)| \geq 5$ for all $v \in V \setminus V_B$.

Then the coloring of x and y can be extended to a proper list coloring of G.

Proof. We prove the claim by induction on $|V|$.

In the base case, $|V| = 3$. The remaining uncolored vertex v has $|C(v)| \geq 3$, so there is a color available for v.

Consider the case $|V| > 3$. We assume by induction on $|V|$ that the claim holds for $|V| - 1$ (or fewer) vertices. Let B be the boundary of G. We have a few cases.

Case 1. Suppose G has a leaf v along its outer boundary. Let H be the graph obtained by removing v. H has one fewer vertex, and the outer boundary of H is the outer boundary of G except without v.

Suppose first that v is not either x or y. H now meets the hypothesis of the claim with one fewer vertex. By induction on the number of vertices, there is a proper list-coloring of H extending the coloring on x and y. Since $|C(v)| \geq 3$, and v has only one neighbor, there is an available color to extend the coloring from v to G.

Now suppose $v = x$ or $v = y$. Without loss of generality we assume $v = x$. Choose another vertex x' adjacent to y, and assign it an available coloring α'. H, x', and y now meet the hypothesis of the claim with one fewer vertex. By induction, there is a proper list coloring of H extending the colors assigned to x' and y. Again, since $|C(v)| \geq 3$, and v has only one neighbor, there is an available color to extend the coloring from v to G.

Case 2. Suppose G has no leaf along its outer boundary B. Then B is a cycle. Here we have two sub-cases.

Case 2.a. Suppose B has a chord.[2] Split B into two sides B_1 and B_2 meeting at u and v. Let G_1 be the planar graph bounded by $B_1 \cup \{u, v\}$ and let G_2 be the planar graph bounded by $B_2 \cup \{u, v\}$. G_1 has fewer vertices than G, so by induction, there is a proper list coloring of G_1.

Now consider G_2, and assign u and v the two colors given by the list coloring G_1. By induction, the colors on u and v can be extended to a proper list-coloring of G_2.

Finally, combining the list-colorings of G_1 and G_2 give a list-coloring on G.

Case 2.b. Now suppose B does not have any chords. Let v be the vertex on the other side of x from y. Since $|C(v)| \geq 3$, $C(v)$ contains at least two additional colors γ, δ distinct from α.

The neighbors of v consist of x and another neighbor u on B, and a number of neighbors $t_1, \ldots, t_k$ in the interior of B. Let H be the graph obtained by removing v and all incident edges from the graph. Observe that $t_1, \ldots, t_k$ are all exterior vertices of H. Consider the list-coloring problem on H where we remove γ and δ from $C(t_i)$ for $i = 1, \ldots, k$. Then all vertices on the boundary of H have colors lists of size 3 or greater, and all interior vertices have color lists of size 5 or greater. By induction on the number of vertices, we can extend the coloring on x and y to a proper list coloring on H.

To then color v, observe that none of the vertices $t_1, \ldots, t_k$ are assigned γ or δ. The only other neighbors of v are u and x, where x is already assigned α. So either γ or δ (or both) is available to color v, which extends the list coloring to all of G. □

2.6 Additional notes and materials

See [Che15], [Eri19, Chapter 0], or [GKP94, Chapter 0] for more background on induction.

There are many different ways to prove Euler's formula; see [Epp]. Thomassen's proof [Tho94] is also described in [AZ18].

Spring 2026 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.

[2] Given a cycle C, a chord is an edge between two non-adjacent vertices of the cycle.

Spring 2025 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

2.7 Exercises

Exercise 2.1. For each of the following variations of the Tower of Hanoi problem, give an algorithm that solves the problem with the minimum number of ring moves. Recursive algorithms should have an explicit recursive spec. We will treat the recursive spec as an induction hypothesis to justify the correctness of the algorithm, and no explicit proof is otherwise required.[3] Non-recursive algorithms require a proof (which should be short).

You may assume as fact that the `Towers-of-Hanoi` algorithm (Fig. 2.1, on page 42) moves n rings from one post to another in the minimum number of steps. Also, your algorithms are allowed to be mutually recursive (i.e., can depend on one another) across subproblems.

1. In the *easy towers of Hanoi* problem, we remove the restriction that the rings always have to be in sorted order on each post. The goal is still to move n sorted rings from post A to post B, in sorted order on post B.

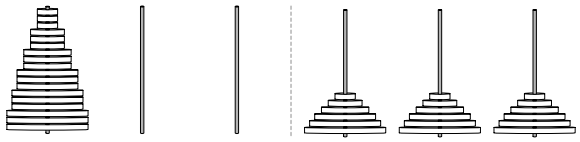
2. In the *cyclic towers of Hanoi* problem, a ring can only be moved in "cyclic" order from post A to post B, from post B to post C, and from post C to post A. The goal is to move n sorted rings from post A to post B.

3. The *double-cyclic towers of Hanoi* is like the cyclic towers of Hanoi problem except now the goal is to the n sorted rings from post A to post C.

[3] You may want to work out a full proof for yourself anyway for extra practice. Additional justification or comments might help the grader's understanding in the event of partial credit.

4. In the *thick towers of Hanoi* problem, we have $3n$ rings of n distinct sizes with three copies of each ring. (Rings of the same size can stack on top of each other.) The goal is to move all $3n$ rings from post A to post B.

5. In the *triple towers of Hanoi* problem, we have $3n$ rings of n distinct sizes with three copies of each ring. The goal is to distribute the rings so that each post has n rings, one of each size, in sorted order.



6. In the *American towers of Hanoi* problem, we have n rings with each ring colored red, white or blue. The three posts are also colored red, white, and blue. Initially all the rings are stacked (in order of size) on the red post. The goal is to stack all the red rings on the red post, the blue rings on the blue post, and the white rings on the white post (again, in order of size).

7. In the *messy towers of Hanoi* problem, the n rings are initially distributed arbitrarily over the three posts. (Each post is still in order.) The goal is to move all the rings to a single designated post.[4]

8. (Extra credit / just for fun) Invent a fun and interesting variant of the towers of Hanoi problem.

Exercise 2.2. For each of the recursive specifications below, design a recursive algorithm implementing the specification. (No proof or analysis is needed.)[5]

[4]You are allowed to look at the current configuration and identify which post has the nth smallest ring, for any given n. You may have different cases depending on which post has the nth smallest ring.

[5]Your algorithms will not be very fast, but their correctness should follow from an induction argument using the recursive spec as the induction hypothesis.

You are welcome to describe your algorithm using high level steps as long as they are clearly easy to implement and would take linear time; e.g., 'let G_1 be the graph obtained by removing such and such vertices and such and such edges." See the sample solutions on page 60.

Some of these problems have well-known efficient algorithms. Efficiency is besides the point; the goal here is to rapidly get some practice thinking recursively.

Lastly, do not change the recursive spec.

1. `subset-sum`$(x_1, \ldots, x_n,\ T)$: Given n integers $x_1, \ldots, x_n \in \mathbb{Z}$, and an additional integer T, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = T$, and `false` otherwise.

2. `min-vertex-cover`$(G = (V, E))$: Given an undirected graph $G = (V, E)$, return the minimum size of any vertex cover of G.

   (A vertex cover is a set of vertices $S \subseteq V$ that includes at least one endpoint of every edge $e \in E$.)

3. `max-matching`$(G = (V, E))$: Given an undirected graph $G = (V, E)$, return the maximum size of any matching.

   (A matching is a set of edges $M \subseteq E$ with disjoint endpoints.)

4. `max-independent-set`$(G = (V, E)$: Given an undirected graph $G = (V, E)$, return the maximum size of any independent set of vertices.

   (Given an undirected graph $G = (V, E)$, an *independent set* is a set of vertices $S \subseteq V$ such that no two vertices $u, v \in S$ are connected by an edge.)

5. `longest-increasing-subsequence-including-first`$(x_1, \ldots, x_n)$: Given a sequence of numbers $x_1, \ldots, x_n$, return the length of the longest (strictly) increasing subsequence of $x_1, \ldots, x_n$ over all subsequences that include x_1. (You may assume $n \geq 1$.)

   (A subsequence of $x_1, \ldots, x_n$ is a sequence of the form $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. A sequence of numbers $y_1, \ldots, y_\ell$ is strictly increasing if $y_i < y_{i+1}$ for $i = 1, \ldots, \ell - 1$.)

6. `reachable`$(G = (V, E), s, t)$: Given a directed graph G and two vertices $s, t \in V$, return `true` if s can reach t in G, and `false` otherwise.

   (s can reach t if either $s = t$ or there is a sequence of (directed) edges of the form $(s, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, t)$. Note that the endpoint of one edge is the initial point of the next edge. This is also called a *(directed) walk* in G from s to t.)

7. `partition`$(x_1, \ldots, x_n)$: Given n integers $x_1, \ldots, x_n \in \mathbb{Z}$, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = \sum_{i \in [n] \setminus S} x_i$, and false otherwise.[6]

[6]This might be the trickiest one. No, you cannot reduce to subset sum. You need to implement *this* recursive spec, recursively.

Exercise 2.3. Let $T = (V, E)$ by a rooted binary tree. Recall that a *full* binary tree is a binary tree where every non-leaf node has exactly. Design and analyze an algorithm that computes the maximum size of any full binary tree that is a subtree of T.

Exercise 2.4. Design and analyze a linear time algorithm for six-coloring a planar graph.

Exercise 2.5. A *partially ordered set* (or *poset*) is a set P with a binary relation $\leq$ that is reflexive, antisymmetric, and transitive. A *chain* is a set of pairwise comparable elements; an *antichain* is a set of pairwise incomparable elements. The *width* of a poset is the maximum size of an antichain.

1. Prove that any partition of P into chains requires at least w chains, where w is the width of P.
2. Suppose we want to partition P into as few chains as possible. A natural greedy approach would take a maximal chain C, remove it, and repeat. Define a poset where this greedy approach uses more than w chains.
3. Let A be a maximum antichain. Let $U = \{x \in P : x \geq a \text{ for some } a \in A\}$ and $D = \{x \in P : x \leq a \text{ for some } a \in A\}$. Prove that $U \cup D = P$ and $U \cap D = A$.
4. Prove that any poset of width w can be partitioned into exactly w chains. (This is called *Dilworth's theorem.*)

Exercise 2.6. Let $G = (L \cup R, E)$ be a bipartite graph. For a set $S \subseteq L$, let $N(S) = \{v \in R : \{u, v\} \in E \text{ for some } u \in S\}$ be the *neighborhood* of S. A matching M *covers* L if every vertex in L is an endpoint of some edge in M.

1. Prove that if G has a matching covering L, then $|N(S)| \geq |S|$ for all $S \subseteq L$.
2. Prove the converse: if $|N(S)| \geq |S|$ for all $S \subseteq L$, then G has a matching saturating L.
3. A bipartite graph is *k-regular* if every vertex has degree exactly k. Prove that every k-regular bipartite graph (for $k \geq 1$) has a prefect matching.

Exercise 2.7. For a graph G and a set $S \subseteq V(G)$, let $\text{odd}(G - S)$ denote the number of connected components of $G - S$ with an odd number of vertices. The *deficiency* of G is defined as

$$\text{defi}(G) \stackrel{\text{def}}{=} \max_{S \subseteq V(G)} (\text{odd}(G - S) - |S|).$$

1. Prove that $\text{defi}(G) \geq 0$ for any graph G.
2. Prove that every matching in G leaves at least $\text{defi}(G)$ vertices unmatched.
3. Prove that if G has a perfect matching, then $\text{odd}(G - S) \leq |S|$ for all $S \subseteq V(G)$.
4. Prove the converse: if $\text{odd}(G - S) \leq |S|$ for all $S \subseteq V(G)$ and $|V(G)|$ is even, then G has a perfect matching. (This is *Tutte's theorem*, and is the trickiest part.)
5. Prove that the maximum matching in G has size $\frac{1}{2}(n - \text{defi}(G))$. (This is called the *Tutte-Berge formula*.)

Exercise 2.8. Recall that the *chromatic number* of a graph G, denoted $\chi(G)$, is the number number of colors needed to properly color the vertices of G. In *list coloring*, each vertex v has a list $L(v)$ of permitted colors, and we seek a proper coloring where each vertex uses a color from its list. A graph G is *k-choosable* if it has such a coloring whenever all lists have size at least k. The *list chromatic number* $\chi_{\text{L}}(G)$ is the minimum k for which G is k-choosable.

1. Prove that $\chi_{\text{L}}(G) \geq \chi(G)$ for any graph G.
2. Prove that every graph G is $(\Delta + 1)$-choosable, where Δ is the maximum degree of G.
3. Prove that every tree is 2-choosable.
4. The *complete bipartite graph* $K_{n,n}$ has $2n$ vertices partitioned into two sides of n vertices each, with edges between every pair of vertices on opposite sides. Prove that $\chi(K_{n,n}) = 2$.
5. Prove that $\chi_{\text{L}}(K_{n,n}) \geq \log_2 n$. (This is tricky.)
6. Prove the $\chi_{\text{L}}(K_{n,n}) \leq O(\log n)$. (This is also tricky; *randomization* might be helpful.)

Exercise 2.9. An *edge coloring* of a graph is an assignment of colors to edges such that no two edges sharing a vertex have the same color. The *edge chromatic number* $\xi(G)$ is the minimum number of colors needed in an edge coloring of G.

1. Prove that $\xi(G) \geq \Delta$ for any graph G with maximum degree G.
2. Let G be a Δ-regular bipartite graph. (Every vertex has degree Δ). Prove that $\xi(G) = \Delta$.[7]
3. Now let G be a bipartite graph with maximum degree Δ. Prove that $\xi(G) = \Delta$. (This is called *König's theorem*.)

Exercise 2.10. Each year, medical students apply to residency programs, and programs decide which applicants to accept. Both sides have preferences: students rank programs, and programs rank applicants. The National Resident Matching Program (NRMP) has used a stable matching algorithm since the 1950s to assign students to programs in a way that prevents "blocking pairs"—a student and program who would both prefer each other over their assigned match.

Formally, consider two disjoint sets A and B with $|A| = |B| = n$. Each element of A has a strict preference ranking over all elements of B, and vice versa. A *perfect matching* is a bijection $M : A \to B$. A matching M is *unstable* if there exist $a \in A$ and $b \in B$ such that a prefers b to $M(a)$ and b prefers a to $M^{-1}(b)$; such a pair (a, b) is called a *blocking pair*. A matching with no blocking pair is *stable*.

The *Gale-Shapley algorithm* builds a perfect matching as follows. At a high level, each $a \in A$ applies to one $b \in B$ at a time, and each $b \in B$ tentatively accepts one of its applicants. It terminates when the tentatively accepted applications form a perfect matching.

In the first round, each $a \in A$ applies to its highest-ranked $b \in B$, and each $b \in B$ tentatively accepts its highest-ranked applicant, rejecting the rest. In each subsequent round, each rejected $a \in A$ applies to the best $b \in B$ it has not yet applied to. Each $b \in B$ tentatively accepts its most preferred among the new applicants and its current tentative match (if any), rejecting the rest.

1. Prove an upper bound on the number of rounds of the Gale-Shapley algorithm before terminating with a perfect matching.
2. Prove that the matching produced is stable.

[7] Hint: you might want to do exercise 2.6 first.

3. Prove that each element of A is matched to the best element of B that they could be matched to in any stable matching.

4. Prove that each element of B is matched to the *worst* element of A they could be matched to in any stable matching.

The last two points imply that, if there is a unique stable matching, the algorithm finds it regardless of which side proposes.

Exercise 2.11. Let $G = (V, E)$ be a directed graph. For distinct vertices $s, t \in V$, an (s, t)-*vertex cut* is a set $X \subseteq V \setminus \{s, t\}$ such that every path from s to t passes through some vertex in X. Two paths are *internally vertex-disjoint* if they share no vertices except possibly s and t. *Menger's theorem* states that the maximum number of internally vertex-disjoint (s, t)-paths equals the minimum size of an (s, t)-cut. Our goal is to prove Menger's theorem.

1. For disjoint sets $S, T \subseteq V$, an (S, T)-*path* is a directed path starting from S and ending in T. Two (S, T)-paths are *vertex-disjoint* if they share no vertices, including their endpoints. An (S, T)-*vertex cut* is a set of vertices $X \subseteq V$ intersecting every (S, T)-path. Prove that if there exist k vertex-disjoint (S, T) paths, then every (S, T)-vertex cut has size at least k.

2. Prove the converse: if every (S, T)-vertex cut has size at least k, then there exist k vertex-disjoint (S, T)-paths.

3. Now prove Menger's theorem (as stated above).

4. There is also an edge version: an s-t *edge cut* is a set of edges whose removal disconnects s from t, and two paths are *edge-disjoint* if they share no edges. Prove that the maximum number of edge-disjoint (s, t)-paths equals the minimum size of any (s, t)-edge cut.

Exercise 2.12. The *Ramsey number* $R(s, t)$ is the minimum n such that every 2-coloring of the edges of K_n (say, red and blue) contains either a red K_s or a blue K_t. Our goal is to prove *Ramsey's theorem*: $R(k, k) \leq \binom{2k-2}{k-1}$ for all k.

1. Prove that $R(s, 2) = s$ for all $s \geq 2$.

2. Prove that $R(3, 3) = 6$.

3. Prove that $R(s, t) \leq R(s - 1, t) + R(s, t - 1)$ for $s, t \geq 3$.

4. Conclude that $R(s,t) \leq \binom{s+t-2}{s-1}$.
5. Prove Ramsey's theorem.
6. Prove that $R(k,k) > 2^{k/2}$. (This is trickier; *randomization* might be helpful.)

Sample solution.

Problem. (Exercise 2.2 problems 1 and 2.) For the following recursive specifications, design a recursive algorithm implementing the specification. (No proof or analysis is needed.)

1. `subset-sum`$(x_1, \dots, x_n,\ T)$: Given n integers $x_1, \dots, x_n \in \mathbb{Z}$, and an additional integer T, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = T$, and `false` otherwise.

2. `min-vertex-cover`$(G = (V, E))$: Given an undirected graph $G = (V, E)$, return the minimum size of any vertex cover of G.

   (A vertex cover is a set of vertices $S \subseteq V$ that includes at least one endpoint of every edge $e \in E$.)

Solution.

`subset-sum`$(x_1, \dots, x_n,\ T)$

1. If $T = 0$ return `true`.
2. If $n = 0$ return `false`.

/* *x_1 is either in the sum or not.* */

3. Return `subset-sum`$(x_2, \dots, x_n,\ T) \vee$ `subset-sum`$(x_2, \dots, x_n,\ T - x_1)$.

`min-vertex-cover`$(G = (V, E))$

1. If $E = \emptyset$ then return 0.
2. Let $e = \{u, v\} \in E$ be any edge.

/* *Any vertex cover must contain either u or v.* */

3. Let $E_u = \{d \in E : u \in d\}$ and $E_v = \{d \in E : v \in d\}$ be the sets of edges incident to u and v, respectively.
4. Let $G_1 = (V - u, E \setminus E_u)$ be the graph obtained by removing u and all edges incident to u.
5. Let $G_2 = (V - v, E \setminus E_v)$ be the graph obtained by removing v and all edges incident to v.
6. Return $1 + \min\{$`min-vertex-cover`(G_1), `min-vertex-cover`$(G_2)\}$.

Rubric. Each problem is worth 2 points:
1 point. Correct base cases.
1 point. Correct induction/recursive step.

Chapter 3

Selection and closest pair in the plane

This chapter is also about recursive algorithms that rely on induction for correctness. For the problems discussed here, it is relatively easy to see that they have correct polynomial time algorithms. The focus instead is on *faster* polynomial running times. We will use recursion to more aggressively *divide* the input and make the subproblems significantly smaller.

This paradigm is called *divide-and-conquer*. We have seen it already in `merge-sort`. In `merge-sort`, we divided the input in half, and recursively sorted each half separately. The two sorted halves could then be combined very efficiently by merging. `merge-sort` was also used as a vehicle to introduce the recursion tree method for analyzing running times. We will see several more algorithms that leverage insights from recursion trees — introducing clever ideas that do just enough to establish a better recurrence, hence a better running time. In general, the "divide" step embodies a running theme of "identifying subproblems". The second "conquer" component, so to speak, highlights an additional angle of "combining subproblem solutions".

3.1 Selection

Given a set of n numbers $A[1..n]$, which is the median? The obvious route is to sort all the numbers, and then pick out the $\lceil n/2 \rceil$th number from the sorted order. This takes $O(n \log n)$ time. Can one do better?

The more general problem is called *selection*: given a set of n numbers $A[1..n]$ in arbitrary order, and an index $k \in [n]$, the goal is to find the kth smallest number in A. Again, we can solve this by sorting A, but our goal is to do this faster.

We will design a recursive algorithm for selection. The first step is define the intent of our recursive algorithm, which establishes the induction hypothesis for the sake of correctness. We do this before developing the algorithm.

> `select(`k`,`$A[1..n]$`)`: Given an array $A[1..n]$ of comparable elements and an integer $k \in \{1, \ldots, n\}$, returns the kth smallest element out of $A[1..n]$.

We now focus on implementing `select(`$k, A[1..n]$`)` exactly as specified. The high-level idea is to aggressively cut down the search space — very much like binary search.

Selection via medians. Our approach is based on the idea of a *pivot*. To motivate this idea, suppose we have a subroutine `median(`$A[1..n]$`)` that can find (specifically) the median of A in linear time. To formalize this, we define

$$\begin{aligned}\texttt{median}(A[1..n]) &\stackrel{\text{def}}{=} \texttt{select}(\lceil n/2 \rceil, A[1..n]) \\ &= \text{the } \lceil n/2 \rceil \text{th smallest element out of } A[1..n].\end{aligned}$$

We assume access to a $O(n)$-time subroutine for `median(`$A[1..n]$`)`. `median(`$A[1..n]$`)` is a special case of `select(`$k, A[1..n]$`)`, and it is interesting to ask if the special can be leveraged to accelerate the general case.

Consider an instance of `select(`$k, A[1..n]$`)`. Let p be the median of $A[1..n]$, produced by `median(`$A[1..n]$`)`. We call p the "pivot". Of course if $k = \lceil n/2 \rceil$ then we return p. Otherwise, either $k < \lceil n/2 \rceil$ or $k > \lceil n/2 \rceil$. If $k < \lceil n/2 \rceil$, then we know the rank k element is among the $\lceil n/2 \rceil - 1$ elements smaller than p. We assemble these elements in an array $B[1..\lceil n/2 \rceil - 1]$, and call `select(`$k, B$`)`. Similarly we recurse on the elements bigger than p if $k > \lceil n/2 \rceil$, though now we are looking for the $(k - \lceil n/2 \rceil)$-rank element remaining. By induction we assume that the recursive calls to `select` on smaller input correctly returns the desired element. See Fig. 3.2 on the next page for pseudocode.

Consider the running time of this implementation of `select(`$k, A[1..n]$`)`. Let $T(n)$ denote the maximum running time of `select(`$k, A[1..n]$`)` on any input of size at most n. `select(`$k, A[1..n]$`)` spends $O(n)$ time to compute the median, and then makes at most one recursive subcall. Because p is always the median, the subcall is a selection over at most $\lfloor n/2 \rfloor$ elements. So we model T recursively as follows:

$$\begin{aligned}T(1) &= 1 \\ T(n) &= T(n/2) + O(n).\end{aligned}$$

If we draw out the recursion tree for T, it hardly looks like a tree. Each problem begets one subproblem of half the size. The total work per level is the number of elements in the lone subproblem at that level. The work per level, over all levels, gives a geometrically decreasing sum of the form $n + n/2 + n/4 + \cdots$. As we know (cf. Section C.1.3), the geometrically decreasing coefficients sum to a constant, and we have

$$T(n) = O(n).$$

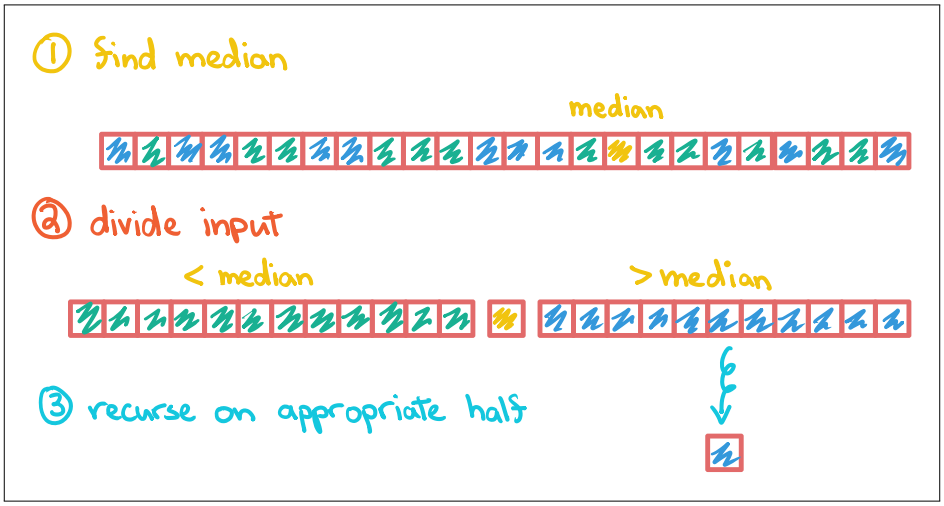


Figure 3.1: Sketch of a divide-and-conquer selection algorithm using the median as the pivot.

`select`$(k, A[1..n])$

/* `select`$(k, A[1..n])$ = *the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties arbitrarily.)* */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median`(A). // *$O(n)$ time by assumption.*
3. If $k = \lceil n/2 \rceil$ then return p.
4. If $k < \lceil n/2 \rceil$ then return `select`(k, B), where $B[1..\lceil n/2 \rceil - 1]$ is an array of the $\lceil n/2 \rceil - 1$ elements smaller than p.
5. If $k > \lceil n/2 \rceil$ then return `select`$(k - \lceil n/2 \rceil, B)$, where $B[1..n - \lceil n/2 \rceil]$ is an array of the $n - \lceil n/2 \rceil$ elements larger than p.

Figure 3.2: A (conceptual) divide-and-conquer algorithm for `select`$(k, A[1..n])$ using `median`$(A[1..n])$ to select the median as a pivot.

Thus, assuming a linear time algorithm for selecting the median, we can select any item in linear time.

Selection via approximate medians. Of course, we do not know how to implement `median(`$A[1..n]$`)` in $O(n)$ time. Suppose we weakened our assumption to assume we have access to an *approximate* median instead. Formally we assume:

> `apx-median(`$A[1..n]$`)` returns an element $p \in A[1..n]$ with rank between $n/10$ and $9n/10$, in $O(n)$ time.

That is, p is always in the middle (4/5)ths of the input. (The choice of 1/10 was arbitrary; the reader can check that any constant factor leads to the same conclusion.)

We can use `apx-median(`$A[1..n]$`)` to generate a pivot p in the exact same way as we used `median(`$A[1..n]$`)`. One difference is that after receiving p, we have to calculate its rank ℓ among A, which takes $O(n)$ time. Thereafter we split the input by p, and recurse on the appropriate subproblem. See Fig. 3.4 on the following page for pseudocode.

For the running time, we now have

$$T(n) \leq T(9n/10) + O(n).$$

This recurrence might appear worse than the previous one, since we are only removing 10% of the input rather than half. But when we sketch out a recursion tree, we again find a geometric series except now the coefficient is (9/10) rather than 1/2. A larger coefficient (as long as it remains below 1) increases the running time by only a constant factor:

$$n + \frac{9}{10}n + \left(\frac{9}{10}\right)^2 n + \left(\frac{9}{10}\right)^3 n + \cdots = O(n).$$

So this second version of `select(`k`,`$A[1..n]$`)`, using a much weaker subroutine to select the pivot, is still a $O(n)$-time algorithm.

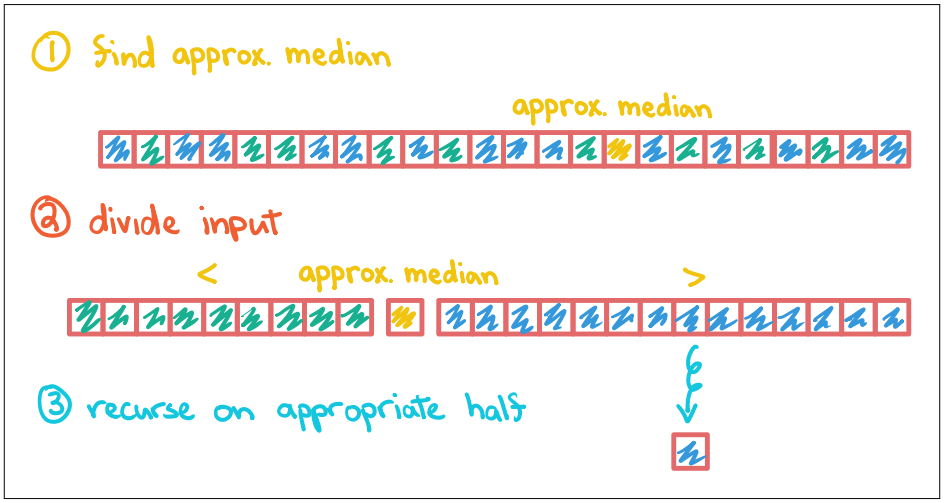


Figure 3.3: Sketch of a divide-and-conquer algorithm using an approximate median as a pivot.

`select`$(k, A[1..n])$

/* `select`$(k, A[1..n])$ = *the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties consistently.)* */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `apx-median`(A). // *$O(n)$ time by assumption.*
3. Compare p to each element in $A[1..n]$ to identify its rank ℓ. // *$O(n)$.*
4. If $k = \ell$ then return p.
5. If $k < \ell$ then return `select`$(k, A[1..\ell - 1])$, where $B[1..\ell - 1]$ is an array of the $\ell - 1$ elements smaller than p.
6. If $k > \ell$ then return `select`$(k - \ell, B)$, where $B[1..n - \ell]$ is an array of the $n - \ell$ elements larger than p.

Figure 3.4: A (second, conceptual) divide-and-conquer algorithm for `select`$(k, A[1..n])$ using `apx-median`$(A[1..n])$ to generate a pivot that is approximately the median.

Median-of-medians. We have now reduced linear time selection to approximating the median in linear time. We present a method for the latter, called the *median-of-medians*, invented by Blum, Floyd, Pratt, Rivest, and Tarjan [BFP+72].[1]

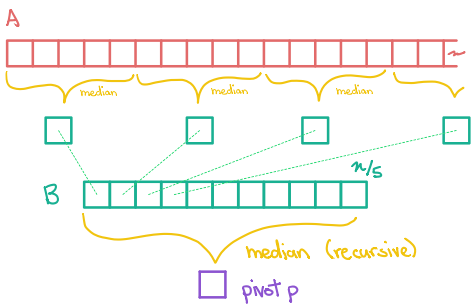
At a high level, [BFP+72] carefully selects a subset $B[1..m]$ of $A[1..n]$ of size $m = n/5$, and recursively computes the median of B. The process to select B (which we will describe momentarily) takes $O(n)$ time, and is carefully designed to ensure that the median of B is a $(7/10)$-approximate median overall. This gives a recurrence of the form

$$T(n) = T(7n/10) + T(n/5) + O(n). \tag{3.1}$$

Applying the recursion tree method to this recurrence gives $T(n) = O(n)$. (Why?) That is, linear time!

Let us now explain how [BFP+72] selects the approximate median. It takes A and partitions it into subintervals of 5 elements each:

$$\{A[1], \dots, A[5]\}, \{A[6], \dots, A[10]\},$$
$$\{A[11], \dots, A[15]\}, \{A[16], \dots, A[20]\}, \dots$$



(The last interval may have fewer than 5.) From each set of 5, we compute the median of those 5 elements in constant time. This takes $O(n)$ time overall and assembles a set $B[1..m]$ of medians, where $m = \lceil n/5 \rceil$. Observing that B is smaller than A by a factor of 5, we recurse on B, computing the "median-of-medians" p. We choose p to be the pivot.

While p is not an exact median, one can show that p has rank roughly inside the range $3n/10$ and $7n/10$ - which is enough to justify the recurrence (3.1) above. Pseudocode is given in Fig. 3.5 on the next page. Figure 3.6 on the following page implements `select(`k`,`$A[1..n]$`)` with `median-of-medians(`$A[1..n]$`)`.

We now formally analyze `median-of-medians(`$A[1..n]$`)`.

Lemma 3.1. `median-of-medians(`$A[1..n]$`)` *returns an element p with rank ℓ in the range*

$$3n/10 - 2 \leq \ell \leq 7n/10 + 2.$$

[1] An old joke:

> *Question:* How many Turing Awards does it take to find the median in $O(n)$?
> *Answer:* 4, plus an unofficial founder of Sun Microsystems!

`median-of-medians(`$A[1..n]$`)`

/ Assemble medians of subsets of 5 elements each. */*

1. Divide $A[1..n]$ into $m = \lceil n/5 \rceil$ subintervals of 5 numbers each, and compute the median of each, producing a list $B[1..m]$ for $m = \lceil n/5 \rceil$. (The last interval may have less than 5.)

/ Recursively compute the median of these medians. */*

2. Return `select(`$\lceil m/2 \rceil$`,`B`)`

Figure 3.5: The median-of-medians subroutine for approximating the median.

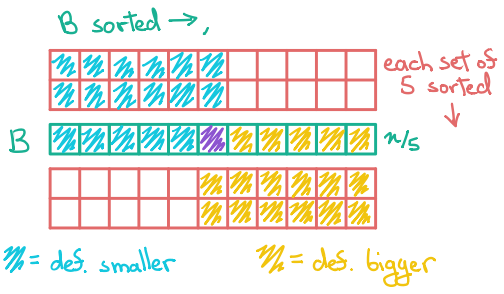
`select(`k`,`$A[1..n]$`)`

/ `select(`k`,`$A[1..n]$`)` = the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties consistently.) */*

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median-of-medians(`A`)`.
3. Compare p to each element in $A[1..n]$ to identify its rank ℓ.
4. If $k = \ell$ then return p.
5. If $k < \ell$ then return `select(`k`,`B`)`, where $B[1..\ell - 1]$ is an array of the $\ell - 1$ elements smaller than p.
6. If $k > \ell$ then return `select(`$k - \ell$`,`B`)`, where $B[1..n - \ell]$ is an array of the $n - \ell$ elements larger than p.

Figure 3.6: The linear time divide-and-conquer algorithm for `select(`k`,`$A[1..n]$`)` by [BFP+72] using the median-of-medians as a pivot.

Proof. Let us first assume that n is divisible by 5 for simplicity. Consider the array B of "intermediate medians" of size $m = \lceil n/5 \rceil$, of which p is the median. p is less than or equal to at least $\lceil m/2 \rceil \geq \lceil n/10 \rceil$ of these intermediate medians. Moreover, each intermediate median, as the median of its own set of 5 elements, is less than 2 more elements from its own set up 5. This $\ell \geq 3\lceil n/10 \rceil \geq 3n/10$.



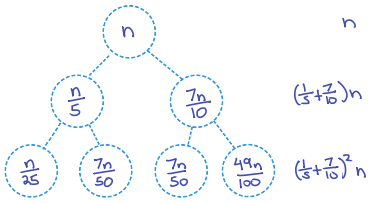
In general when n is not divisible by 5, there may be one set with less than 5 elements. For that deficient subset, we can only guarantee that the intermediate

median is less than or equal to at most one element (namely, itself) from its set. So instead we get $\ell \leq 7n/10 + 2$.

Similarly one can show that p is greater than or equal to at least $3n/10 - 2$ elements. □

Running time. Above we asked the reader to try to analyze the recurrence

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$



themselves. We now provide the details for the sake of completeness. A quick recursion tree reveals that the total amount of work at the kth level is

$$(1/5 + 7/10)^k n = (9/10)^k n.$$

Thus the total running time is bounded above by

$$O\left(\sum_{k=0}^{\infty} (9/10)^k n\right) = O(n).$$

In conclusion:

Theorem 3.2. *[BFP+72] Given an array of n comparable elements $A[1..n]$ and an index $k \in [n]$, the kth smallest number of A can be computed in $O(n)$ time.*

Let us briefly mention a simpler, *randomized* variant that also runs in linear time *on average*. We haven't yet developed the tools to analyze it but the algorithm is simple enough (and perhaps obvious). Rather than by "median-of-medians", simply pick a number at random as a pivot. Divide A into the set of numbers smaller and bigger than the pivot. Recurse appropriately.

The intuition to the randomized analysis is also very simple. With probability $1/2$, the pivot is one of the middle $(n/2)$-numbers, and we eliminate at least $n/4$ numbers from the search. So imagine flipping a bunch of coins. Each coin toss represents one iteration, which takes time linear in the number of elements remaining. Each heads cuts down the input by half. How many coin tosses until the input halves? In expectation, at most a constant. This leads to a $O(n)$ running time and we will formalize the argument later.

The randomized algorithm works better in practice.

3.2 Minimum distance in $\mathbb{R}^2$



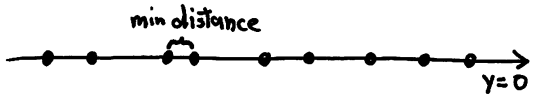
We consider the problem of computing the minimum pairwise distance among a set of points in the plane. The input consists of n points P in $\mathbb{R}^2$. The goal is to find the minimum Euclidean distance between any two points $p, q \in P$. Here we recall that for two points $p = (p_1, p_2)$ and $q = (q_1, q_2)$, the Euclidean distance $\|p - q\|$ is given by

$$\|p - q\| = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}.$$

We assume for simplicity that all the coordinates of all the points are distinct.[2] Of course there is an $O(n^2)$-time algorithm where we simply calculate the distance of every pair. The question is whether one could do better. In anticipation of a recursive algorithm, we first declare a recursive spec.

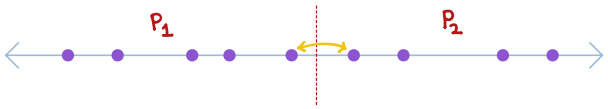
> `min-distance(`P`)` = the minimum Euclidean distance between any pair of points in P (or $+\infty$ if there are less than 2 points in P).

The goal is to implement `min-distance(`P`)` exactly as described above.



To build some intuition, it is helpful to consider the 1-dimensional case; or equivalently, assume that all points have the same y-coordinate, (say) $y = 0$. (We continue to assume the x-coordinates are distinct.) The most obvious approach is to sort the points by x-coordinate, and then scan and check all the consecutive pairs in sorted order. This takes $O(n \log n)$ time. This is very good but it does not seem to generalize to general y-coordinates. In $\mathbb{R}^2$, the minimum distance is not necessarily attained by a consecutive pair of points when listed in increasing order of x.

[2]The fancy way to state this assumption is to say that the points are in "general position". In practice, we can often assume general position by first perturbing the points by a sufficiently small amount. We also point out that the algorithms here are simple enough to adapt to duplicate coordinates. We assume general position to keep the presentation clean and focused on the high-level ideas.



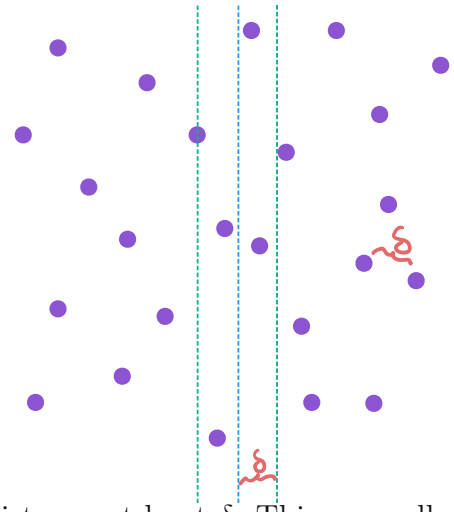
A second approach to the one-dimensional ($y = 0$) setting is to first find the median x-coordinate a. (We can compute a in $O(n)$ time via `select` from Section 3.1). Let P_1 be the subset of points with x-coordinate $< a$ and P_2 be the subset of points with x-coordinate $\geq a$. We recursively compute the minimum distance in P_1 and P_2. Let δ be the minimum of the two values returned. δ is the minimum distance among pairs of points within P_1 and within P_2; it remains to check the minimum distance of pairs *across* the two halves. To this end, we calculate the distance between the right-most-point on the left half P_1, and the left-most-point on the right half P_2. We return the minimum of this distance and δ.

The second approach almost generalizes to $\mathbb{R}^2$, except for the last step where we get the minimum distance across the two halves. The minimum distance across P_1 and P_2 is not necessarily between the two points closest to the dividing line $x = a$.

We do know that that any point in the minimum cross pair has x-coordinate within δ of a. (Otherwise the x-coordinate alone makes the distance of the cross-pair $> \delta$). Let

$$P_3 = \{(x, y) \in P : |x - a| \leq \delta\}$$

gather up all the points in P whose x-coordinates are within δ of a. Now consider the following geometric facts.



1. P_3 lies in a narrow, vertical band of width 2δ.
2. Any two points in $P_1 \cap P_3$ have distance at least δ, and any two points in $P_2 \cap P_3$ have distance at least δ. This generally forces P_3 to be spread apart.

Combining the two facts leads to the following observation (formalized in Lemma 3.3 below):

> For any point $p \in P_3$, there are at most a constant number of other points with y-coordinate within δ of p.

We take advantage of this by sorting P_3 by y-coordinate, and only compute the distance between pairs of points that are within 10 ranks of one another in the sorted list. (By Lemma 3.3, 10 is a sufficiently large constant.)

`min-distance(`P`)`

/ We assume P is sorted by y-coordinate. (Otherwise sort P once in a preprocessing step.) */*

1. If $n \leq 1$ then return $+\infty$.
2. If $n = 2$ then return the distance between the two points in P.
3. Let a be the median of the x-coordinates. // $O(n)$

/ Divide P along the vertical line $x = a$ */*

4. Let $P_1 = \{(x, y) \in P : x < a\}$ and $P_2 = \{(x, y) \in P : x \geq a\}$ // $O(n)$

/ Find the minimum distances within each half P_1 and P_2 recursively. */*

5. Let $\delta_1 =$ `min-distance(`P_1`)`, $\delta_2 =$ `min-distance(`P_2`)`, and $\delta = \min\{\delta_1, \delta_2\}$. // $2T(n/2)$

/ It remains to find the minimum distance across P_1 and P_2. */*

6. Let $P_3 = \{(x, y) \in P : |x - a| < \delta\}$ and let $p_1, \ldots, p_k$ list P_3 in decreasing order of y-coordinate. // $O(n)$
7. Let δ_3 be the minimum of $\|p_i - p_j\|$ over all i, j with $i < j < i + 10$. // $O(n)$
8. Return $\min\{\delta, \delta_3\}$.

Figure 3.7: Computing the minimum distance in $\mathbb{R}^2$ by divide-and-conquer along the horizontal axis.

Pseudocode of the divide-and-conquer algorithm described above is given in Fig. 3.7. As a relatively minor point, instead of sorting by y-coordinate in each recursive call, we can sort all of P once in a preprocessing step. We assume that P is already sorted by y-coordinate in the implementation in Fig. 3.7.

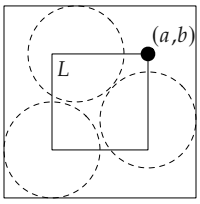
Lemma 3.3. *Let $b \in \mathbb{R}$. There are at most 10 points in P with coordinate in the rectangle $[a - \delta, a + \delta] \times [b - \delta, b]$,*

Proof. The following is an example of a "packing argument", which is common in computational geometry.

We split the rectangle into two squares

$$L = [a - \delta, a) \times [b - \delta, b] \text{ and } R = [a, a + \delta] \times [b - \delta, b].$$

For each half we show that there are at most 5 points.



Consider the left half L. Any pair of points in L, being both in P_1, has distance at least δ. For each point $p \in P \cap L$, draw a ball B_p with radius $\delta/2$ centered at p. Each ball B_p has area $\pi\delta^2/4$, and lies in the "padded" rectangle $[a - 1.5\delta, a + .5\delta] \times [b - 1.5\delta, b + .5\delta]$ of area $2\delta \times 2\delta = 4\delta^2$. These balls are also interior disjoint because the minimum distance between their centers is δ. The number of these balls that can fit in the padded rectangle, by comparing areas, is at most

$$\frac{4\delta^2}{\pi\delta^2/4} = \frac{16}{\pi} \approx 5.09.$$

Since the maximum number is also integer, we conclude there can be at most 5 balls, hence 5 points in $P \cap L$.

By the same argument (now applied to P_2, instead of P_1) there are at most 5 points in $R \cap P$. □

The overall proof of correctness is as follows. We will prove that Fig. 3.7 correctly implements `min-distance(`P`)` as specified above by induction on $|P|$. The base cases are when $|P| \leq 2$; if $|P| < 1$ then the distance is automatically $+\infty$; if $|P| = 2$, there is only one pairwise distance to consider and that is the value returned. Consider the general case where $|P| > 2$. We assume by induction that the implementation is correct for all strictly smaller point sets.

After dividing the input size into P_1 and P_2, we know that the minimum distance is either within P_1, within P_2, or across P_1 and P_2. By induction we assume that `min-distance(`P_1`)` and `min-distance(`P_2`)` return the (true) minimum distance within P_1 and P_2, respectively. For pairs going across P_1 and P_2, it suffices to compute the minimum distance in P_3 since otherwise the difference in x-coordinate is at least δ. Within P_3, by Lemma 3.3, it suffices to compare each point $p \in P_3$ with the 10 points with closest y-value, which is exactly what the algorithm does. So the algorithm also finds the minimum distance within P_3 correctly. The minimum distance from the three sets P_1, P_2, and P_3 gives the minimum distance of all of P, as claimed.

As for the running time, we have two recursive calls to point sets of size $n/2$, plus

$O(n)$ additional work (as annotated in the code). This gives a recursion of the form

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 2, \\ 2T(n/2) + O(n) & \text{otherwise.} \end{cases}$$

This is the same recurrence as `merge-sort` and is solved by $T(n) = O(n \log n)$.

3.3 Additional notes and references

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 Lecture 11. The second half of this lecture is about selection. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 Lecture 12. The first half of this lecture is about the closest pair problem. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

3.4 Exercises

Many exercises can be found in [KT06, Chapter 5], [DPV08, Chapter 2], and [Eri19, Chapter 1]. When designing and analyzing divide-and-conquer algorithms, make sure to address the following (unless specified otherwise):

1. A recursive spec for your algorithm.
2. A recursive implementation of your recursive spec.
3. An brief explanation of how to use the recursive algorithm to solve the original problem. (e.g., what is the initial call to the algorithm, or any post-processing and pre-processing details.)
4. An analysis of the running time.
5. A proof by induction, where the induction hypothesis is based on the recursive spec.

Exercise 3.1. Let $A[1..n]$ be an array of elements. The *frequency* of an element is the number of times it appears in $A[1..n]$. An element is a *majority element* if it has frequency at least $n/2$.

If the elements are comparable then we can identify the majority element by sorting all the elements and scanning the sorted list. Here we do *not* assume that the elements are pairwise comparable; we can only test if two elements $A[i]$ and $A[j]$ are equal.

1. Write a recursive spec for the majority element problem.
2. Design a divide-and-conquer algorithm (implementing your recursive spec) for the majority element problem that divides the input in half, and runs in $O(n \log n)$.[3]
3. Prove your algorithm is correct by induction, where the recursive spec is your induction hypothesis.
4. Analyze the running time of your algorithm.

[3]There is also a well-known $O(n)$ time algorithm which we are not asking for.

Exercise 3.2. Here we assume the same model as exercise J.4 – an array $A[1..n]$ of incomparable elements[4] – and generalize the notion of a majority element. For $k > 1$, an element is a $(1/k)$-*heavy-hitter* if it has frequency at least n/k. (e.g., majority elements are $(1/2)$-heavy hitters.) Note that there are at most k $(1/k)$-heavy-hitters in $A[1..n]$.

1. (2.5 pt.) Give a recursive spec for the $(1/k)$-heavy hitter problem.
2. (2.5 pt.) Give a divide-and-conquer algorithm for the $(1/k)$-heavy hitter problem implementing your recursive spec (the faster the better). (This algorithm should be similar to your majority element algorithm.)
3. (2.5 pt.) Analyze the running time of your algorithm.
4. (2.5 pt.) Prove your algorithm is correct by induction, where the recursive spec should act as your induction hypothesis.

Your solutions should generalize that of finding the majority elements. That is, plugging in $k = 2$ should recover the same results as for the majority problem.[5]

Exercise 3.3. Let P be a set of n points in $\mathbb{R}^2$.[6] Suppose you want to compute a list of all pairs of points with distance with a factor of 2 of the minimum distance between all pairs of points. Design and analyze an algorithm for this problem.

(Food for thought: what does your running time say about the total number of pairs of points with distance within a constant factor of the minimum distance between all pairs of points?)

Exercise 3.4. Recall that given a set P of n points in $\mathbb{R}^2$, we can compute the minimum distance between any pair of points in P in $O(n \log n)$ time. Consider now the problem of computing the second smallest distance between any pair of points in P. Design and analyze an algorithm for this problem.

Exercise 3.5. Recall that ants leave *trail pheromones* to help other ants follow its trail, particularly if the first ant identifies a good source of food.

[4] In addition to disallowing sorting, no hash tables are allowed as well.

[5] Your algorithm should be faster than $O(n^2)$ (which is the trivial running time) for, e.g., $k = n^{1/3}$.

[6] For simplicity you may assume the points are in general position; that is, no two points have the same x-coordinate or the same y-coordinate.

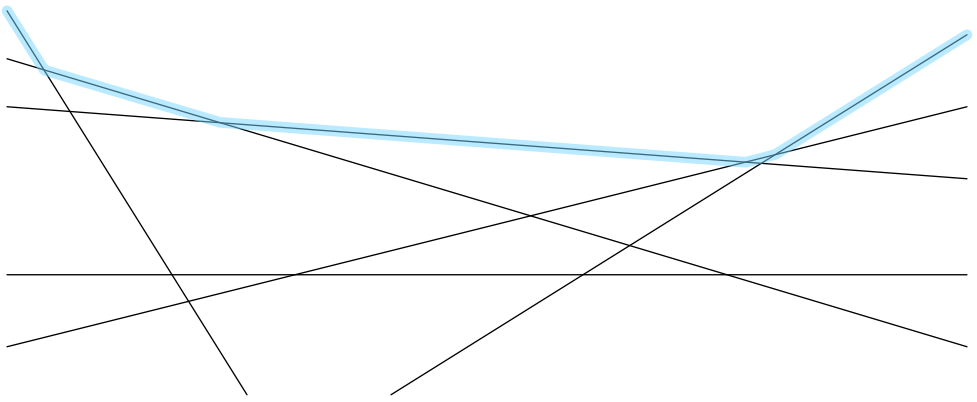
Consider an $n \times n$ grid of cells identified by coordinates $\{(i, j) : 1 \leq i, j \leq n\}$. We say two cells are adjacent if they share a side. We are told an ant was placed in cell $(1, 1)$, and then walked around the grid, always from one cell to an adjacent cell, until stopping at some cell and then being taken off the grid. (The ant cannot move diagonally, hop over cells, and so forth. It may revisit the same cell.) The goal is to identify the final cell where the ant finished walking.

For every ordered pair of adjacent cells (c, d) (where $c, d \in [n] \times [n]$), the pheromones allow us to deduce the number of times the ant moved from cell c to cell d. We want to use this information to identify the final cell of the ant. We assume an *oracle* model where in a single query to the oracle, the user specifies an ordered pair of adjacent cells (c, d), and the oracle returns the number of times the ant moved from cell c to cell d. Design and analyze an algorithm to identify the final cell where the ant was lifted out of the grid. As always, the faster the better. We treat each call to the oracle as a constant time operation.

Exercise 3.6. Let L be a set of n lines of the plane, where the ith line is of the form $y_i(x) = a_i x + b_i$ for $a_i, b_i \in \mathbb{R}$. The *upper envelope* is the function

$$U_L(x) = \max_{i \in [n]} y_i(x).$$

Observe that U_L is piecewise-linear and convex. We represent the upper envelope as a list of *breakpoints* $(x_1, \ell_1), (x_2, \ell_2), \ldots, (x_m, \ell_m)$ where $x_1 < x_2 < \cdots < x_m$ are the x-coordinates where the topmost line changes, and ℓ_i is the line achieving the maximum for $x \in [x_i, x_{i+1})$.[7]



[7] You should also specify the line on top for $x < x_1$; handle this however you like.

For example, given three lines $\ell_1 \colon y = -x + 4$, $\ell_2 \colon y = 2$, and $\ell_3 \colon y = x$, the upper envelope is: ℓ_1 for $x < 1$, then ℓ_2 for $x \in [1, 2]$, then ℓ_3 for $x > 2$.

Design and analyze a recursive algorithm, as fast as possible, computing the upper envelope of n lines in the plane.[8]

Remark 3.4. The upper envelope appears in computational geometry (convex hulls, linear programming) and in the "convex hull trick" for optimizing certain dynamic programs.

Exercise 3.7. Given a set of numbers $x_1, \ldots, x_n$, the *prefix sums* are the n numbers,

$$x_1,\ x_1 + x_2,\ x_1 + x_2 + x_3,\ \ldots,\ x_1 + \cdots + x_n.$$

Of course the prefix sums can be computed in $O(n)$ time, one prefix sum at a time. What about in *parallel*; e.g., on a GPU?

Here we consider a *parallel* computation model, called *PRAM*. To motivate this model, imagine one had arbitrarily many processors, with shared memory, trying to solve a common problem. Even with tons of computing, there may still be sequential bottlenecks — some steps have to be executed before others — and issues of synchronization.

In the PRAM model, the processors have access to shared memory. To simplify the synchronization, we restrict our model to work in *rounds*. In each round, each processor reads a constant amount of information from memory, does a constant amount of work, and writes to a constant number of locations in memory. We will restrict ourselves to algorithms where in each round, different processors always write to different locations, so we can avoid issues of contention resolution.

The number of rounds needed to complete the task is the *parallel running time*. The total number of operations, over all processes, is called the *total amount of work*. The ideal parallel algorithm has total work similar to the best known sequential algorithm, but has a much faster parallel running time.

For example, here is a first attempt at a parallel divide-and-conquer algorithm for prefix sums. We first define our recursive spec:

> `prefix-sum(`$A[1..n]$`)`: Given an array of numbers $A[1..n]$, returns an array of the prefix sums of A.

Our first implementation divides A into equal-size subarrays and recursively computes the prefix sums on each. Observe that the prefix sums of the second half are missing

`prefix-sum(`$A[1..n]$`)`

/ Given an input array of numbers $A[1..n]$, returns an array of the prefix sums of A. */*

1. If $n \leq 1$ then return A.
2. Let $k = \lceil n/2 \rceil$.
3. $B[1..k] \leftarrow$ `prefix-sum(`$A[1..k]$`)` and $C[1..n-k] \leftarrow$ `prefix-sum(`$A[k+1..n]$`)` (in parallel).
4. $D[i] \leftarrow B[k] + C[i]$ for $i = 1, \ldots, n-k$. *// $O(n)$ work in $O(1)$ rounds*
5. Return the concatenation of $B[1..k]$ and $D[1..n-k]$. *// $O(n)$ work in $O(1)$ rounds*

Figure 3.8: A parallel prefix sum algorithm with $O(\log n)$ time and $O(n \log n)$ work.

the sum from the first half. We fix this by taking the last prefix sum of the first half and adding it to each prefix sum in the second half. See Fig. 3.8 for pseudocode.

Let $T(n)$ denote the parallel running time on an input of size n. We have

$$\begin{aligned} T(0) &= T(1) = 1 \\ T(n) &= T(n/2) + O(1) \qquad\qquad \text{for } n > 1. \end{aligned}$$

Here we observed that although parallel prefix sum has two recursive calls, they are made in parallel. This is the same recurrence as (sequential) binary search, and implies a $O(\log n)$ parallel running time.

Now we compute the total amount of work. This is the same as the running time if the `parallel-prefix-sum` algorithm was simulated sequentially. Letting $W(n)$ be the total amount of work on an input of size n, we have

$$\begin{aligned} W(0) &= W(1) = 1 \\ W(n) &= 2W(n/2) + n/2 \qquad\qquad \text{for } n > 1. \end{aligned}$$

This is the same recurrence as merge sort and other algorithms we've seen, and gives $O(n \log n)$ total work.

[8] *Hint:* Of course n lines induces $\binom{n}{2}$ points of intersection, but how many breakpoints can the upper envelope have?

This parallel algorithm is very fast, but has a $O(\log n)$-factor more work than the obvious sequential algorithm. The goal of this exercise is to develop a (recursive) parallel divide-and-conquer algorithm that is just as fast — $O(\log n)$ parallel time — but has $O(n)$ total work. In addition to designing the algorithm, you should analyze the running time and the work similar to how we did above. (You are welcome to use the same recursive spec, but don't forget to write it out.)

It may be fun to try to figure this out without any hints. (Maybe you'll come up with a new approach!) But if you want a little help, then there are some guiding questions in the following footnote.[9]

Remark 3.5. To generalize beyond summing numbers, observe that the only property of addition we used was the associative property:

$$a + (b + c) = (a + b) + c.$$

So with essentially the same algorithm, one can compute generalized prefix sums for any associative binary operator in $O(\log n)$ rounds and $O(n)$ work.

Exercise 3.8. Consider the following generic recursion.

$$T(n) = \alpha T(\beta n) + n^\gamma$$
$$T(1) = \delta$$

where $\alpha, \beta, \gamma, \delta > 0$ are fixed constants. Prove each of the following using the recursion tree method.

1. For any $\beta < 1$, $T(n)$ is at most a polynomial in n. (Even if, say, $\alpha = 2^{260}$. Here $\alpha, \beta, \gamma, \delta$) may appear in the exponent of the polynomial.) For simplicity, you may assume that α is an integer.

2. Show, using the recursion tree method, that if $\alpha\beta^\gamma < 1$, then $T(n) \leq O(n^\gamma)$.

3. Show, using the recursion tree method, that if $\alpha\beta^\gamma = 1$, and $\beta < 1$, $T(n) \leq O(n^\gamma \log(n))$.

[9] Hint: Suppose you had the prefix sums $y_i = x_1 + x_2 + \cdots + x_i$ for all *even* i. Can you get the prefix sums for all *odd* i in $O(1)$ rounds and $O(n)$ work?

And then: how do you get the prefix sums for even i efficiently? Of course you could compute all the prefix sums, and then pick out the even indices i. But that's not going to be help us speed up computing all the prefix sums. Can you get the prefix sums for even i by creating another instance of the prefix sums problem with significantly fewer numbers?

Chapter 4

Fast multiplication and Fourier transform

We continue our theme from Chapter 3, developing fast recursive algorithms based on dividing up the input with each recursive calls. The applications from Chapter 3, selection and closest pair in $\mathbb{R}^2$, divided their input by some kind of median or approximate median. The applications here have a more numerical flavor, and will present different ways to divide the input.

4.1 Multiplication

Our next example is integer multiplication. Let $x, y \in \{0,1\}^n$ be two n-bit integers; we want to compute the product $x \times y$. The standard approach to multiplication I learned in grade school is an $O(n^2)$-time algorithm, that multiplies every bit in x with every bit in y (along with some addition and carrying). In this section we will use divide-and-conquer to obtain a faster algorithm running in roughly $O(n^{1.585})$.

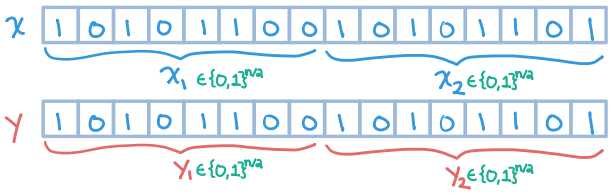
Improving the polynomial dependency on the number of bits may not seem very practical, since most programs work with fixed bit widths, and modern CPU's have specialized hardware dedicated to multiplying fixed width integers very quickly. Our algorithm will mostly be useful for extremely large integers. (Maybe our tricks could also help the CPU designers). Still we present integer multiplication for the following reasons (besides the usual punt that "this is theory"). First, both the problem and the algorithm are (in our opinion) simple. We understand all of the mathematics concerning integer multiplication and the only novelties here are algorithmic.

Second, the algorithm we present has some historical significance. Surely we know all there is to know about multiplying integers, and there's no deficiencies in the $O(n^2)$ approach. So in the 1950's the famous mathematician Andrey Kolmogorov conjectured that it was the best possible algorithm; that is, he conjectured a $\Omega(n^2)$ lower bound for this problem. But in 1960 Anatoly Karatsuba found a substantially faster algorithm and this is the algorithm that we discuss. It was a surprising revelation that more broadly suggests that the "obvious" approaches to many other

well-understood problems can be improved.

We first explain Karatsuba's algorithm at a high level. Let $x, y \in \{0,1\}^n$ be two n-bit numbers we want to multiply. We assume n is a power of 2 for simplicity (and otherwise pad x and y with leading 0's).

Let us break the bits in two halves $x = (x_1, x_2)$ and $y = (y_1, y_2)$, where $x_1 \in \{0,1\}^{n/2}$ (resp. $y_1 \in \{0,1\}^{n/2}$) are the $n/2$ most significant bits of x (resp. of y), and $x_2 \in \{0,1\}^{n/2}$ (resp. $y_2 \in \{0,1\}^{n/2}$) represent the remaining, $n/2$ least significant bits of x, (resp. y). That is,



$$x = x_1 2^{n/2} + x_2 \text{ and } y = y_1 2^{n/2} + y_2.$$

Nothing special. We have

$$xy = \left(x_1 2^{n/2} + x_2\right)\left(y_1 2^{n/2} + y_2\right) = x_1 y_1 2^n + (x_1 y_2 + x_2 y_1) 2^{n/2} + x_2 y_2.$$

Again, nothing special. We have broken one multiplication of n-bit numbers into 4 multiplications of $n/2$-bit numbers which has no effect on the running time. But consider the middle term $x_1 y_2 + x_2 y_1$. Observe that

$$x_1 y_2 + x_2 y_1 = (x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2.$$

The equation above rewrites two products as three, which seems like a step backwards. *Except two of them, $x_1 y_1$ and $x_2 y_2$, are already being computed.* With only *one more multiplication* (of $(x_1 + x_2)(y_1 + y_2)$), plus some $O(n)$-time addition and subtraction, we replace *two multiplications* in $x_1 y_2 + x_2 y_1$.[1]

Let's make this concrete in an algorithm. We are designing a recursive algorithm so first we declare our recursive spec / induction hypothesis.

> `multiply(`$x[1..n]$`,`$y[1..n]$`)` = the ($2n$-bit integer) product of x and y (as n-bit integers).

In Fig. 4.1, we implement `multiply(`$x[1..n]$`,`$y[1..n]$`)` as a divide-and-conquer algorithm leveraging the reduction to 3 integer multiplications of (roughly) half the bits.

[1] Note that $(x_1 + x_2)(y_1 + y_2)$ multiples two $(n/2 + 1)$-bit integers, instead of $n/2$, which makes our analysis a little messier.

`multiply(`$x[1..n]$`,`$y[1..n]$`)`

1. If $n = 0$ then return 0.
2. If $n = 1$ then return $x[1] * y[1]$.
3. If n is not even then pad x and y with a leading 0, making n even. // $O(n)$.
4. Let $x_1 = x[1..n/2]$, $x_2 = [n/2 + 1..n]$, $y_1 = y[1..n/2]$, and $y_2 = y[n/2 + 1..n]$ (as $n/2$-bit integers). // $O(n)$.
5. Let $x_3[1..n/2 + 1] = x_1 + x_2$ and $y_3[1..n/2 + 1] = (y_1 + y_2)$ (as $(n/2 + 1)$-bit integers). // $O(n)$.
6. Let $Z_1 =$ `multiply(`x_1`,`y_1`)`, $Z_2 =$ `multiply(`x_2`,`y_2`)`, and $Z_3 =$ `multiply(`x_3`,`y_3`)`. // $3T((n + 2)/2)$.

/* *Note that multiplying by 2^n is just a bit-shift and takes $O(n)$ time.* */

7. Return $2^n * Z_1 + 2^{n/2}(Z_3 - Z_1 - Z_2) + Z_2$.

Figure 4.1: Karatsuba's algorithm for integer multiplication.

4.1.1 Running time analysis

Let $T(n)$ denote the running time of `multiply(`$x[1..n]$`,`$y[1..n]$`)` on two n-bit integers. As annotated in the code, the algorithm consists of three recursive calls to multiply integers with at most $n/2+2$ bits, plus $O(n)$ additional work. This gives the recurrence

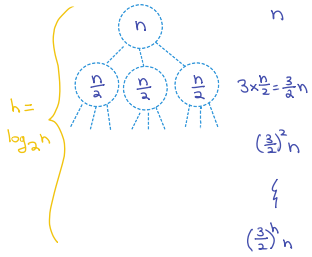
$$T(n) \leq 3T(n/2 + 2) + O(n) \tag{4.1}$$

for n larger than a constant (e.g., 5), and $O(1)$ time for n below this constant. As discussed in Section 1.2 on page 25, we can omit the additive factor of 2, and instead consider the cleaner recurrence

$$S(n) \leq 3S(n/2) + O(n) \tag{4.2}$$

A quick sketch of a recursion tree for $S(n)$ shows that:

1. There are 3^k problems on the kth level each contributing $n/2^k$ work. This gives $(3/2)^k n$ work per level.
2. The height h is bounded above by $h \leq \log_2(n) + O(1)$.



We have

$$S(n) \leq \sum_{i=1}^{h} \begin{pmatrix} \text{total work} \\ \text{on level } i \end{pmatrix} = \sum_{i=1}^{h} \left(\frac{3}{2}\right)^i n \overset{\text{(a)}}{=} O\left(\left(\frac{3}{2}\right)^h n\right)$$
$$= O\left(\left(\frac{3}{2}\right)^{\log_2(n)} n\right) = O\left(n^{\log_2(3)}\right) \approx O\left(n^{1.585}\right).$$

Here (a) is because the sum of coefficients in a geometrically increasing sequence is dominated by the leading term (cf. Section C.1.3).

4.2 Matrix multiplication

Let A and B be two $n \times n$ matrices. We can compute the product AB in $O(n^3)$ based on the formula

$$(AB)_{ij} = \sum_k A_{ik} B_{kj}.$$

In 1969, Strassen [Str69] showed that one can compute AB faster by leveraging divide-and-conquer, which was very surprising. Write

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ and } B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

where $A_{11}, \ldots, B_{2,2} \in \mathbb{R}^{n/2 \times n/2}$ are all square matrices with half the dimension. Recall that

$$AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

This divides AB into 8 matrix multiplications of smaller matrices. Strassen showed that in fact 7 multiplications suffice. We describe the scheme momentarily. First we observe that this gives a recursion of the form

$$T(n) = 7T(n/2) + O(n).$$

A quick sketch of a recursion tree reveals:

1. The kth level has 7^k subproblems each with $O\big(n/2^k\big)$ work.
2. There are at most $\lceil \log n \rceil$ levels.

Thus the total running time is

$$O\left(\sum_{i=1}^{\log(n)} \left(\frac{7}{2}\right)^k n\right) = O\big(n^{\log_2(7)}\big) \approx O\big(n^{2.8074}\big).$$

We will describe Strassen's algorithm for the sake of completeness but by no means do we claim it is obvious, even in hindsight. Consider the following 7 products.

$$(A_{11} + A_{22})(B_{11} + B_{22}) = A_{11}B_{11} + A_{22}B_{11} + A_{11}B_{22} + A_{11}B_{22} \tag{4.3}$$
$$(A_{11} - A_{21})(B_{11} - B_{12}) = A_{11}B_{11} - A_{21}B_{11} - A_{11}B_{12} + A_{21}B_{12} \tag{4.4}$$
$$(A_{12} - A_{21})(B_{21} + B_{22}) = A_{12}B_{12} - A_{21}B_{21} - A_{21}B_{22} + A_{12}B_{22} \tag{4.5}$$
$$A_{11}(B_{12} - B_{22}) = A_{11}(B_{12} - B_{22}) \tag{4.6}$$
$$(A_{11} + A_{12})B_{22} = A_{11}B_{22} + A_{12}B_{22} \tag{4.7}$$
$$(A_{11} + A_{22})(B_{11} + B_{22}) = A_{11}B_{11} + A_{11}B_{11} + A_{22}B_{11} + A_{22}B_{22} \tag{4.8}$$
$$(A_{11} - A_{21})(B_{11} + B_{12}) = A_{11}B_{11} - A_{21}B_{11} + A_{11}B_{12} - A_{21}B_{12} \tag{4.9}$$

Then

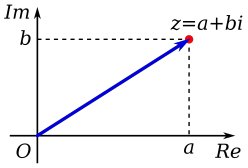
$$\begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$
$$= \begin{pmatrix} (4.3) + (4.4) - (4.7) + (4.5) & (4.6) + (4.7) \\ (4.8) + (4.4) & (4.6) + (4.3) - (4.8) - (4.9) \end{pmatrix}$$

Even more clever combinations and strategies have driven the exponent down further. At the time of writing, the best running time was $O(n^\omega)$ for $\omega \approx 2.37$ [AW20]. It is conjectured that ω will eventually be driven down to 2 (a natural lower bound). Admittedly, the theoretically best algorithms are so complicated that they are not preferred in practice. (The hidden constant strikes back!)

4.3 Fourier transforms

Complex numbers. The discrete Fourier transform is an operation applied to polynomials that takes place in the complex plane $\mathbb{C}$. Here we will present the minimal algebraic facts regarding $\mathbb{C}$. Recall that a complex number $x \in \mathbb{C}$ has the form

$$x = a + \imath b,$$



The complex plane [IW].

where $a, b \in \mathbb{R}$ are real-valued numbers. a is called the "real part" of x and b is called the "complex part" of b. $\imath$ denotes "imaginary i", defined by $\imath^2 = -1$. We have

$$(a + \imath b)(c + \imath d) = ac + (bc + ad)\imath + bd\imath^2 \overset{(a)}{=} ac + (bc + ad)\imath - bd;$$

here (a) applies the identity $\imath^2 = -1$. With this, polynomials

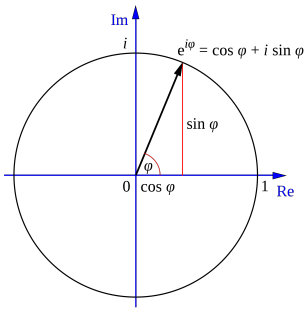
$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

are well defined for $x \in \mathbb{C}$. So are functions that can be expressed as convergent power series, such as the exponential function:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Euler's formula gives a geometric interpretation of the exponential function:

$$e^{\theta\imath} = \cos\theta + \imath \sin\theta.$$



Euler's formula [GW].

From this we have Euler's identity,

$$e^{\pi\imath} = -1,$$

as well as the identity

$$e^{2\pi\imath} = 1.$$

Roots of unity. Now, let $n \in \mathbb{N}$ be fixed. An *nth root of unity* is any (complex) number ω satisfying

$$\omega^n = 1.$$

For example, 1 is an nth root of unity for any n, and -1 is an nth root of unity for any even n. $\imath$ is a 4th root of unity. More generally,

$$\omega_n \overset{\text{def}}{=} e^{2\pi\imath/n}$$

is an nth root of unity, as well as ω_n^j for any $j \in \mathbb{Z}$. In fact, the n distinct powers of ω_n,

$$1 = \omega_n^0,\ \omega_n,\ \omega_n^2,\ \ldots,\ \omega_n^{n-1}$$

give all the nth roots of unity.

The discrete Fourier transform. The *(nth) discrete Fourier transform*, denoted F_n, takes as input a polynomial

$$p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$$

represented by its coefficients

$$a_0, \ldots, a_{n-1} \in \mathbb{C},$$

and produces the vector obtained by evaluating $p(x)$ at each of the n roots of unity.[2] Here, identifying p with its coefficients $a \in \mathbb{C}^n$, we write

$$F_n a \stackrel{\text{def}}{=} \Big(p(1), p(\omega_n), p(\omega_n^2), \ldots, p(\omega_n^{n-1})\Big).$$

The *conjugate (nth) discrete Fourier transform*, denoted $F_n^* a$, instead lists the values for the powers of ω_n^{-1}:

$$F_n^* a \stackrel{\text{def}}{=} \Big(p(1), p(\omega_n^{-1}), p(\omega_n^{-2}), \ldots, p(\omega_n^{1-n})\Big)$$

Since $\omega_n^{-1} = \omega_n^{n-1}$, F_n^* actually evaluates the same points as $F_n a$, but the order of values is partly reversed.

Later we will show that F_n and F_n^* are nearly inverse to one another, in the following sense. In particular, they can each be used to "undo" the other.

Fact 4.1. *For all $a \in \mathbb{C}^n$, $F_n^* F_n a = na$.*

What is the fastest way to compute F_n? The most direct way is to just evaluate each $p(\omega_n^i)$ separately, in $O(n^2)$ time overall. The *fast Fourier transform* algorithm applies divide and conquer to give the following improved running time when n is a power of 2.

Theorem 4.2. *For n a power of 2, F_n and F_n^* can be computed in $O(n \log n)$ time.*

4.3.1 Application: multiplying polynomials.

Below we will have a more technical discussion that proves both fact 4.1 and Theorem 4.2. But to motivate that discussion we first ask: why is F_n useful? We first mention that it is incredibly useful for signal processing and refer to the wikipedia article for more on this application.

[2] We mention that there are other "continuous" variants for Fourier transforms that we do not discuss.

As another basic application that requires no new definitions, consider the problem of multiplying two degree-$(n-1)$ polynomials $p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ and $q(x) = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}$, represented by coefficients $a, b \in \mathbb{C}^n$ in respectively. Without loss of generality we may assume n is a power of 2. (Otherwise pad a and b with 0's.) Let $r(x) = p(x)q(x)$ be their product, with coefficients $c \in \mathbb{C}^{2n}$. We can compute the coefficients of $r(x)$ explicitly in $O(n^2)$ time, by take the appropriate sums of products of coefficients of p and q. For example, the coefficient c_k corresponding to x^k in r is given by the *convolution*

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

This is not a complicated formula but computationally it does represent a loop of size k, where k can be as large as $2n$. Using the convolution above to compute $r = pq$, we take $O(n)$ time per coefficient c_k, and $O(n^2)$ time total.

Instead of focusing on the coefficients of r, let us consider its Fourier transform $F_{2n}r$. The values of $r(x) = p(x)q(x)$ at the $(2n)$th roots of unity ω_{2n}^i are given by

$$r(\omega_{2n}^i) = p(\omega_{2n}^i)q(\omega_{2n}^i)$$

for each i. Each value takes $O(1)$ time given the values on the RHS. Thus, given $F_{2n}p$ and $F_{2n}q$, computing $F_{2n}r$ takes only $O(n)$ time.

Now recall that F_{2n} computes all $2n$ roots of unity, and F_{2n}^* effectively reverses the process and recovers the polynomial (times a factor of $2n$) from the values at the roots of unity. We can compute $r = pq$ much more quickly by (a) taking the Fourier transforms of p and q, (b) multiplying the values together to get the Fourier transform of r, and then (c) using the conjugate Fourier transform (and dividing by $2n$) to get the coefficients of r. By the fast Fourier transform (Theorem 4.2), the whole procedure takes only $O(n \log n)$ time!

$$(p, q) \xrightarrow{O(n \log n)} (F_{2n}p, F_{2n}q) \xrightarrow{O(n)} F_{2n}r \xrightarrow{O(n \log n)} F_{2n}^* F_{2n} r = 2nr$$

Corollary 4.3. *Two polynomials of degree $n-1$ can be multiplied in $O(n \log n)$ time.*

We also point out that the fastest integer multiplication algorithms are based on the Fourier transform. Indeed, multiplying integers is not so different from multiplying polynomials – think of the bits of an integer in binary notation as the n coefficients of a degree-n polynomial.

4.3.2 The fast Fourier transform

We defer the proof of fact 4.1 and first discuss the divide-and-conquer algorithms for F_n and F_n^*, which is the main pedagogical point. As noted above, F_n^* is a rearrangement of the coordinates produced by F_n, so it suffices to only present an algorithm for F_n.

Let $p(x)$ be a degree $(n-1)$ polynomial; with coefficients $a_0, \ldots, a_{n-1} \in \mathbb{C}$:

$$p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}.$$

For ease of discussion let us assume that n is a power of 2. Our goal is to compute the vector of values

$$F_n a \stackrel{\text{def}}{=} \Big(p(1),\, p(\omega_n),\, p(\omega_n^2),\, ...,\, p(\omega_n^{n-1})\Big)$$

As always, we start with a recursive spec for our algorithm.

> `Fourier(`$a_0, \ldots, a_{n-1}$`)`: Given n values $a \in \mathbb{C}^n$, where n is a power of 2, returns the nth discrete Fourier transform
>
> $$F_n a = \Big(p(1), p(\omega_n), p(\omega_n^2), \ldots, p(\omega_n^{n-1})\Big),$$
>
> where $p(x)$ is the polynomial $p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$.

Now we implement `Fourier(`$a_0, \ldots, a_{2n-1}$`)` recursively. The divide and conquer argument exploits the following identity for ω_n (for all $n \in \mathbb{N}$):

$$\omega_{2n}^2 = \omega_n. \tag{4.10}$$

Indeed, recalling that $\omega_n \stackrel{\text{def}}{=} e^{2\pi\imath/n}$, we have

$$\omega_{2n}^2 = e^{(2\pi\imath/2n)^2} = e^{2\pi\imath/n} = \omega_n.$$

Now, consider $(F_{2n}a)_k$ for fixed $n \in \mathbb{N}$ and $a \in \mathbb{C}^{2n}$. We have

$$\begin{aligned}(F_{2n}a)_k &\stackrel{\text{(a)}}{=} a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1} \\ &\stackrel{\text{(b)}}{=} \Big(a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{(2n-2)}\Big) \\ &\quad + \Big(a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{(2n-1)}\Big).\end{aligned} \tag{4.11}$$

Here (a) is by definition of F_{2n}, and (b) groups the sum into one sum for the even-index coordinates, and another sum for the odd-index coordinates.

Consider the first group with the even-index coefficients. All the powers of ω_{2n} are even, so by applying the identity (4.10), we have

$$a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{(2n-2)} = a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1}.$$

Now the RHS can be interpreted as evaluating a degree $n-1$ polynomial with coefficients $a_0, a_2, \ldots, a_{2n-2}$ at ω_n^k. More specifically, if we let $a_{\text{even}} = (a_0, a_2, \ldots, a_{2n-2})$, then we have

$$a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1} = (F_n a_{\text{even}})_{k \bmod n}$$

– the nth Fourier transform of the even-index coefficients! (When $k > n$, $\omega_n^k = \omega_n^{k \bmod n}$.)

We can do something similar for the odd coefficients. Let $a_{\text{odd}} = (a_1, a_3, \ldots, a_{n-1})$ be the vector of odd coefficients, and consider the second sum in (4.11). We have

$$\begin{aligned}
&a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{(2n-1)} \\
&= \omega_{2n}^k\Big(a_1 + a_3(\omega_n^k)^2 + a_5(\omega_n^k)^4 + \cdots + a_{2n-2}(\omega_n^k)^{n-1}\Big) \\
&= \omega_{2n}^k\Big(a_1 + a_3(\omega_n^k)^2 + a_5(\omega_n^k)^4 + \cdots + a_{2n-2}(\omega_n^k)^{n-1}\Big) \\
&= \omega_{2n}^k(F_n a_{\text{odd}})_{k \bmod n}.
\end{aligned}$$

Plugging all this back into (4.11), we have the clean identity

$$(F_{2n}a)_k = (F_n a_{\text{even}})_{k \bmod n} + \omega_{2n}^k(F_n a_{\text{odd}})_{k \bmod n}$$

for $k = 0, \ldots, n-1$. This equation reduces computing $F_{2n}a$ to computing $F_n a_{\text{even}}$ and $F_n a_{\text{odd}}$. Given $F_n a_{\text{even}}$ and $F_n a_{\text{odd}}$, we can compute $F_{2n}a$ in $O(n)$ time. Meanwhile $F_n a_{\text{even}}$ and $F_n a_{\text{odd}}$ can be computed recursively, where each subproblem has half as many coefficients.

We translate the discussion above to pseudocode in Fig. 4.2 on the following page.

We now analyze the running time of `Fourier`$(a_0, \ldots, a_{n-1})$. As annotated in the pseudocode, `Fourier`$(a_0, \ldots, a_{n-1})$ consists of 2 recursive calls with half the number of coefficients, plus $O(n)$ work. Thus the running time obeys the recurrence

$$\begin{aligned}
T(1) &= O(1) \\
T(n) &= 2T(n/2) + O(n).
\end{aligned}$$

This is the same recurrence as in `merge-sort` (cf. Section 1.2) and is solved by

$$T(n) = O(n \log n).$$

This completes the proof of Theorem 4.2.

`Fourier`$(a_0, \ldots, a_{n-1})$

/* *Given n values $a \in \mathbb{C}^n$, where n is a power of 2, returns the nth discrete Fourier transform*

$$F_n a = \big(p(1), p(\omega_n), p(\omega_n^2), \ldots, p(\omega_n^{n-1})\big),$$

where $p(x)$ is the polynomial $p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$. */

1. If $n = 0$ then return an empty list ( ).
2. If $n = 1$ then return the singleton list (a_0).
3. $a_{\text{even}} = (a_0, a_2, a_4, \ldots, a_{n-2})$ be the vector of $n/2$ even-index coefficients and let $a_{\text{odd}} = (a_1, a_3, a_5, \ldots, a_{n-1})$ be the vector of $n/2$ odd-index coefficients. // $O(n)$
4. Let $(y_0, \ldots, y_{n/2-1}) =$ `Fourier`(a_{even}) // $T(n/2)$
5. Let $(z_0, \ldots, z_{n/2-1}) =$ `Fourier`(a_{odd}). // $T(n/2)$
6. For $k = 0, 1, \ldots, n-1$, let $x_k = y_{k'} + \omega_n^k z_{k'}$ where $k' = k \bmod n/2$. // $O(n)$
7. Return $(x_0, \ldots, x_{n-1})$.

Figure 4.2: The fast Fourier transform.

4.3.3 Inverting F_n and F_n^*

Lastly we prove fact 4.1, which described a near-inverse relationship between F_n and F_n^*. Fact 4.1 is just as critical to applications as the $O(n \log n)$ algorithm above, and this discussion was deferred to the end only for pedagogical reasons. (In particular the discussion is purely mathematical.) We first require the following identity.

Lemma 4.4. *For any nth root of unity ω except for $\omega = 1$, we have*

$$1 + \omega + \omega^2 + \cdots + \omega^{n-1} = 0.$$

For $\omega = 1$, the LHS is n.

Proof. We have

$$(1 - \omega)\Big(1 + \omega + \omega^2 + \cdots + \omega^{n-1}\Big) = 1 - \omega^n = 0,$$

so either $\omega = 1$ or $1 + \omega + \cdots + \omega^{n-1} = 0$. □

Now we prove fact 4.1. We restate the claim below for the reader's convenience.

Fact 4.1. *For all $a \in \mathbb{C}^n$, $F_n^* F_n a = na$.*

Proof. Take any $a \in \mathbb{C}^n$; we want to show that $F_n^* F_n a = na$. Indeed, for each coordinate $h \in \{0, \dots, n-1\}$, we have

$$(F_n^* F_n a)_h \overset{\text{(a)}}{=} \sum_{j=0}^{n-1} (F_n a)_j (\omega_n^{-h})^j \overset{\text{(b)}}{=} \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} a_k (\omega_n^j)^k \right) (\omega_n^{-h})^j$$
$$\overset{\text{(c)}}{=} \sum_{k=0}^{n-1} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j \overset{\text{(d)}}{=} n a_h.$$

Here (a) and (b) expand out the definitions of F_n^* and F_n. (c) interchanges sums. (d) is by Lemma 4.4: unless $k - h = 0$ (or some other multiple of n), we have $\sum_{j=0}^{n-1} (\omega_n^{k-h})^j = 0$. □

One implication of fact 4.1 is as follows.

Corollary 4.5. *A polynomial $p(x)$ of degree $n-1$ is uniquely identified by its value at the n roots of unity $p(1), p(\omega_n), \dots, p(\omega_n^{n-1})$.*

The above is a special case of the fundamental theorem of algebra.

4.4 Additional notes and references

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 Lecture 11. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 Lecture 12. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

4.5 Exercises

Exercise 4.1. Above we observed that two degree n polynomials $p(x)$ and $q(x)$ can be multiplied in $O(n \log n)$ time. Consider the case when the degrees are very unbalanced; say, $p(x)$ has degree n and $q(x)$ has degree k for $k \ll n$. Design and analyze an algorithm computing the product $p(x)q(x)$ in $O(n \log k)$ time.

Exercise 4.2. Let $p(x, y) = \sum_{i=0}^{m} \sum_{j=0}^{n} a_{ij} x^i y^j$ and $q(x, y) = \sum_{i=0}^{m} \sum_{j=0}^{n} b_{ij} x^i y^j$ be two multivariate polynomials each with maximum degree m in x and maximum degree n in y. Consider the product $r(x, y) = p(x, y)q(x, y)$, which has maximum degree $2m$ in x and maximum degree $2n$ in y. It is easy to see that r can be computed in $O(m^2 n^2)$ time with nested loops. Design and analyze a faster algorithm computing the product r.

Exercise 4.3. In Corollary 4.3, we saw that FFT can multiply two degree-n polynomials in $O(n \log n)$ time. Now consider the problem of multiplying n degree-1 polynomials. The input consists of n linear polynomials

$$a_1 + b_1 x, , \; a_2 + b_2 x, \; \ldots, \; a_n + b_n x,$$

and the output is the coefficients of their product, a degree n polynomial. Design and analyze an algorithm, as fast as possible, for this problem.

Exercise 4.4. Karatsuba's algorithm gave an $O(n^{.585})$ time algorithm for multiplying two n-bit integers. Design and analyze a nearly linear time algorithm (i.e., $O\left(n \log^{O(1)} n\right)$ time) for integer multiplication.

Exercise 4.5. Let X and Y be two sets of integers. We define the *unique sums* of X and Y as the set of integers of the form

$$z = x + y$$

where $x \in X$, $y \in Y$, and the choice of $(x, y) \in X \times Y$ is unique.[3] You may assume that all the integers in X are distinct (amongst themselves), and that all the integers in Y are distinct (amongst themselves).[4]

1. (2 pt.) Suppose X and Y each have n integers. Design and analyze a $O(n^2 \log(n))$ time algorithm to compute the unique sums of X and Y.

2. (6 pt.) Suppose all the integers in X are Y are between 0 and M for a fixed value $M \in \mathbb{N}$. Design and analyze a $O(M \log M)$ time algorithm to compute the unique sums of X and Y.[5]

3. (2 pt.) Suppose that now we had k sets $X_1, \ldots, X_k$, each consisting of integers between 1 and M. (You may again assume no duplicates within each X_i.) A unique sum of $X_1, \ldots, X_k$ is defined as an integer of the form

   $$x_1 + \cdots + x_k,$$

   where $x_1 \in X_1$, $x_2 \in X_2$, $\ldots$, $x_k \in X_k$, and the choice of $(x_1, \ldots, x_k) \in X_1 \times \cdots \times X_k$ is unique. Design and analyze an algorithm that computes the unique sums of $X_1, \ldots, X_k$ in $O(kM \log(Mk) \log(k))$ time. You may assume for simplicity that k is a power of 2.[6]

Hint: For part 2, try "changing the input" to an algorithm from Section 4.3 (that is not the FFT, at least directly).

Exercise 4.6. For two polynomials $p(x) = a_0 + a_1 x + \cdots + a_n x^n$ and $q(x) = b_0 + b_1 x + \cdots + b_{n-1} x^n$, their product can be written as

$$p(x)q(x) = \sum_{i,j} a_i b_j x^{i+j}$$

[3] That is, for fixed z, there is only one choice of $x \in X$ and one choice of $y \in Y$ such that $z = x + y$.

[4] For example, for $X = \{0, 1\}$ and $Y = \{0, 1\}$, the unique sums are $\{0, 2\}$. 1 can be obtained by $0 + 1$ or $1 + 0$ so it is not a unique sum.

[5] It might be helpful to work through very simple examples such as $X = Y = \{1\}$ and $X = Y = \{0, 1\}$.

[6] It might help to work through $k = 4$ to build your intuition.

where i, j both range from 0 through n. Suppose we instead wanted to compute the following "skew-product",

$$\sum_{i<j} a_i b_j x^{i+j},$$

where again i and j both range from 0 through n, but restricted to $i < j$. Design and analyze a fast algorithm for computing the skew-product of two polynomials.

Exercise 4.7. A Hadamard matrix is a family of square $\{-1, 1\}$-matrices H_0, H_1, H_2 defined recursively by:

$$H_0 = \begin{pmatrix} 1 \end{pmatrix} \in \mathbb{R}^{1\times 1}$$
$$H_k = \begin{pmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{pmatrix} \in \mathbb{R}^{2^k \times 2^k}, \text{ for } k \geq 1.$$

1. Prove that for all k, $H_k H_k^\mathsf{T} = H_k^\mathsf{T} H_k = 2^k I$, where I is the identity matrix.
2. Design and analyze a fast algorithm that, given $x \in \mathbb{R}^n$ for $n = 2^k$, computes the vector $H_k x \in \mathbb{R}^n$.

Exercise 4.8. Let $X[1..n]$ be a sequence of integers in the range $[-k, k]$. A *consecutive sum* is defined as the sum of a consecutive subsequence of the form $X[i..j]$. Consider the problem of computing the number of *distinct* consecutive sums in X. Of course we can compute this in $O(n^2 \log n)$ or $O(n^2 + nk)$ time, by generating all prefix sums $X[1..i]$, and then marking all consecutive sums $X[i..j] = X[1..j] - X[1..i]$ in a lookup table (implemented as a balanced tree or as an $n \times k$ array). But suppose k is much smaller than n. Design and analyze an algorithm computing the number of distinct consecutive sums in $O(n \operatorname{poly}(k, \log(n)))$ time. (Of course, the smaller the $\operatorname{poly}(k, \log n)$ term, the better.)

Chapter 5

Sequential optimization with caching

5.1 Edit distance: An algorithmic metric for strings

You're a software engineer debugging a massive codebase tracked by `git`. You refactor a module, moving functions around, inserting new code, deleting dead code. Ready to check in, you run `git diff`, and it shows you a clean summary of all the changes. How? Does it optimize for the cleanest summary?

Now you're a virologist. A new strain of a virus has emerged. Where did it come from? Did it recently branch off from other known strains? How fast is it mutating? Public health policy, vaccine design, and treatment all depend on the answers. The data is a library of genetic sequences of viral strains, each a string of tens of thousands of characters over the alphabet $\{A, C, G, T\}$. A comprehensive analysis requires measuring the pairwise similarity among all the strains under consideration.

Both problems require a way to measure the distance between two strings. The *Hamming distance*[1] counts the number of differing characters in two strings of the same length. For a fixed alphabet $\mathcal{A}$ and two strings $x, y \in \mathcal{A}^k$ of the same length, the Hamming distance is

$$\mathsf{Hamming}(x, y) = |\{i \in [k] : x_i \neq y_i\}|.$$

Hamming distance is symmetric, and satisfies the triangle inequality:

$$\mathsf{Hamming}(x, y) \leq \mathsf{Hamming}(x, z) + \mathsf{Hamming}(z, y)$$

for any three strings $x, y, z \in \{0, 1\}^k$. (Why?) Together with the fact that $\mathsf{Hamming}(x, y) = 0$ iff $x = y$, these make Hamming distance a *metric*. Hamming distance is the metric of choice in coding theory.

[1] Richard Hamming (1915 – 1998) was an American mathematician and computer scientist who, among many other contributions, invented error correcting codes, and was an early pioneer of computing at Bell Labs. His well-known talk "You and Your Research" (pdf, video) is highly recommended.

But it has limitations as well. First, it only compares strings of the same length. Source files and genetic sequences rarely have the same length. More fundamentally, Hamming distance is extremely sensitive to shifts. Insert a line in the middle of a program and Hamming distance sees a total rewrite. A mutation inserts one letter in the middle of a genome and Hamming distance acts like it's a completely different species.

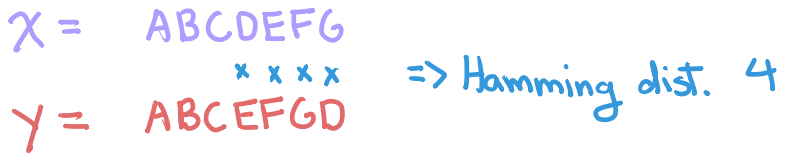


Figure 5.1: An example of Hamming distance.

The *edit distance* between two strings $s, t \in \mathcal{A}^*$ is the minimum number of *insertions*, *deletions*, and *substitutions* required to convert one into the other.[2] There are three edit operations:

1. An insertion inserts a single letter at some point in the string.
2. A deletion deletes a single letter from the string.
3. A substitution replaces one letter in the string with another.

Edit distance is well-defined. The distance between two strings x and y is bounded below by 0 and bounded above by the length of x plus the length of y, so it can always be computed by brute force.

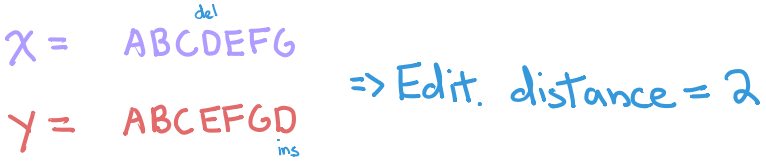


Figure 5.2: An example of edit distance

Edit distance is also a metric. It is symmetric (why?), and satisfies the triangle inequality:

$$\texttt{edit}(x, y) \leq \texttt{edit}(x, z) + \texttt{edit}(z, y).$$

[2]Here $\mathcal{A}^*$ denotes the family of all finite strings with characters from $\mathcal{A}$. The $*$ indicates that the string may be of any length.

(Again, why?)

Edit distance may be the right model, but can we compute it efficiently? We start with a recursive spec:

> $\texttt{edit}(x[1..m], y[1..n])$: Given two strings x and y, returns the minimum number of edit operations needed to convert x into y.

Base cases first. If both strings are empty, the edit distance is 0:

$$\texttt{edit}(\emptyset, \emptyset) = 0,$$

where $\emptyset$ denotes the empty string. If only one of the strings is empty, then the edit distance is the length of the nonempty string:

$$\texttt{edit}(x[1..k], \emptyset) = k, \qquad \texttt{edit}(\emptyset, y[1..\ell]) = \ell.$$

Suppose both strings are nonempty and $x[1] \neq y[1]$. Any edit sequence from x to y must, as its first move, either

(a) insert $y[1]$ before $x[1]$,
(b) delete $x[1]$, or
(c) substitute $y[1]$ for $x[1]$.

If not, the first character would remain $x[1]$, not matching $y[1]$.

Which choice is best? Substitution seems efficient — it addresses both $x[1]$ and $y[1]$ in one edit. But if $x[1] = y[2]$, substituting disrupts a free match later. The downstream effects are tangled and the best choice is unclear. A safe, non-thinking option tries all three. *If* $x[1] \neq y[1]$:

$$\texttt{edit}(x[1..k], y[1..\ell]) = 1 + \min \begin{cases} \texttt{edit}(x[2..k], y[1..\ell]), \\ \texttt{edit}(x[1..k], y[2..\ell]), \\ \texttt{edit}(x[2..k], y[2..\ell]). \end{cases}$$

The first line in the "min" models deleting $x[1]$, the second models inserting $y[1]$, and the third models substituting $y[1]$ for $x[1]$. We don't know which one of the choices is right, but we have deduced that it must be one of them.

If $x[1] = y[1]$, we can match them and recurse on the rest. Is matching always optimal? It might seem like it, but one has to prove it. Alternatively, one can try all options and skip the proof. *If* $x[1] = y[1]$:

$$\texttt{edit}(x[1..k], y[1..\ell]) = \min \begin{cases} \texttt{edit}(x[2..k], y[2..\ell]), \\ 1 + \texttt{edit}(x[2..k], y[1..\ell]), \\ 1 + \texttt{edit}(x[1..k], y[2..\ell]), \end{cases}$$

`edit`$(x[1..k], y[1..\ell])$

1. If $k = 0$ or $\ell = 0$ then return $k + \ell$.
2. If $x[1] \neq y[1]$ then return the minimum of
   - A. $1 + \texttt{edit}(x[2..k], y[1..\ell])$ // *delete* $x[1]$
   - B. $1 + \texttt{edit}(x[1..k], y[2..\ell])$ // *insert* $y[1]$
   - C. $1 + \texttt{edit}(x[2..k], y[2..\ell])$ // *replace* $x[1]$ *with* $y[1]$
3. If $x[1] = y[1]$ then return the minimum of
   - A. $\texttt{edit}(x[2..k], y[2..\ell])$ // *do nothing*
   - B. $1 + \texttt{edit}(x[1..k], y[2..\ell])$ // *insert* $y[1]$
   - C. $1 + \texttt{edit}(x[2..k], y[1..\ell])$ // *delete* $x[1]$

Figure 5.3: A slow, recursive algorithm for computing the edit distance.

This covers all cases. The full pseudocode is given in Fig. 5.3.

One can show by induction on $k + \ell$ that the pseudocode is correct. This is not surprising. It is essentially brute force, searching all possible edit sequences from x to y. For the same reason, it takes $O(3^{m+n})$ time.

Can we make this algorithm run in polynomial time? Essentially: can we retain the correctness of recursive enumeration, without paying for the running time?

Each subproblem in the recursive algorithm is defined by two suffixes, $x[i..m]$ and $y[j..n]$. The algorithm revisits and solves the same subproblem exponentially many times. This is obviously excessive and the fix is equally obvious. Save the answers. Never solve the same subproblem twice.

Fix the input strings $x[1..m]$ and $y[1..n]$ (globally). For $i, j \in \mathbb{N}$, we define

> $\texttt{edit}(i, j)$ = the minimum number of edit operations to convert $x[i..m]$ into $y[j..n]$.

Then $\texttt{edit}(1, 1)$ gives the edit distance between $x[1..m]$ and $y[1..n]$.

`edit`(i,j)

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j + 1$.
3. If $j > n$ then return $m - i + 1$.
4. If $x[i] \neq y[j]$ then return $\min\{1 + \texttt{edit}(i+1, j), 1 + \texttt{edit}(i, j+1), 1 + \texttt{edit}(i+1, j+1)\}$.
5. Otherwise return $\min\{\texttt{edit}(i+1, j+1), 1 + \texttt{edit}(i+1, j), 1 + \texttt{edit}(i, j+1)\}$.

We cache (save) the answers to all our recursive calls in a table $A[1..m, 1..n]$, initially set to `null`. In every recursive call, we first check A. If the answer is already stored, we return it. If not, we compute it and store it before returning. See Fig. 5.4.

`cached-edit`(i,j)

/* *Implements the same recursive spec as* `edit`(i,j)*, but assumes there is an* $m \times n$ *array* $A[1..m, 1..n]$ *("A" for "Answer") allocated in global memory, with all entries initially set to* `null`. */

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j + 1$.
3. If $j > n$ then return $m - i + 1$.
4. If $A[i, j]$ is `null`:
   A. If $x[i] \neq y[j]$ then set
   $$A[i,j] = \min\{1 + \texttt{cached-edit}(i+1, j), 1 + \texttt{cached-edit}(i, j+1), 1 + \texttt{cached-edit}(i+1, j+1)\}.$$
   B. Otherwise set
   $$A[i,j] = \min\{\texttt{cached-edit}(i+1, j+1), 1 + \texttt{cached-edit}(i+1, j), 1 + \texttt{cached-edit}(i, j+1)\}.$$
5. Return $A[i, j]$.

Figure 5.4: Computing edit distance recursively and with caching.

The algorithm is conceptually recursive, but the cached version is not modeled correctly by recursion trees.

We organize our runtime analysis by *subproblem*. Observe that the parameters i and j range from 1 to $m + 1$ and from 1 to $n + 1$, respectively. Thus, the *number of*

subproblems — i.e., the number of choices of i and j — is

$$(\#\text{ subproblems}) \leq O(mn).$$

Each subproblem takes $O(1)$ time excluding recursive subcalls. That is,

$$\left(\begin{array}{c}\text{time per subproblem}\\ \text{excl. recursive calls}\end{array}\right) = O(1).$$

All put together, we have

$$(\text{total runtime}) \leq (\#\text{ subproblems})\left(\begin{array}{c}\text{time per subproblem}\\ \text{excl. recursive calls}\end{array}\right) \leq O(mn).$$

With a little extra memory, we have reduced our running time from exponential to quadratic.

5.2 Caching, a.k.a. dynamic programming: a meta-algorithm

The difference in running time comes from the caching trick. It is a general technique that can be extended to all sorts of recursions, and is more commonly known as *dynamic programming*. The recipe is simple. Recursion, plus a little bit of memory to avoid redundancy, makes for more efficient algorithms.

The conversion from recursion to caching was mechanical. The critical thinking lies in the recursive spec. The following steps structure our solutions around this principle.

1. A concise specification of the recursive function you are trying to implement, such as our one sentence description of `edit(`i`,`j`)`. It should be clear what is the input, and what is the desired output. The parameters may be indices to objects defined globally; which objects these refer to should be made clear.

   The specification provides the induction hypothesis that justifies the correctness of your algorithm. It is the most important step.

2. Some kind of pseudocode or formula that implements the specification, such as the five line code for `edit(`i`,`k`)` on page 103.

3. Some explicit *mention* of the use of caching or dynamic programming to avoid recomputing subproblems.

4. An analysis of the total running time, which often breaks down as the number of subproblems times the amount of time per subproblem, excluding recursive calls.

5. A brief description of how to use the recursive function to produce the answer we are actually looking for. For edit distance, `edit(1,1)` returns the edit distance of $x[1..m]$ and $y[1..n]$.

   (There will be problems where there is a bigger difference between the recursive spec and what the problem is asking for, which makes this step nontrivial.)

6. When needed, some justification or proof for the correctness of the algorithm.

   (In general the overall proof should follow by induction, using the recursive spec for the induction hypothesis. This high-level setup is boilerplate. Sometimes some justification is needed to explain the choices the algorithm is making, such as how we showed the algorithm must make one of the three edit operations when $x[i] \neq y[j]$.)

A sample solution for edit distance, fulfilling the steps above, is given on the following page.

The recursive spec is the most important step. A clearer spec gives a more useful induction hypothesis, and a more straightforward implementation. Conversely, if it is difficult to implement the code in step 2, this may indicate that the specification is not quite right. Sometimes, introducing another parameter, or strengthening the guarantees of the recursive spec, can unlock the implementation.

Once steps 1 and 2 are done correctly and clearly, then the remaining steps can be straightforward. In particular, the correctness of the algorithm is often self-evident and hardly requires a proof.[3] If one of the later steps feels awkward or difficult, then this might be best addressed by clarifying the specification or the pseudocode.

We require the above steps in all solutions. We do not require:

- Literally implementing the caching (which is mechanical and clutters the code).
- Iterating through the subproblems with loops. Recursion handles this automatically.
- Analyzing the space usage (which is bounded by the running time).

Bottom-up loop implementations are not accepted. Even in the lucky event that they produce the right answer, they obscure the structure, reasoning, and proof. Loops confuse execution with structure and do not reliably generalize to more complex problems.

[3] On tests or other situations where we favor expedience, we may not ask for a proof, and instead we will interpret your recursive spec as the induction hypothesis of a proof by induction and fill in the rest of the details ourselves.

Sample solution.

Problem. Design and analyze an algorithm that, given two strings $x[1..m]$ and $y[1..n]$, computes the edit distance from x to y.

Solution.

1. Recursive spec / induction hypothesis. Let $x[1..m]$ and $y[1..n]$ be two strings. We define

$$\texttt{edit}(i,j) = \text{the minimum number of edit operations to convert } x[i..m] \text{ into } y[j..n].$$

2. Recursive implementation.

`edit(`i`,`j`)`

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $n - j + 1$.
3. If $j > n$ then return $m - i + 1$.
4. If $x[i] \neq y[j]$ then return $\min\{1 + \texttt{edit}(i+1, j), 1 + \texttt{edit}(i, j+1), 1 + \texttt{edit}(i+1, j+1)\}$.
5. Otherwise return $\min\{\texttt{edit}(i+1, j+1), 1 + \texttt{edit}(i+1, j), 1 + \texttt{edit}(i, j+1)\}$.

3. Solving the original problem. $\texttt{edit}(1,1)$ gives the edit distance between $x[1..m]$ and $y[1..n]$.

4. Running time with caching / dynamic programming. We apply dynamic programming. There are $O(mn)$ subproblems, each of which takes $O(1)$ time excluding recursive subcalls. Thus the total running time is bounded by

$$(\#\text{ subproblems}) \begin{pmatrix} \text{time per} \\ \text{subproblem} \\ \text{excl. recursive calls} \end{pmatrix} \leq O(mn).$$

5. Proof. Let $\text{OPT}_{i,j}$ denote the minimum number of edit operations to convert $x[i..m]$ into $y[j..n]$. We claim that $\texttt{edit}(i,j) = \text{OPT}_{i,j}$ for all i and j, by induction on $\min\{m-i, n-j\}$. The base cases, where $i > m$ or $j > n$, are immediate. The only option is to insert or delete the remaining characters.

For the inductive step, $\texttt{edit}(i,j) \geq \text{OPT}_{i,j}$ because the algorithm returns the cost of some edit sequence. For the other direction, any edit sequence from $x[i..m]$ to $y[j..n]$ must begin with one of a few moves.

If $x[i] \neq y[j]$, the first move must be a deletion, insertion, or substitution. The best edit sequence starting with each move has the following costs:

- Delete $x[i]$: $1 + \text{OPT}_{i+1,j} = 1 + \texttt{edit}(i+1, j)$ by induction.

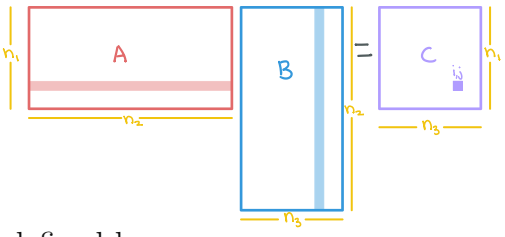
- Insert $y[j]$: $1 + \mathrm{OPT}_{i,j+1} = 1 + \texttt{edit}(i, j+1)$ by induction.
- Substitute $y[j]$ for $x[i]$: $1 + \mathrm{OPT}_{i+1,j+1} = 1 + \texttt{edit}(i+1, j+1)$ by induction.

The case $x[i] = y[j]$ is similar. The first move is a deletion, insertion, or free match. The best edit sequence starting with each move has the following costs:

- Delete $x[i]$: $1 + \mathrm{OPT}_{i+1,j} = 1 + \texttt{edit}(i+1, j)$ by induction.
- Insert $y[j]$: $1 + \mathrm{OPT}_{i,j+1} = 1 + \texttt{edit}(i, j+1)$ by induction.
- Match $x[i]$ to $y[j]$: $\mathrm{OPT}_{i+1,j+1} = \texttt{edit}(i+1, j+1)$ by induction.

In both cases, $\mathrm{OPT}_{i,j}$ is the minimum of the three options, which is exactly what $\texttt{edit}(i,j)$ computes. So $\texttt{edit}(i,j) \leq \mathrm{OPT}_{i,j}$.

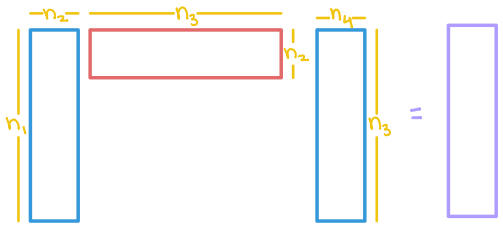
5.3 Sequential matrix multiplication



The composition $f(g(x))$ of two linear functions f and g is again a linear function. Every linear function can be represented by a matrix; e.g., $f(x) = Ax$ and $g(x) = Bx$. The composition $f(g(x))$ has the form $(AB)x$ where AB is the *matrix product*. More formally, for $A \in \mathbb{R}^{n_1 \times n_2}$ and $B \in \mathbb{R}^{n_2 \times n_3}$, the product AB is defined by

$$(AB)_{i,j} = \sum_{k=1}^{n_2} A_{i,k} B_{k,j}.$$

Each entry is a sum of n_2 terms, a zipping up along the inner dimension. Computing all $n_1 n_3$ entries takes $O(n_1 n_2 n_3)$ time.



Let $A_1 \in \mathbb{R}^{n_1 \times n_2}$ $A_2 \in \mathbb{R}^{n_2 \times n_3}$ and $A_3 \in \mathbb{R}^{n_3 \times n_4}$ be three matrices, and consider the matrix product

$$A_1 A_2 A_3.$$

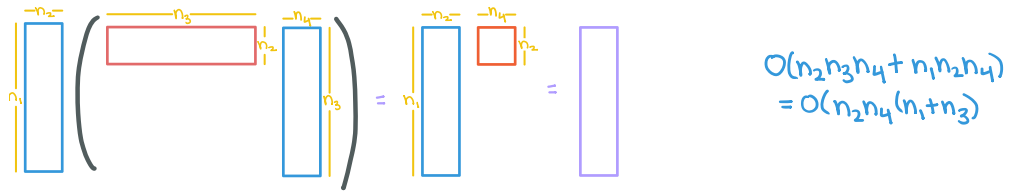
Matrix multiplication is associative:[4]

$$A_1(A_2 A_3) = (A_1 A_2) A_3.$$

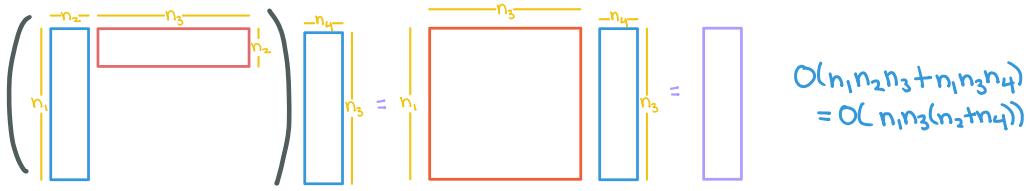
The product $A_1 A_2 A_3$ is unambiguous. But the *cost* of computing it depends on the order of operations.

[4]We can verify this directly: for $i_1 \in [n_1]$ and $i_4 \in [n_4]$, we have

$$(A_1(A_2 A_3))_{i_1,i_4} = \sum_{i_2=1}^{n_2} A_{1,i_1,i_2} (A_2 A_3)_{i_2,i_4} = \sum_{i_2=1}^{n_2} A_{1,i_1,i_2} \sum_{i_3=1}^{n_3} A_{i_2,i_3} A_{i_3,i_4} = \sum_{i_2=1}^{n_2} \sum_{i_3=1}^{n_3} A_{1,i_1,i_2} A_{i_2,i_3} A_{i_3,i_4};$$



Consider first computing the product as $A_1(A_2A_3)$. Computing (A_2A_3) takes $O(n_2n_3n_4)$ time and produces a $n_2 \times n_4$-dimension matrix. Multiplying $A_1(A_2A_3)$ then takes $O(n_1n_2n_4)$ time. Overall we spend $O(n_2n_3n_4 + n_1n_2n_4) = O((n_1+n_3)n_2n_4)$ time.



Now consider $(A_1A_2)A_3$. Computing (A_1A_2) takes $O(n_1n_2n_3)$ time and produces a $n_1 \times n_3$-dimension matrix. Multiplying $(A_1A_2)A_3$ takes $O(n_1n_3n_4)$ time. Together we spend $O(n_1n_2n_3 + n_1n_3n_4) = O(n_1n_3(n_2+n_4))$ total time.

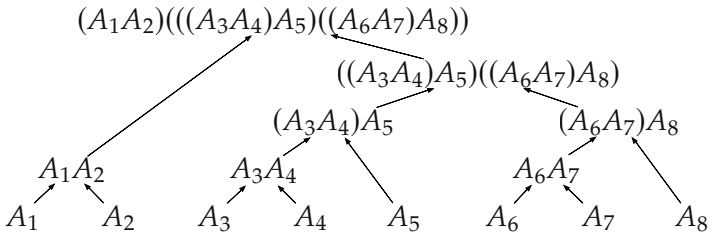
For example, if $n_1 = n_3 = m$ and $n_2 = n_4 = 1$, then $A_1(A_2A_3)$ takes $O(m)$ time and $(A_1A_2)A_3$ takes $O(m^2)$.

There are two ways to multiply three matrices, but exponentially many ways to multiply k matrices. Given k matrices $A_1, \ldots, A_k$ where $A_i \in \mathbb{R}^{n_i \times n_{i+1}}$, what is the fastest way to multiply them? (Try it.)

A multiplication strategy is really a parenthesization, like

$$(A_1A_2)(((A_3A_4)A_5)((A_6A_7)A_8))$$

We can also envision the multiplications as a tree:



similarly one can show that $((A_1A_2)A_3)_{i_1,i_4} = \sum_{i_2=1}^{n_2} \sum_{i_3=1}^{n_3} A_{1,i_1,i_2} A_{i_2,i_3} A_{i_3,i_4}$ by expanding out the appropriate sums.

Two natural greedy rules suggest themselves, but neither is optimal.

1. We can take a greedy bottom up approach, where in the first matrices, we take the product A_iA_{i+1} minimizing $n_in_{i+1}n_{i+2}$; replace A_i, A_{i+1} with their product (A_iA_{i+1}), and continue in the same fashion.

2. We can take a greedy top down approach, where in the first iteration, we identify the split $(A_1 \cdots A_i)(A_{i+1} \cdots A_k)$ minimizing $n_1n_in_{k+1}$, and then recursively compute the two halves $(A_1 \cdots A_i)$ and $(A_{i+1} \cdots A_k)$ in the same fashion.

First, the recursive spec. For k matrices $A_1, \ldots, A_k$, define

> `Fastest`(i, j) = the minimum number of operations to compute the matrix product $A_i \cdots A_j$.

The implementation is short.

`Fastest(`i`,`j`)`

1. If $i \geq j$ then return 0.
2. Return the minimum, over all $k = i, \ldots, j-1$, of

$$\texttt{Fastest(}i\texttt{,}k\texttt{)} + \texttt{Fastest(}k+1\texttt{,}j\texttt{)} + n_in_{k+1}n_j.$$

To justify the pseudocode, fix an instance `Fastest(`i`,`j`)`. Inductively (on the number of matrices) we assume that the subcalls to `Fastest` with fewer matrices are correct. In this case, for each k,

$$\texttt{Fastest(}i\texttt{,}k\texttt{)} + \texttt{Fastest(}k+1\texttt{,}j\texttt{)} + n_in_{k+1}n_j$$

gives the minimum number of multiplications among all those that have the split $(A_i \cdots A_k)(A_{k+1} \cdots A_j)$ at the top of the tree. Which k gives the best initial split? Who knows?! So instead we exhaustively test all choices of k and return the best one.

`Fastest(`1`,`k`)` is exponentially slow without caching. With caching, the running time is

$$(\#\text{ subproblems}) \times (\text{time per subproblem}) = k^2 \times O(k) = O\Big(k^3\Big).$$

`Fastest(`1`,`k`)` reports the minimum cost, not the strategy itself. To reconstruct the optimal parenthesization, record the choice of ℓ for each subproblem and follow the pointers from the root subproblem.

5.4 TEX

These notes (and hopefully yours too) are typeset using a system called LaTeX, built on top of TeX, a typesetting system designed and largely written by Donald Knuth.

In the 1970's, Knuth was writing his (still unfinished) series on *The Art of Computer Programming*, as the printing industry moved from hot metal to digital typesetting. The digital galley proofs of his book looked so bad that he built his own typesetting system: TeX.

TeX works like a compiler, not a word processor. A word processor typesets on the fly, like a typewriter. Knuth modeled typesetting as a global optimization problem, and the compiler solves it.

Knuth first described TeX in a 1977 memo[5]. One of its core problems is line-breaking: given a paragraph of text, where should the lines break? Too many words on a line and they crowd together. Too few and they drift apart.

One simple fix sets a minimum white space and breaks the line as soon as the next word would violate it. Myopic and local, each line break ignores the words that follow.

Ultimately we want the best overall line breaks. But what does "best" really mean?

Knuth proposed a *loss function* $\texttt{loss}(w_1, \dots, w_k)$ that measures how bad a line of words $w_1, \dots, w_k$ would look[6]. The goal is to partition a sequence of words $w_1, \dots, w_n$ into consecutive subsequences $(w_1, \dots, w_{i_1-1}), (w_{i_1}, \dots, w_{i_2-1}), \dots, (w_{i_k}, \dots, w_n)$ to minimize the total loss,

$$\texttt{loss}(w_1, \dots, w_{i_1-1}) + \texttt{loss}(w_{i_1}, \dots, w_{i_2-1}) + \cdots + \texttt{loss}(w_{i_k}, \dots, w_n).$$

There are finitely many partitions, so the optimum is well-defined. Can we compute it efficiently?

First, the recursive spec. Fix a sequence of words $w_1, \dots, w_n$ and define $\texttt{BLB}(i)$ ("best line-breaking") as

$$\texttt{BLB}(i) = \text{minimum loss of any partition of } (w_i, \dots, w_n) \text{ into lines.}$$

We want $\texttt{BLB}(1)$. If $i > n$ there are no words to partition and the loss is 0:

$$\texttt{BLB}(i) = 0 \qquad \text{if } i > n. \tag{5.1}$$

[5] https://www.saildart.org/TEXDR.AFT%5B1,DEK%5D

[6] We treat `loss` as a black box and refer to [KP81] for implementation details.

Suppose $i \leq n$. Any partition of $(w_i, \ldots, w_n)$ has a first line $(w_i, \ldots, w_j)$. The rest is a partition of $(w_{j+1}, \ldots, w_n)$, which by induction has optimum loss $\texttt{BLB}(j+1)$. We don't know where to end the first line, so we try all choices:

$$\texttt{BLB}(i) = \min_{j=i,\ldots,n} \{\texttt{loss}(w_i, \ldots, w_j) + \texttt{BLB}(j+1)\} \qquad \text{if } i \leq n \tag{5.2}$$

The naive running time satisfies

$$T(0) = 1,$$
$$T(n) = \sum_{i=0}^{n-1} T(i) + O(n),$$

which is larger than 2^n. But there are only n distinct subproblems. With caching,

$$(\#\text{ subproblems}) \times (\text{time per subproblem}) = n \times O(n) = O\Big(n^2\Big).$$

And your notes look great.

5.5 Additional notes and references

The content in this chapter overlaps with [Eri19, chapter 3], [KT06, chapter 6], [DPV08, chapter 6], and [DD11, (video) lectures 19 – 21].

The line-breaking algorithm discussed here was published by Knuth and Plass [KP81]. TEX was later extended with a set of user-friendly macros by Leslie Lamport to form LATEX . They remain widely used in academia.

Spring 2026 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

5.6 Exercises

Many additional exercises with similar themes can be found in [Eri19, Chapter 3].[7]

Exercise 5.1. Each of the following problems takes as input an array $A[1..n]$ of integers and asks for an optimal subsequence.[8] Design and analyze an algorithm for each, addressing Items 1–5 from Section 5.2.[9]

1. A sequence of numbers $x_1, \ldots, x_k$ is (strictly) increasing if $x_i < x_{i+1}$ for $i = 1, \ldots, k-1$. Compute the length of the longest increasing subsequence of A.

2. A sequence of numbers $x_1, \ldots, x_k$ is *convex* if $x_{i+1} - x_i \geq x_i - x_{i-1}$ for $i = 2, \ldots, k-1$. Compute the length of the longest convex subsequence of A.

3. Compute the maximum sum of any increasing subsequence of A.

4. Compute both:

   - the length of the longest increasing subsequence of A where the sum of integers is even,
   - the length of the longest increasing subsequence where the sum of integers is odd.

[7] The exercises in [Eri19, Chapter 3] under the heading of "Trees and Subtrees" are better suited for Chapter 13 of these notes.

[8] A "subsequence" of $x_1, \ldots, x_n$ is a sequence of the form $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

[9] In particular, no proof is required. We will interpret your recursive spec as the inductive hypothesis of a proof by induction, and fill in the rest of the details ourselves. That said, some additional commentary may help the grader, especially in the event of partial credit.

5. Recall that the Fibonacci sequence

   $$1, 1, 2, 3, 5, 8, ...$$

   can be characterized as a sequence starting with 1, 1 and such that every successive number is the sum of the previous two numbers. ($1+1=2$, $1+2=3$, $2+3=5$, ...) Generalizing this notion, we say that a sequence $y_1, y_2, \ldots, y_k$ is *sub-Fibonacci* if $y_i \leq y_{i-2} + y_{i-1}$ for $i = 3, 4, \ldots, k$. That is, each successive number is at most the sum of the previous two.

   Compute the length of the longest increasing and sub-Fibonacci subsequence of A.

6. Suppose each entry in A is also colored red, white, or blue. We say that a sequence is *American* if the colors alternate red, white, blue, red, white, blue, ... The first number in the sequence can be any color. Compute the length of the longest increasing American subsequence of A. (You can assume that you can look up the color of $A[i]$, for any $i \in [n]$, in constant time.)

7. A sequence $x_1, \ldots, x_k$ is a *palindrome* if the reversed sequence is the same; i.e., $(x_1, \ldots, x_k) = (x_k, \ldots, x_1)$. For example,

   mom, *dad*, *racecar*, and *gohangasalamiimalasagnahog*

   are all palindromes. Compute the length of the longest palindrome subsequence of A.

8. Compute the length of the shortest supersequence of A that is a palindrome.[10]

9. Suppose each entry of A is colored red, white or blue. We say that a sequence is *un-American* if the colors in the sequence never cycle through the American dream; that is, no three consecutive entries have colors that form any of the following three sequences: (red, white, blue), (white, blue, red), or (blue, red, white). Compute the length of the longest increasing un-American subsequence of A. (You can assume that you can look up the color of $A[i]$, for any $i \in [n]$, in constant time.)

10. (Just for fun) Invent your own fun and interesting version of a dynamic programming problem over A.

[10] $x_1, \ldots, x_k$ is a *supersequence* of $y_1, \ldots, y_\ell$ if $y_1, \ldots, y_\ell$ is a subsequence of $x_1, \ldots, x_k$.

Exercise 5.2. Let $A[1..m]$ and $B[1..n]$ be two arrays. A common subsequence of A and B is a sequence that is a subsequence of both. Design and analyze an algorithm for each of the following problems, addressing Items 1–5 from Section 5.2.[16]

1. Compute the length of the longest common subsequence of A and B.
2. Compute the length of the shortest common supersequence of A and B.
3. Compute the length of the longest common subsequence of A and B that is a palindrome.
4. Suppose A and B consist of integers. Compute the length of the longest common increasing subsequence of A and B.
5. Compute the length of the shortest common palindrome supersequence of A and B.
6. Suppose A and B consist of integers. Compute the length of the longest common convex subsequence of A and B. (Cf. problem 2)

Exercise 5.3. Problem 7 of exercise 5.1 considered the problem of computing the longest palindrome subsequence. Here we recall that a palindrome is a string spelled the same forwards and backwards, such as "racecar". Here we consider the problem of computing the maximum weight palindrome subsequence where there characters of the input string are each given weights.

For example, consider the array of characters below where we annotate each character with its weight in the subscript.

$$\mathtt{x}_8\ \mathtt{r}_5\ \mathtt{y}_1\ \mathtt{a}_3\ \mathtt{c}_7\ \mathtt{z}_4\ \mathtt{e}_2\ \mathtt{c}_4\ \mathtt{x}_{-5}\ \mathtt{a}_5\ \mathtt{r}_{-1}$$

Above, `racecar` is a palindrome subsequence, with total weight $22 = 5 + 3 + 7 + 2 + 4 + 5 - 1$.

The input consists of a string $A[1..n]$ and an array $W[1..n]$ of real-valued weights. Each $W[i]$ gives the weight of character $A[i]$. The goal is to compute the maximum total weight of any palindrome subsequence of $A[1..n]$. Design and analyze an algorithm for this problem.

Exercise 5.4. Let $A[1..n]$ be an array of n integers and let $B[1..n]$ be an array of n characters. We say that a subsequence $i_1, i_2, ..., i_k$ of $1, \ldots, n$ is *increasing in* A if the corresponding subsequence of A, $A[i_1], \ldots, A[i_k]$, is (weakly) increasing (i.e.,

$A[i_i] \leq A[i_2] \leq \cdots \leq A[i_k]$). We say that a subsequence $i_1, i_2, \ldots, i_k$ of $1, \ldots, n$ is a *palindrome in* B if the corresponding subsequence of B, $B[i_1], B[i_2], \ldots, B[i_k]$, forms a palindrome.

Consider the problem of computing the length of the longest subsequence of $1, \ldots, n$ that is both increasing in A and a palindrome in B. Design and analyze an algorithm for this problem.

Exercise 5.5. Recall that a sequence of numbers $y_1, y_2, \ldots, y_k$ is *increasing* if $y_i < y_{i+1}$ for each i. Recall from MCQ that a common subsequence of two arrays A and B is another sequence that is a subsequence[11] of both A and B. Consider the problem of computing the length of the *longest common increasing subsequence* of two arrays of numbers, $A[1..m]$ and $B[1..n]$. For example,

$$(1, 4, 5, 6, 7, 9)$$

is the longest common increasing subsequence of the sequences

$$(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3) \text{ and } (1, 4, 1, 4, 2, 1, 3, 5, 6, 2, 3, 7, 3, 0, 9, 5).$$

Design and analyze an algorithm for this problem.

Exercise 5.6. Consider the edit distance problem from Section 5.1. In Section 5.1, insertions, deletions, and substitutions were all treated equal. Suppose instead that we had different costs $\alpha, \beta, \gamma > 0$ for each insertion, deletion, and substitution, respectively. Show how to modify the algorithm from Section 5.1 for these nonuniform costs. This model is useful in biological settings where α, β, γ reflect prior assumptions on the likelihood of each operation.

Exercise 5.7. Recall that different cost models are designed to reflect assumptions on how edits are naturally generated. For example, a reasonable hypothesis is that k consecutive deletions, or k consecutive insertions, are more likely than k independent single-character insertions or deletions. For example, in writing these notes, the author frequently finds himself deleting entire sentences[12] at a time. We can factor this propensity into our edit distance model by a "insert-or-delete-at-bulk" cost. Let

[11] As always, a subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings `C`, `DAMN`, `YAIOAI`, and `DYNAMICPROGRAMMING` are all subsequences of the string `DYNAMICPROGRAMMING`.

[12] or paragraphs, or more... sigh...

$f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a fixed nonnegative function. We introduce the operations of "bulk insertion" and "bulk deletion", where one deletes k characters or inserts a string of k characters, at a cost of $f(k)$, for any $k \in \mathbb{N}$. *We assume that f is monotonically increasing in k; that is, $f(k) < f(k+1)$ for all k.* We can then define the *bulk edit distance* of two strings x and y to be the minimum cost of any sequence of edit, bulk insertion, or bulk deletion operations.

Assume that f is provided as a subroutine that takes $O(1)$ time to evaluate. Design and analyze an algorithm for this problem.

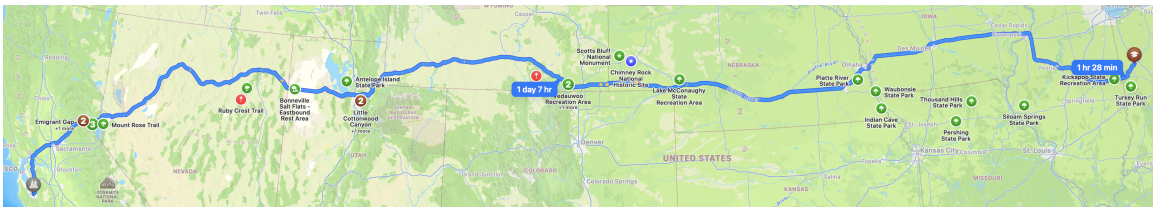
Exercise 5.8. Consider the line-breaking problem from Section 5.4. Suppose that you are given an additional input parameter k, which defines the maximum number of words allowed in a line. (You may assume $k \leq n$, and morally, $k \ll n$). Show how to modify the algorithm from Section 5.4 to give a faster running time (in terms of n and k).

Exercise 5.9. Suppose you are given a set of n points $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ in the plane.

1. Design and analyze an algorithm computing the maximum cardinality $|S|$ of any subset $S \subseteq P$ where every pair of points in S is comparable. (Cf. exercise 1.8 on page 35.)

2. Design and analyze an algorithm computing the maximum cardinality $|S|$ of any subset $S \subseteq P$ where every pair of points in S is incomparable.

Exercise 5.10. Imagine you are planning a long road trip to San Jose, CA and you've already fixed your route (through Illinois, Iowa, Nebraska, Wyoming, Utah, Nevada, and then California via Tahoe).



You don't mind the drive, but you hate having to stop to get gas. What a pain! And some stations take longer than others. The car can drive at most 500 miles before needing to stop to get gas. The high-level goal is to plan your gas stops to minimize

the amount of time spent pumping gas, while making sure you don't run out of gas between them.

We let n denote the total number of gas stations, and number the gas stations $1, \ldots, n$ in order along the route. The input consists of two arrays $D[1..n]$ and $T[1..n]$, and an additional value $Z > 0$. For each i, $D[i] \in \mathbb{R}_{\geq 0}$ is a nonnegative real number representing the distance along the route at which the gas station appears. (Again, the gas stations are ordered to be nondecreasing in $D[i]$.) $T[i]$ is a positive real number representing the amount of time it takes to pump gas at station i. Z represents the distance to the destination. You start with a full tank of gas.

Design and analyze an algorithm to compute the minimum amount of time one has to spend at gas stations in going from location 0 to location Z, while ensuring that you never drive more than 500 miles without visiting a gas station. (You should return $+\infty$ if it is impossible to do so.)

Exercise 5.11. In fair redistricting, the goal is to divide up a population into geographic regions so that all the local elections are as close as possible. This is like the opposite of gerrymandering; having close elections maximizes the influence of each constituent's vote.

For simplicity we consider the following 1-dimensional formulation of the problem. Let $A[1..n]$ be a sequence of n (positive or negative) integers. Each index $i \in [n]$ represents a neighborhood block in a sequence of n blocks. We have two political parties, the Apples and the Oranges. For each index i, $A[i] \in \mathbb{Z}$ represents the number of people in block i that have registered for the Apple party, minus the number of people in block i that have registered for the Orange party; i.e.,

$$A[i] = \begin{pmatrix} \text{\# Apples} \\ \text{in block } i \end{pmatrix} - \begin{pmatrix} \text{\# Oranges} \\ \text{in block } i \end{pmatrix}.$$

A *district* is defined as a consecutive interval $[i..j]$ of blocks, where $1 \leq i \leq j \leq n$. The *disparity* of a district $[i..j]$ is the absolute difference between the number of Apples and number of Oranges in the district:

$$(\text{disparity of } [i..j]) = \left| \begin{pmatrix} \text{\# Apples} \\ \text{in blocks } i..j \end{pmatrix} - \begin{pmatrix} \text{\# Oranges} \\ \text{in blocks } i..j \end{pmatrix} \right| = \left| \sum_{k=i}^{j} A[k] \right|.$$

Given $A[1..n]$ and $L \in [n]$, the goal is compute the minimum total disparity,

$$\sum_{k=1}^{L} (\text{disparity of } A[i_k..j_k])$$

over all partitions of the sequence blocks $[1..n]$ into L (nonempty) districts $[i_1..j_1], \ldots, [i_L..j_L]$. ("Partition" means that the districts are disjoint and cover all of the n blocks.) Design and analyze an algorithm for this problem.

Exercise 5.12. Word segmentation is the problem of dividing a string of written language into its component words. In English and many other languages using some form of the Latin alphabet, the space is a good approximation of a word divider (word delimiter), although this concept has limits because of the variability with which languages emically regard collocations and compounds. Many English compound nouns are variably written (for example, ice box = ice-box = icebox; pig sty = pig-sty = pigsty) with a corresponding variation in whether speakers think of them as noun phrases or single nouns; there are trends in how norms are set, such as that open compounds often tend eventually to solidify by widespread convention, but variation remains systemic. In contrast, German compound nouns show less orthographic variation, with solidification being a stronger norm. However, the equivalent to the word space character is not found in all written scripts, and without it word segmentation is a difficult problem.

Languages which do not have a trivial word segmentation process include Chinese, Japanese, where sentences but not words are delimited, Thai and Lao, where phrases and sentences but not words are delimited, and Vietnamese, where syllables but not words are delimited. In some writing systems however, such as the Ge'ez script used for Amharic and Tigrinya among other languages, words are explicitly delimited (at least historically) with a non-whitespace character.

The Unicode Consortium has published a Standard Annex on Text Segmentation,[1] exploring the issues of segmentation in multiscript texts.[13]

We model the word segmentation problem as follows. Let $A[1..n]$ be a sequence of letters from a fixed alphabet. A *word segmentation* is a partition of A into contiguous intervals $A[i_0 + 1..i_1], A[i_1 + 1..i_2], \ldots, A[i_\ell + 1..i_{\ell+1}]$; where $i_0 = 0$, $i_{\ell+1} = n$, and ℓ is any integer; such that each interval $A[i_j + 1..i_{j+1}]$ is a word. For example, the text

"youareawesome" has the word segmentation "you", "are", "awesome".

To determine which sequences form words, we assume access to a precomputed boolean array $\texttt{words}(1..n, 1..n)$, where for $i \leq j$, $\texttt{words}(i, j) = \texttt{true}$ iff $A[i..j]$ forms a word.

Given $A[1..n]$ and $\texttt{words}(1..n, 1..n)$, consider the problem of deciding if there is a word segmentation of A. Design and analyze an algorithm for this problem.

[13] https://en.m.wikipedia.org/wiki/Text_segmentation#Word_segmentation

Exercise 5.13. Subset Subs™, the world's first dynamically optimal deli, has the following delicious business model. Subset Subs makes only two submarine sandwiches each day:

- A ham sub m inches long, and
- A vegetarian sub n inches long,

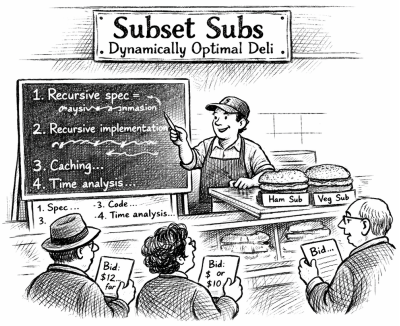
for integers m and n, depending on market conditions. There are also plans to expand to a vegan option, but not hoagies.[14]

Subset Subs sells its sandwiches fractionally and dynamically as follows. Each day, k customers submit bids on the sandwiches, each taking one of the following forms:

1. x dollars for y inches of the ham sandwich.
2. x dollars for y inches of the vegetarian sandwich.
3. A *disjunctive order*, specifying either x_1 dollars for y_1 inches of the ham sandwich, *or* x_2 dollars for y_2 inches of the vegetarian sandwich.

The disjunctive order means the customer would accept either of their listed options. The number of inches is always given as an integer.

There is usually too much demand to serve all of the bids. Each day, Subset Subs decides which of the bids to serve, with the objective of making the maximum amount of money. This means they choose a subset of ham orders, vegetarian orders, and disjunctive orders, choosing one of the two options for each disjunctive order they take, subject to having enough ham sandwich and enough vegetarian sandwich to serve all the orders they take. (We call this a "feasible" subset of orders.)



"Your sandwich will be ready in polynomial time."

In addition to the dollar amounts from the bids, Subset Subs also takes into account the ancillary costs when serving a piece of the sandwich, as a function of its length. This can reflect small things, like the wrapping paper, to the unusual logistics involved when someone orders an extremely large piece.

Ultimately, the input consists of the lengths m and n; k bids, each of one of the

[14] According the Guinness Book of World Records, the longest sandwich measured 735 m (2,411 ft 5 in) and was created by members of three teams in total. These teams were Groupe Notre Dame Hazmieh-Scouts de L'Independence (Lebanon), Municipality of Hazmieh (Lebanon)and Mini-B chain restaurants (Lebanon). The attempt took place in Hazmieh village, Beirut, Lebanon, on 22 May 2011.

The sandwich started at Notre Dame des Soeurs Antonines School and ended on Elie Street, in Hazmieh, Beirut, Lebanon. The width of the sandwich was 12.5cm and the overall estimated weight of the sandwich is 577.03kg.

three types above; and a lookup function $\texttt{cost} : \mathbb{Z}_{\geq 0} \to \mathbb{R}_{>0}$ that, given x, returns the ancillary cost of producing an x-inch long piece of sandwich, in $O(1)$ time. Design and analyze an algorithm to compute the maximum amount of profit that can be obtained by accepting a feasible set of orders. (As usual, the faster the better. Here we allow running times that are polynomial in m and n, even though they are specified in $\log(m)$ and $\log(n)$ bits, respectively.)[15]

Exercise 5.14. Who needs friends to play Set? The card game *Set* has 81 different cards varying in four features across three possibilities for each kind of feature. The four features are:

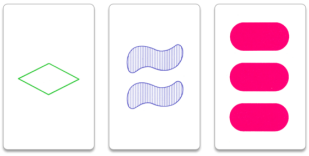
1. The shape, which is either a diamond, a squiggle, or an oval.
2. The number of shapes, which is either 1, 2, or 3.
3. The color of the shape, which is either red, green, purple.
4. The shading of the shape, which is either solid, striped, or open.

There are multiple variations of Set but they all are based on trying to form particular "sets" of three cards. A set consists of three cards satisfying *all* of these conditions:

1. They all have the same number or have three different numbers.
2. They all have the same shape or have three different shapes.
3. They all have the same shading or have three different shadings.
4. They all have the same color or have three different colors.

Loosely speaking, the rules of Set are summarized by: If you can sort a group of three cards into "two of _____ and one of _____", then it is not a set.

For example, the three cards on the right form a set. We have one card of each color, one of each shape, one of each shading, and one of each number. (In case this is printed in black and white, we mention that the first card is green, the second is purple, and the third is red.)



We will study a solitaire version of Set. Solitaire Set starts with 3 decks of cards. Each round you choose one of the following options.

1. If the three cards on top of the deck form a set, you may remove all three from the top of their decks, and set them aside.

2. You may discard one card from the top of one of the decks, revealing a new card underneath (unless that deck is empty).

[15] You might find it helpful to first solve the simpler case of just one type of sandwich and no "disjunction orders".

(Even if the top three cards form a set, you can choose to discard one of them instead.) The goal is to form as many sets as possible by the time you empty all the decks.

We will consider the game as an optimization problem where we also know the full sequence of cards in the deck. The three decks are represented by three arrays $A[1..m]$, $B[1..n]$, and $C[1..p]$.[16] The cards are listed from top to bottom in each deck. *For simplicity, we assume access to a constant-time subroutine* `IsItASet(i,j,k)` *that takes as input three indices* i, j, k *and outputs* `True` *if the cards* $(A[i], B[j], C[k])$ *form a set, and* `False` *if they do not.*

The problem is to compute the maximum number of sets that can be obtained in a game of Solitaire Set over the decks $A[1..m]$, $B[1..n]$, and $C[1..p]$. Design and analyze an algorithm for this problem.

Exercise 5.15. John McCarthy[17] invented Lisp in 1960. Lisp has an extraordinarily simple syntax: a Lisp program is a nested composition of *S-expressions*. An S-expression is either

1. an *atom*: a single token representing a name or value (such as `x`, `42`, or `nil`), or
2. a *list*: written in prefix notation as `(`f e_1 e_2 e_3 $\cdots$ e_k`)` where f is a token (the *operator*) and $e_1, \ldots, e_k$ are S-expressions (the *arguments*). In this problem we restrict to $k \in \{0, 1, 2\}$.[18]

This uniform representation, in which code and data are the same kind of object, makes Lisp *homoiconic*, and is the source of much of its expressive power.[19]

The parentheses make the structure of a Lisp program completely unambiguous. But suppose a Lisp program has been "flattened" and lost all of its parentheses, leaving only the flat sequence of tokens read left to right. For example, the S-expression

```
(cons (quote x) (quote y))
```

flattens to the sequence

```
cons quote x quote y
```

Can we recover a valid S-expression from such a sequence?

[16] The cards in these decks are not distinct; a particular card may have multiple copies. So, for example, there may be many more than 81 cards total.

[17] McCarthy's 1971 Turing Award lecture, "Generality in Artificial Intelligence," is available at https://dl.acm.org/doi/epdf/10.1145/33447.33448.

[18] For simplicity, our operators are restricted to single tokens, and not nested S-expressions.

[19] See "The Roots of Lisp", by Paul Graham, available at https://paulgraham.com/rootsoflisp.html

We model this as follows. Let $C[1..n]$ be an array of n tokens. We say that $C[1..n]$ is *feasible* if there exists a single S-expression e where the flattened sequence of e equals C. We say that C is *uniquely feasible* if the choice of e is unique.

Suppose you have access to a constant time function `type` that returns, for each token $C[i]$, the set of roles it can play:

$$\texttt{type}(C[i]) \subseteq \{\texttt{atom}, 0, 1, 2\}.$$

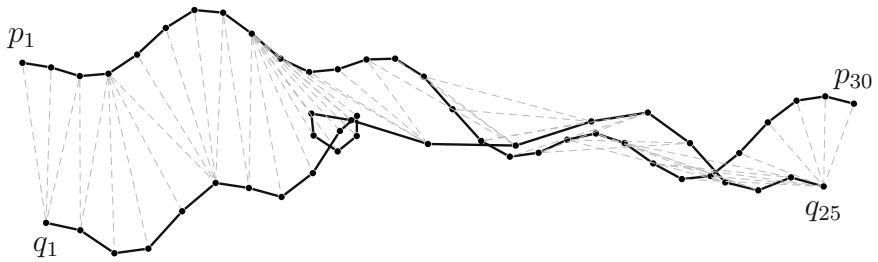
Here `atom` means the token can serve as an atom, and $k \in \{0, 1, 2\}$ means it can serve as an operator taking k arguments. A single token may have multiple roles; for example, a token like "`-`" might have one argument (`(- x)` negates x) or two (`(- x y)` subtracts y from x). A 0-argument operator represents a function that takes no arguments.

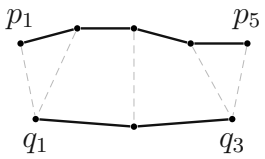
1. (9 points) Design and analyze an algorithm (the faster the better) that, given $C[1..n]$, decides whether C is feasible.

2. (1 point) Briefly describe how to modify your algorithm to decide whether C is *uniquely* feasible.

Exercise 5.16. Walking the dog. You take your dog out for a walk, on a leash. You walk along a path with stops $p_1, p_2, \ldots, p_m$, while your dog walks along its own path with stops $q_1, q_2, \ldots, q_n$.

Each "step", either you advance to your next point p_i, the dog advances to its next point q_j, or you both advance to p_i and q_j, respectively. (Neither you nor the dog can go backwards.) Each step, the leash needs to be at least as long as the straight line segment between you (at some point p_i) and the dog (at some point q_j). The high-level goal is to minimize the length of the leash, coordinating your and your dog's steps as you traverse the paths.





In the example above, the line segment between you and the dog at each step is drawn as a dashed line. The leash length would be the maximum length over all these dashed lines. Specifically, you take the first two steps, then the dog takes a step, then you take a step, and then the dog takes the next four steps, and so forth. A simpler example is given on the right.

Consider the problem of computing the minimum possible leash length over all possible ways of stepping through the two paths. The input consists of a table of distances $D[1..m, 1..n]$, where $D[i, j]$ encodes the distance between p_i and q_j. Design and analyze an algorithm for this problem.

Sample solution.

Problem. The following problem models the line breaking problem from Section 5.4. The input consists of an array $X[1..n]$ where each $X[i]$ represents a word. We are also given a "loss function" `loss(`i`,`j`)` that takes as input two indices $i \leq j$, and outputs a numerical value that represents the "badness" of $X[i], \ldots, X[j]$ as a line of text. For simplicity we assume that `loss(`i`,`j`)` takes constant time to evaluate.

The high-level goal is to partition X into contiguous subsequences

$$X[1..i_1], X[i_1 + 1..i_2], X[i_2 + 1..i_3], \ldots, X[i_{k-1} + 1..n]$$

minimizing the total loss:

$$\texttt{loss(1,}i_1\texttt{)} + \texttt{loss(}i_1 + 1\texttt{,}i_2\texttt{)} + \texttt{loss(}i_3 + 1\texttt{,}i_4\texttt{)} + \cdots + \texttt{loss(}i_{k-1} + 1\texttt{,}n\texttt{)}.$$

Here the number of contiguous subsequences (above, k) is arbitrary. The problem is to compute the minimum loss over all such partitions of X.

Solution.

1. *Recursive spec / induction hypothesis.* For $i = 1, \ldots, n + 1$, we define

   `min-loss(`i`)` = the minimum loss over all partitions of $X[i..n]$. (For $i > n$, $X[i..n]$ indicates the empty sequence.)

2. *Recursive implementation.*

   `min-loss(`i`)`

   1. If $i > n$ then return 0.
   2. Otherwise return the minimum, over all $j = i, i + 1, \ldots, n$, of

      $$\texttt{loss(}i\texttt{,}j\texttt{)} + \texttt{min-loss(}j + 1\texttt{)}.$$

3. *Solving the original problem.* The solution to the problem is given by `min-loss(1)`.

4. *Mention "dynamic programming" or "caching".* We apply dynamic programming to the recursive algorithm and cache the solutions to the subproblems.

5. *Running time.* For $i = 1, \dots, n$, `loss(`i`)` has a loop with at most n iterations, and each iteration takes constant time. So each subproblem takes $O(n)$ time. Over all n subproblems, the algorithm takes $O(n^2)$ time in total.

Sample solution.

Problem. (Exercise 5.1 problem 1) Let $A[1..n]$ be an array of integers. A sequence of numbers $x_1, \dots, x_k$ is (strictly) increasing if $x_i < x_{i+1}$ for $i = 1, \dots, k-1$. Compute the length of the longest increasing subsequence of A.

Solution.

1. Recursive spec / induction hypothesis. For $i \in \{1, \dots, n\}$, we define

> `LIS(`i`)` = the length of the longest increasing subsequence of $A[i..n]$ where the subsequence starts with $A[i]$.

(Here `LIS` stands for longest increasing subsequence.)

2. Recursive implementation.

> `LIS(`i`)`
>
> `1.` If $A[i] \geq A[j]$ for all $j > i$ then return 1.
>
> `2.` Otherwise return the maximum of
>
> $$1 + \texttt{LIS(}j\texttt{)}$$
>
> over all indices $j > i$ with $A[j] > A[i]$.

3. Solving the original problem. The maximum of `LIS(`i`)` over all $i \in [n]$ gives the length of the longest increasing subsequence in A.

4. Running time with caching / dynamic programming. With dynamic programming, the running time is $O(n^2)$, where we have n subproblems each of which take $O(n)$ time (to iterate over $j > i$, in steps `(1)` and `(2)`).

Remark 5.1. We emphasize that the recursive spec essentially solved the entire problem.

Remark 5.2. It is well known that there are faster algorithms but that is besides the point.

Chapter 6

Searching and sorting graphs

Recall that graphs consist of vertices (denoted V) and edges (denoted E) that connect pairs of vertices. This chapter is exclusively about *directed graphs*, where edges are directed and represented by ordered pairs (u, v). Here u and v are vertices, and the edge (u, v) is said to be directed from u to v. Directed edges are also known as arcs. For a directed edge/arc $a = (u, v)$, u is called the *tail* or the *source*, and v is called the *head* or the *target*, of a. A directed graph is allowed to have edges in both directions (u, v) and (v, u) between two pairs.

A *walk* is a sequence of arcs $e_1, \ldots, e_k$ where the head/target of one arc e_i is the tail/source of the next arc e_{i+1}, for $i \in \{1, \ldots, k-1\}$. The source of e_1 is called the starting point, and the target of e_k is called the end point or terminating point of the walk. For two vertices u and v, a *walk from u to v* is a walk starting at u and ending at v. We say that u can *reach* a vertex v if there is a walk from u to v.

Seemingly anything can be modeled as a graph. We give a few examples where the modeling is fairly straightforward. The world wide web is a directed graph. Every website is a node. Every link from one website to another is a directed edge from the first website to the second. Modern search engines are based on analyzing this graph.

Another example comes from programming. Modern programming languages allocate most of their data from a heap, where small blocks of memory are encapsulated by objects. These objects refer to each other by their locations in virtual memory. Here we have a graph where the objects are nodes and the pointers across objects are directed edges. In this model, *garbage collection* can be understood as a graph theoretic problem where the goal is to identify all the objects that cannot be reached by any sequence of pointers starting from a pointer on the stack, so that we can recycle the underlying memory.

Lastly, consider any road map. We can think of each intersection as a node, and the streets between the intersections as edges. Most streets will be undirected (or equivalently bidirected) edges since they allow traffic in both directions. One-way

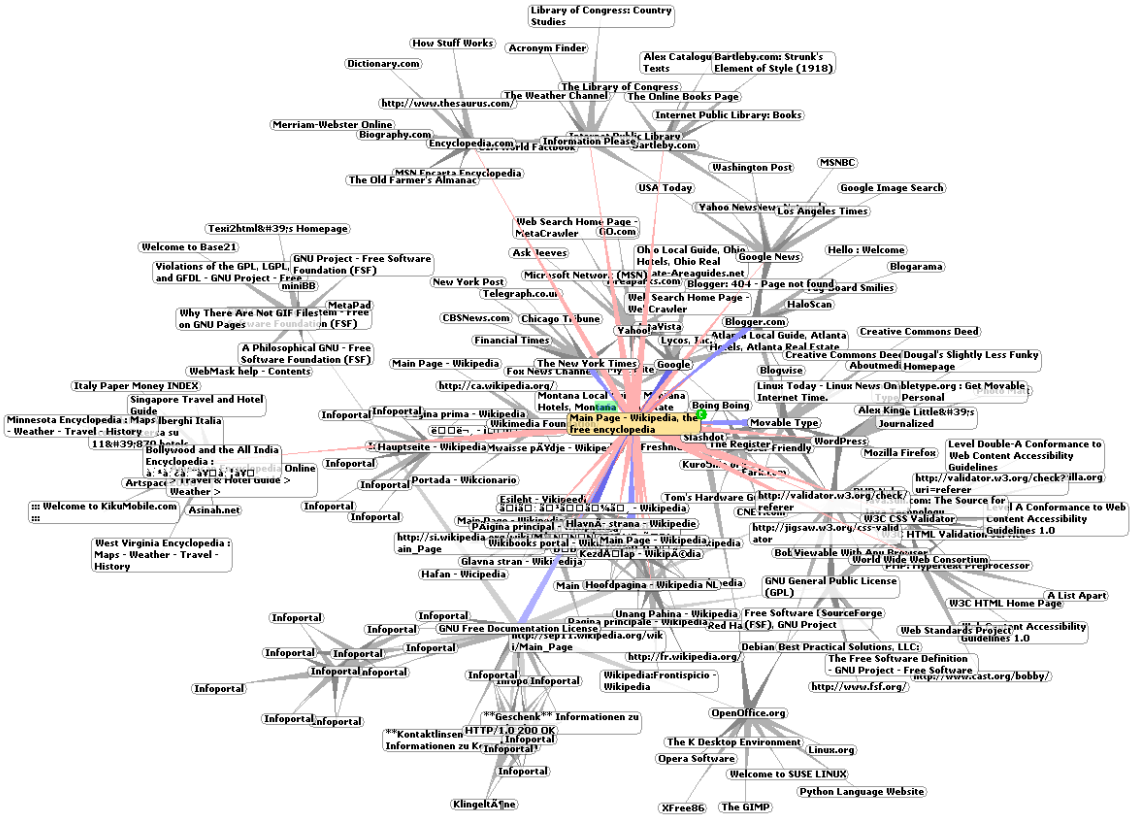


Figure 6.1: Map of a small part of the world wide web around `wikipedia.org` [CW].

streets are modeled by a directed edge in the direction of traffic. We can also annotate the edges with the speed limit and the length of the road, which gives us enough information to estimate how long it takes to go traverse that segment of the road.

We usually assume that graphs are given as a list of (possibly weighted) edges. With this list, one can build different data structures to make certain operations more convenient. Here we list the two most common ones. The first is to maintain, for each vertex $v \in V$, the list of all edges leaving v and (optionally) the list of all edges entering v. This is called the *adjacency list* representation. The second is to encode the edges in a large $V \times V$ matrix A (i.e., a two-dimensional array indexed by vertices V), as follows[1]. For every edge (u, v), we set $A[u, v] = 1$. Alternatively, if the edges

[1]Here, to use the array, we assume that the vertices are numbered from 1 to n. If not, then we can preprocess the input and reindex the vertices from 1 to n. Alternatively, instead of an array, one can use a hashmap to store the values. We will discuss hashmaps in greater detail later, but morally,

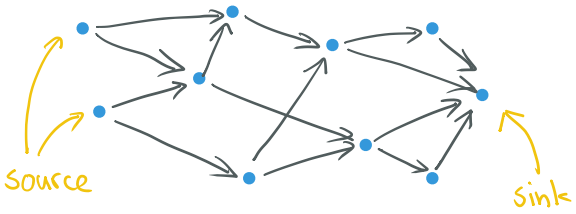
are weighted, then we set $A[u, v]$ to be the weight of the edge. The other coordinates corresponding to non-edges are left as 0.

There is an important functional difference between the adjacency matrix and the adjacency map. The adjacency list allows us to query, for each vertex v, the list of neighbors of v in $O(1)$ time per neighbor. The adjacency matrix does not, as one has to loop through the array to find each successive edge. On the other hand, the adjacency matrix allows us to query, for every pair of vertices $u, v \in V$, whether there is an edge from u to v, in $O(1)$ time. The adjacency list does not, since one has to scan the list of u's neighbors to see if v is there.

Can one get the best of both worlds? Easily: just have both data structures on hand. One can also consider extensions such as using data structures (like search trees or hash maps) for each list of neighbors in the adjacency list data structure. We may discuss more advanced graph data structures later.

6.1 Sorting directed acyclic graphs[2]

A *cycle* is a walk that starts and ends at the same vertex. We start with an important special case of directed graphs where there are no directed cycles called *directed acyclic graphs*, and abbreviated *DAGs*.



We say that a vertex $v \in V$ is a *source* if there are no edges directed into v, and a *sink* if there are no edges directed out of v. Every DAG has at least one source and at least one sink. (Why?)

Now, let $G = (V, E)$ be a DAG. Let v_1 be any source vertex in G. Observe that the graph G_1 obtained by removing v_1 from G (and all incident edges) is again a DAG. In particular, G_1 has at least one source vertex v_2 (which was either a source in G, or had only one incoming, which was from v_1). We can remove v_2 from G_1 to obtain a new DAG G_2, and continue in the fashion, repeatedly removing vertices $v_3, v_4, v_5, \ldots$

this is like randomly reindexing the vertices from 1 to $O(n)$.

[2]This section overlaps with Section 9.3, which was about circuits, an important example of directed acyclic graphs. We repeat some parts about DAGS in general because we ran out of time to fully discuss circuits in lecture.

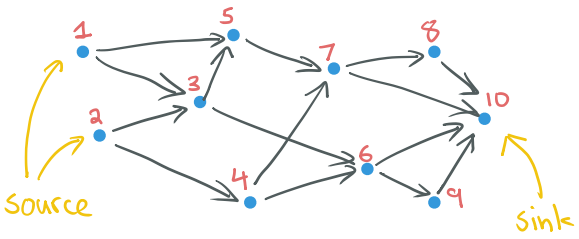
where for each i, v_i is a source in the graph G_{i-1} obtained by removing $v_1 \dots, v_{i-1}$ from G. The following subroutine encodes this process.

`source-peeling`$(G = (V, E))$

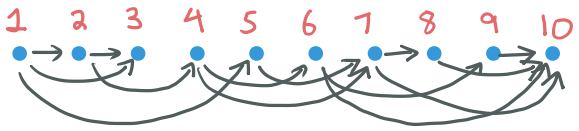
/ We assume G is a DAG. */*

1. While there is a source $v \in V$
   A. Append v to the output list.
   B. Remove v and all incident edges from G.

Below we number the vertices of a DAG as they are peeled.



Consider the vertices $v_1, \dots, v_n$ in the order they are iteratively peeled from the graph. Let v_i and v_j be two vertices where v_i was peeled before v_j (i.e., $i < j$). When v_i was peeled, it was a source in a subgraph that included v_j. In particular, we cannot have a directed edge (v_j, v_i) from v_j to v_i. Thus all edges go from v_i to v_j for $i < j$.



The list output by the source-peeling algorithm is an example of a *topological order*. In general, an ordering $v_1, \dots, v_n$ of V is a *topological order* iff every directed edge in G goes from v_i to v_j for $i < j$. Equivalently, $v_1, \dots, v_n$ is a topological order if $i < j$ whenever v_i can reach v_j.

Topological orderings are extremely useful algorithmically. They give us a *sequential structure* of the graph on which we can apply the dynamic programming techniques of Chapter 5 (as we will see later). For these applications it is worth pointing out that `source-peeling` can also be made to run in linear time. Here one spends linear time to first calculate the in-degree[3] of each vertex. The sources are the

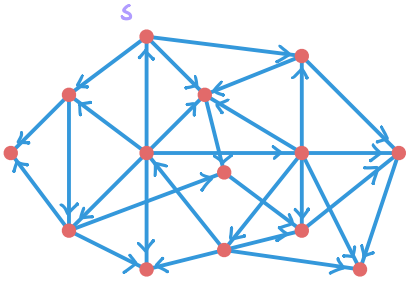
[3]The (unweighted) in-degree of a vertex v is the number of edges directed into v.

vertices with in-degree 0. Whenever we peel a source v, for each arc (v, w) leaving v, we also decrease the in-degree of w as we delete that edge. We keep track of the vertices whose in-degrees are 0 so that the next source can be identified instantly.

Theorem 6.1. *Given a DAG $G = (V, E)$ with m edges and n vertices, one can compute a topological ordering of V in $O(m + n)$ time.*

6.2 Searching graphs

We consider the elementary problem of connectivity. Let $G = (V, E)$ be a directed graph with m edges and n vertices, and fix $s \in V$. We want to identify the set of vertices that s can reach. For example, consider the following directed graph. Can s reach all of the vertices in this graph?



Identifying the set of reachable vertices is a very practical computational task. Of course there is the literal interpretation of physical reachibility on a map. Garbage collection, mentioned earlier, is also a matter of reachability. In planning problems, we interpret vertices as some kind of state, and edges as actions that take an entity from one state to another. Then reachability is asking whether it is possible to get from one initial state to another, desired state.

Let us first focus on the (s, t)-reachability problem. Here we have two vertices s and t and simply want to know if s can reach t. Consider the following natural recursive algorithm.

`recursive-search(`s`,`t`)`

`/*` *Spec: Returns* `true` *if s can reach t, and* `false` *otherwise.* `*/`

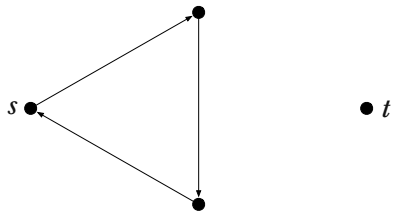
1. If $s = t$ then return `true`

`/*` *See if any outgoing edge from s leads to t.* `*/`

2. For each edge $(s, u) \in E$
   A. If `recursive-search(`u`,`t`)` returns true then return `true`
3. Return `false`

The logic is clear. We are searching for t and are currently at s. If we are already on t then of course we are done. Otherwise, any path to t from s must start with an outgoing edge from s. So we try each of them and see if any of them will lead to s with a recursive call. If any succeed, then we return `true`. Otherwise, we return `false`.

There is a catch, however; more precisely, an *algorithmic* flaw in our argument. Consider, for example, the following graph.



The recursive search algorithm will spin around the triangle indefinitely. In general, any cycle in the graph will send `recursive-search` in a loop.

In general, when using induction to justify a recursion, we require some kind of ordering to avoid these "bad" cyclic recursive dependencies. That is why dynamic programming was so effective on sequences, where there are clear notions of "left and right" and we naturally steered clear of cyclic dependencies.

A helpful analogy is mazes. A maze is essentially an (s, t)-reachability problem. The entrance of the maze is s and the exit is t. When imagining a maze, though, keep in mind that the algorithm's perspective is not the convenient birds-eye view that we see in puzzle books. Rather, we should think of ourselves as being *in* the maze, with tall walls to either side, and not having a clear sense of absolute location or direction. Imagine running through a maze, revisiting the same intersection again and again, feeling lost. That's how `recursive-search` feels when it falls into a cycle.

6.2.1 Depth-first search

To solve mazes, let us take inspiration from folktales: breadcrumbs. Imagine being in a maze, or a dense forest. If we left breadcrumbs along our trail, we could keep track of where we've already been, and avoid a lot of wasted effort. Algorithmically, this is embodied by the simple idea of *marking*: whenever we visit a vertex v we mark it in memory. If it was already marked, then we stop exploring v and return (to wherever we were before visiting v). Otherwise we continue to explore v's edges. The following algorithm, called `depth-first-search` (and abbreviated `DFS`), augments `recursive-search` with marking.

`DFS(`v`)`

1. if v is unmarked
   A. mark v
   B. for each edge (v, w) from v
      1. `DFS(`w`)`

The algorithm is pretty simple. The following observation is obvious but it is helpful to make it explicit.

Observation 6.2. *If vertex v is marked, and $(v, w) \in E$, then w is (eventually) marked.*

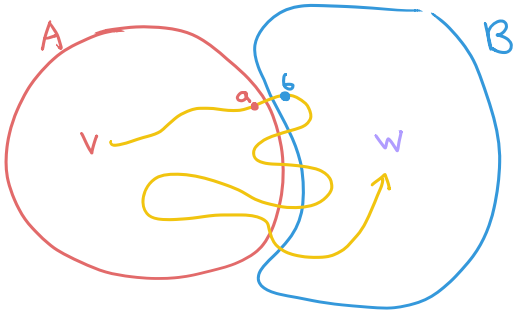
This sets up the main structural invariant of depth-first search.

Lemma 6.3. *When* `DFS(`v`)` *returns, all vertices reachable from v have been marked.*

Proof. Consider the moment that `DFS(`v`)` returns. Let

$$A = \{\text{marked vertices}\} \text{ and } B = \{\text{unmarked vertices}\}.$$

Let w be reachable from v, and suppose by contradiction that $w \in B$. Since w is reachable from v, there is a walk p from v to w. Since that path starts in A and ends in B, there must be a directed edge $(a, b) \in p$ such that $a \in A$ and $b \in B$.



But having a marked, b unmarked, and $(a, b) \in E$ contradicts Observation 6.2. □

Remark 6.4. Lemma 6.3 applies to `recursive-search` as well, except for that `recursive-search` may never terminate.

Now we analyze the running time.

Lemma 6.5. `DFS(`v`)` *runs in $O(m + n)$ time.*

Proof. We only traverse an edge $e = (v, w)$ after marking the initial point v. Each vertex gets marked once, so each is traversed at most once. □

The two lemma's combine above give the following theorem about DFS.

Theorem 6.6. *In $O(m+n)$ time,* DFS(v) *marks all the vertices reachable from v.*

So we can identify all vertices reachable from a vertex v in linear time.

6.3 Post-DFS Order

Having seen that DFS(v) will search all vertices reachable from v, it is also interesting to study the *order* in which DFS searches the vertices.

In particular, consider the order in which DFS marks and finishes calls to DFS(v). To make our discussion more concerete, let us modify DFS to prints out vertices as we finish searching them.

DFS(v)

1. If v is unmarked:
   A. Mark v.
   B. For each edge (v, w) from v:
      1. DFS(w).
   /* *The following line is the only difference from the first version of* DFS. */
   C. Append v to the output

In case one call to DFS(v) does not search and print out all the vertices in the graph, we can wrap it in a simple driver that loops DFS over all v.

DFS-driver($G = (V, E)$)

1. For each vertex $v \in V$
   A. DFS(v)

Ultimately DFS-driver prints out the vertices in some order $v_1, \ldots, v_n$. We call any ordering $v_1, \ldots, v_n$ of V produced by DFS-driver a *post-DFS ordering* of V. ("post" because each v is printed at the end of DFS(v).) To understand this ordering, we first recall the basic invariant of DFS: DFS(v) returns only when all vertices reachable from v have been marked. A slightly stronger observation is as follows.

Observation 6.7. *Consider a call to* DFS(v) *for a vertex v. In the time between the beginning and end of the call,* DFS *only searches vertices that are reachable from v.*

This sets up the following lemma.

Lemma 6.8. *Let v, w be two vertices in a graph such that v can reach w but w cannot reach v. Then any post-DFS ordering lists w before v.*

Proof. We have two cases, depending on whether DFS marks v or w first. Suppose DFS visits v first. By Lemma 6.3, DFS does not return from the recursive call `DFS(`v`)` until all vertices reachable from v have been marked. In particular we call and complete `DFS(`w`)` before the call to `DFS(`v`)` completes.

In the second case, suppose w is marked first. Then the depth-first search will only explore vertices that w can reach before completing the call to `DFS(`w`)` - and in particular, it will not search v before `DFS(`w`)` terminates. □

Now, suppose G is a DAG, and u can reach v. Then v cannot reach u because otherwise we would have a cycle. So v is always listed before u in the post-DFS order. If we apply this observation to every edge $(u, v) \in E$, we see that the edges are always directed up the list in the post-DFS ordering. That is, the post-DFS ordering gives a *reverse* topological ordering.

Lemma 6.9. *In $O(m + n)$, `DFS-driver(`$G = (V, E)$`)` outputs a list of the vertices V in reverse topological order.*

So DFS (plus reversing the list) gives a second way to topologically sort a directed acyclic graph.

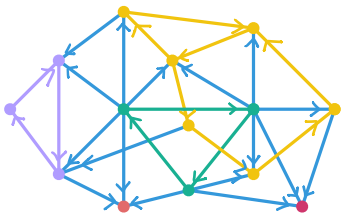
6.4 Strongly-connected components

Two vertices $s, t \in V$ are *strongly connected* if s can reach t and t can reach s. A graph is strongly connected if every pair of vertices is strongly connected.

Lemma 6.10. *Strong connectivity defines an equivalence relationship. (In particular, if u and v are strongly connected, and v and w are strongly connected, then u and w are strongly connected.)*



We only sketch the proof as it is straight forward. Suppose u and v are strongly connected, and v and w are strongly connected. A path from u to v and a path from v to w can be concatenated to give a walk from u to v. Similarly we obtain a walk from w to u. So u and w are strongly connected.



Each maximal set of strongly connected vertices is called a *strongly connected component*. The strongly connected components partition V. On the right, we have colored the vertices of a graph by their strongly connected component.

We now consider the problem of computing all of the connected components in a directed graph G. It is easy to see that there is a polynomial time algorithm. For example, we can run `DFS(`v`)` from every vertex v, fill out a two-dimensional table with the reachability results, and identify all the strongly connected pairs. (This should take $O(mn)$ time, assuming $m \geq n$.) Can we do better?

Let us consider the simpler problem of identifying the strongly connected component of a particular vertex v. We can run `DFS(`v`)` to identify all the vertices that v can reach. What is the fastest way to identify all the vertices that can reach v?

We first mention a few reasonable ideas that have been suggested in lectures.

1. Run `DFS(`u`)` from every vertex u visited by `DFS(`v`)`, and take the subset of vertices u that could reach v. This will work, but the $O(n(m+n))$ running time is not very impressive.

2. Run `DFS(`u`)` only from the leaves u of the search tree carved out by `DFS(`v`)`. The intuition is that if u is a leaf of the search tree, and u can reach v, then all other vertices from v to u along the (u, v)-path in the search tree can also reach v. Therefore we do not need to test these intermediate vertices. We can also omit the vertices not visited by `DFS(`v`)` since we already know that they are not strongly connected to v.

   Again, this works, and while there may be fewer leaves than vertices overall, it is not true that in the worst case there are substantially fewer than n of them. So the worst case running time is still $O(n(m+n))$.

3. Another suggestion is to somehow adjust the `DFS(`v`)` routine to try to detect back-edges of the search tree and, somehow, detect the strongly connected components on the fly. I'm not sure if this can be made to work; certainly, it seems hard to prove it works.

Here is a helpful tip for this and many more problems: *don't change the algorithm; change the graph.* Consider the graph $G_r = (V, E_r)$ that has the same vertex set as G but reverses the directions of all the edges. It takes only linear time to build G_r. Then

a can reach b in G iff b can reach a in G_r. `DFS(`v`)` in G_r will identify all the vertices that can reach v in G in $O(m+n)$ time. The strongly connected components of v is then the set of all vertices marked by `DFS(`v`)` in both G and G_r. This algorithm takes only $O(m+n)$ time.

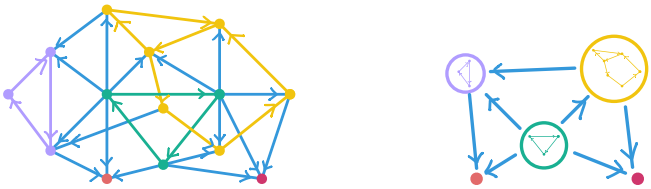
Thus we can find a single strongly connected component in $O(m+n)$ time. Let's move on to the greater task of identifying *all* the strongly connected components. A natural idea is to repeatedly use our clever $O(m+n)$ algorithm to identify and remove one strongly connected component at a time.

`SCC-peeling(`$G=(V,E)$`)`

1. Until $V = \emptyset$
   A. Let v be any vertex in V
   B. Compute the strongly connected component C of v and add C to the output.
   C. Remove C and all incident edges from G

Consider the running time of `SCC-peeling`. Potentially, we might remove only one vertex at a time. This gives a $O(n(m+n))$ running time which is hardly impressive. We press on.

Taking a step back, consider the graph G_C obtained by contracting each strongly connected component to a single vertex.



There is something conspicuous about this contracted graph: there are no directed cycles. This follows from maximality of the connected components. (If there was a cycle among the components in the contracted graphs, then all of these components would have to be strongly connected, a contradiction.) So the strongly connected components form a DAG. Consequently we can also speak about topological orderings of the DAG of SCC's, as well as sink SCC's and and source SCC's.

Now, consider the peeling algorithm above, with the one modification that we always select a vertex from a sink component. Note that we don't know how to easily identify a vertex from a sink component quickly, but we will assume it for the moment and explore the consequences first.

`sink-first-SCC`$(G = (V, E))$

1. Until $V = \emptyset$

   `/*` *We currently don't know how to identify the following vertex v.* `*/`

   A. let v be any vertex in a *sink component* of G

   `/*` `dfs(`v`)` *will search exactly the sink component of v.* `*/`

   B. $C \leftarrow$ all vertices marked by `dfs(`v`)`.

   C. Append C to the output.

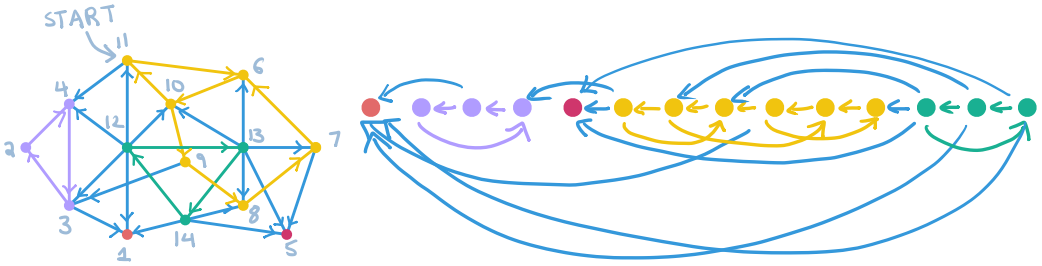
   D. Remove C and all incident edges from G

Observe that if v is always selected from the sink component of the remaining graph, then `dfs(`v`)` never leaves the sink component (by definition of sink). Thus, over the course of processing all the sink components, we only search each vertex and traverse each edge once. This gives a total running time of $O(m + n)$, a great improvement from $O(n(m + n))$. Of course, the algorithm is incomplete, because we won't know how to find vertices in a "sink-first" fashion. Thus the question remains: can we find a "sink-first" ordering of components?

Recall that depth-first search gives a (reversed) topological orderings in a DAG via the post-DFS order. More generally we proved (in Lemma 6.8) that the post-DFS ordering has the following property in any directed graph:

If u can reach v but v cannot reach u, then v is listed before u. $(\star)$

This property does not tell us about the order between strongly connected pairs u and v. But suppose u and v are *not* in the same strongly connected component, and suppose there was a directed edge from u to v. Then u can reach v, and v cannot reach u, so v will be listed before u, and the edge (u, v) will be directed up the post-DFS order (so to speak). That is, if we list the vertices in post-DFS order from left to right, all of the edges that go *across* SCC's will be directed from right to left.



One might be tempted to declare that the first vertex must be from a sink component. (Which is suggested by the picture above.) Actually this is not quite true. But we can say something about the *last vertex* instead.

Lemma 6.11. *The* last *vertex in the post-DFS ordering is a vertex from a source component. More generally, given any ordering of the vertices satisfying* ($\star$) *above, the last vertex is a vertex from a source component.*

Proof. Let $v_1, \ldots, v_n$ satisfy ($\star$); the claim is that v_n is in a source component. If not, then there exists some vertex v_i, where $i < n$, that can reach v_n, but v_n cannot reach v_i. But this contradicts ($\star$). □

Recall that we want vertices from sink components of G. We now have access to vertices from the source component. How do we get vertices from the sink component instead?

Don't change the algorithm; change the graph! Consider the reverse graph G_r. Sink component in G are source components in G_r and vice-versa.

Thus, finally, we can extract sinks components in the following fashion. We compute a post-DFS ordering $v_1, \ldots, v_n$ in the reversed graph G_r; we will fix that ordering for the rest of the algorithm. We take the last vertex v in this ordering, which belongs to a sink component in G. We find the connected component of v and remove it from G and G_r. Observe that the remaining ordering still satisfies ($\star$). The last (remaining) vertex v belongs to a sink component in the remaining subgraph of G. Continuing in this fashion we keep extracting sink components from G and source components from G_r. The (corrected) pseudocode is as follows.

`sink-first-SCC(`$G = (V, E)$`)`

1. Compute a post-DFS ordering of V in the *reversed* graph G_r
2. Until $V = \emptyset$
   - A. Let v be the *last* (remaining) vertex in V in the post-DFS order of G_r
   - `//` *v is in a sink component of G.*
   - B. $C \leftarrow$ all vertices marked by `dfs(`v`)`
   - C. Append C to the output
   - D. Remove C and all incident edges from G

Thus we have shown the following.

Theorem 6.12. *Let $G = (V, E)$ be a directed graph with m edges and n vertices. In $O(m + n)$ time,* `sink-first-SCC(G)` *computes the strongly connected components of G. Moreover, it outputs a list of the strongly connected components in reverse topological order with respect to the DAG obtained by contracting the strongly connected components.*

The result is due to Tarjan [Tar72].

6.5 Additional notes and materials

This chapter overlaps with [Eri19, Chapters 5 and 6], [KT06, §3.1–3.3, 3.5–3.6], [DPV08, Chapter 3], [CLR+09, Chapter 22], and [DD11, (video) lecture 14].

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

6.6 Exercises

Many more exercises related to this chapter can be found in Chapter 5 and Chapter 6 in Jeff's notes.

Exercise 6.1. Prove that every directed acyclic graph has at least one source, and at least one sink.

Exercise 6.2. Prove or disprove: every walk in a directed acyclic graph has finite length.

Exercise 6.3. Consider the following recursive algorithm for exploring a directed graph with n vertices and m edges.

`edge-search(v)`

1. for each edge $e = (v \to w)$ leaving v
   A. if e is unmarked
      1. mark e
      2. `edge-search(w)`

Analyze `edge-search` and give an upper bound (as tight as possible) on the running time, as a function of m and n.

Exercise 6.4. Let $G = (V, E)$ be a directed graph. Design and analyze an algorithm for each of the following problems.

1. Decide if there is a vertex x such that x can reach all other vertices.
2. Decide if there is a vertex x such that x can be reached by all other vertices.
3. Decide if there is a vertex x such that every vertex can either reach x or be reached from x.

Exercise 6.5. Let $G = (V, E)$ be a directed graph. We say two vertices $s, t \in V$ are *half-connected* if either s can reach t or t can reach s. We say that the graph G is *half-connected* if every pair of vertices is half-connected. We say that two vertices s and t are *strictly half-connected* if either s can reach t, or t can reach s, but not both. We say that G is strictly half-connected if every pair of vertices is strictly half-connected.

1. Consider the problem of deciding if G is half-connected. Design and analyze an algorithm for this problem.
2. Consider the problem of deciding if G is strictly half-connected. Design and analyze an algorithm for this problem.

Exercise 6.6. Let $G = (V, E)$ be a directed graph, and consider the problem of adding edges to G to make G strongly connected. The goal is to make G strongly connected by adding the minimum number of edges. Design and analyze an algorithm for this problem.

Exercise 6.7. Let $G = (V, E)$ be a directed graph with m edges and n vertices, such that there is at least one directed path between every pair of vertices (in at least one direction or the other). Suppose each vertex $v \in V$ is labeled by a distinct integer $\pi(v) \in [n]$ (that is, $\pi : V \to [n]$ is injective). We say a pair of vertices (u, v) is *reachably-misordered* if u can reach v and $\pi(v) < \pi(u)$.

Design and analyze an algorithm (as fast as possible) that counts the number of reachably-misordered pairs of vertices.

Exercise 6.8. Most modern object-oriented programming languages use *garbage collection* to manage memory. In this model, we have a family of objects, each of which have pointers to other objects. We also have "external pointers" (on the stack) which point to some of the objects. At a given point in time, a computer program has direct access to objects via the external pointers on the stack, and then access additional objects by chasing/following pointers from one object to the next.

Clearly, when an object has no pointers to it (either external pointers, or pointers from other objects), then it is no longer accessible from the rest of the program, and it is safe to "garbage collect" the object and free up the underlying memory for the rest of the program to use. But more generally, any object that cannot be accessed by following pointers from object to object, starting from an external pointer, can be garbage collected since it cannot be accessed from the stack.

Garbage collection can have a cascading effect. When you garbage collect an object x, and delete its pointers, there may be more objects that are freed up. The goal is to identify these objects.

We can think of the object system as a directed graph $G = (V, E)$, where each vertex $v \in V$ is an object, and each arc (u, v) models a pointer from object u to object v. We model external pointers with a set $X \subseteq V$; each $x \in X$ indicates an external pointer to object x. "Chasing / following pointers" corresponds to a walk in this "object graph", and an object is garbage collected when it is no longer reachable from any $x \in X$.

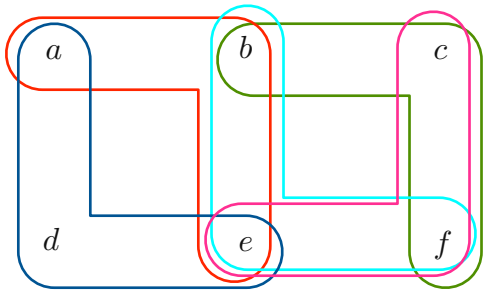
For simplicity, you can assume that there are no duplicate pointers: there is at most one pointer (u, v) from u to v for any pair $u, v \in V$; and at most one external pointer to each object. That is, both X and E are sets. (We can assume this without loss of generality, as one can always preprocess the input to enforce this.)

Suppose you have an object system described by the graph $G = (V, E)$ and the set of external pointers by X. We let $n = |V|$, $m = |E|$, and $\ell = |X|$. Given an external pointer $x \in X$, design and analyze an algorithm that identify the set of all objects that would be garbage collected upon removing x from the system. (To be clear, the input consists of $G = (V, E)$, X, and x.)

Exercise 6.9. Recall that a directed graph $G = (V, E)$ consists of a set of vertices V and edges E that are ordered pairs of vertices (u, v) (where $u, v \in V$). A walk in G is a sequence $v_1, v_2, \ldots, v_k$ where each consecutive pair, (v_i, v_{i+1}), is an edge.

Generalizing this idea, we define a *rank-3 directed hypergraph* $G = (V, E)$ as consisting of a set of vertices v, and "directed hyperedges" E which consist of *ordered* triples (a, b, c) where $a, b, c \in V$. (a, b, c don't have to be distinct.) A *feasible walk* in the hypergraph G is a sequence of vertices $v_1, v_2, \ldots, v_k$ with at least 3 vertices ($k \geq 3$) and such that each consecutive triple, (v_i, v_{i+1}, v_{i+2}) for $i = 1, \ldots, k-2$, is a directed hyperedge.

Below is a picture of a directed hypergraph with 5 directed hyperedges, listed on the right. (Note that the drawing does not convey the order of vertices within each edge; this should be inferred from the list of directed hyperedges.)



(a, b, e)
(a, d, e)
(b, e, f)
(b, c, f)
(e, f, c)

The walk

$$a \to b \to e \to f$$

is a feasible (a, f)-walk because each consecutive subsequence of 3 vertices – (a, b, e) and (e, b, f) – are directed hyperedges. The (a, f)-walk

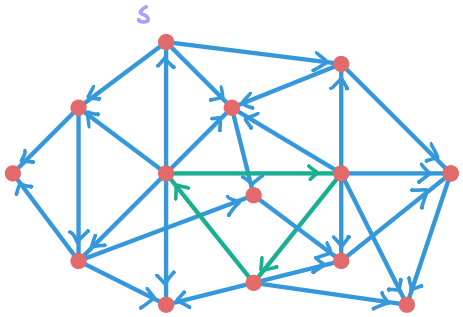
$$a \to d \to e \to f$$

is *not* a feasible walk because (d, e, f) is *not* a directed hyperedge. Meanwhile, there is no feasible walk from d to any other vertex because there is no directed hyperedge starting from d. (Similarly for c and f.)

Design and analyze an algorithm that, given a directed hypergraph $G = (V, E)$, and vertices s and t, decides if there is a feasible walk from s to t. When analyzing the running time, we let $n = |V|$ and $m = |E|$. You can assume that E is given as a list of triplets.

Exercise 6.10. Let $G = (V, E)$ be a directed graph. We say an edge (u, v) is *irreversible* if there is no walk from v to u in the graph. Design and analyze an algorithm (the faster the better) that computes all the irreversible edges in G.

In Section 6.2, we asked if a vertex s can reach all other vertices in a particular graph. The answer is no:



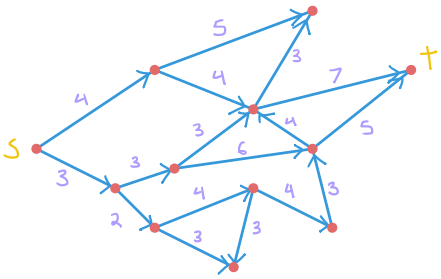
Chapter 7

Shortest walks

We have previously discussed algorithms for reachability: whether or not there *exists* a path between two vertices. Today's discussion is instead about finding the *shortest* path between two vertices.

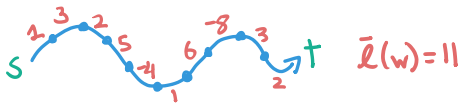
Of course this is a very practical problem; given a road map perhaps annotated with traffic (or weather!) conditions, we want the shortest driving directions from home to work. In a more general sense of planning problems, where vertices correspond to states, edges correspond to transitions between states, and edge lengths correspond to the time or cost of each transition, we may want to go from one state to another with minimum total cost.

7.1 Distances



Let $G = (V, E)$ be a directed graph and let $\ell : E \to \mathbb{R}$ assigned real-valued *lengths* to all the edges. The *length of a walk* w is defined as the sum of edge lengths. We denote the length of w as

$$\bar{\ell}(w) \stackrel{\text{def}}{=} \sum_{e \in w} \ell(e)$$



Here the sum is over all edges in the walk w (with repetition).

The *distance* between two vertices s and t is defined (mathematically) as the

infimum[1]over the lengths of all walks from s to t:

$$d(s,t) = \inf\Big\{\bar{\ell}(w) : w \text{ is a walk from } s \text{ to } t\Big\}.$$

By definition, this quantity is ∞ if s cannot reach t, and $-\infty$ if this quantity is arbitrarily small[2]. This chapter is about the problem of computing the distance from one vertex s to another vertex t.

The most basic setting is unweighted graphs (i.e., $\ell(e) = 1$ for all e). Then the distance is the minimum number of edges required to get from s to t. The next most basic setting is positively weighted graphs. That is, $\ell(e) > 0$ for all edges e. The most general allows for both positive and negative weights. This chapter focuses on the unweighted and positive settings and defers negative edge weights to the following chapter.

Thus let $G = (V, E)$ have positive edge weights $\ell : E \to \mathbb{R}_{>0}$. For simplicity, the first-time reader may want to assume all edges have length 1. Given two vertices $s, t \in V$, the goal is to compute the length of the shortest walk from s to t. Observe that for positive weights, the shortest (s,t)-walk is always a path. (Why?) So the problem is the same as finding the shortest (s,t)-path.

7.2 Distances in a DAG

Let $G = (V, E)$ be a DAG, with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. Given $s, t \in V$, the goal is to compute the shortest (s,t)-path. Towards a recursive algorithm let us declare

> `distance(`v`)` as the length of the shortest path from s to v.

We want to compute `distance(`t`)`. Now, if $t = s$, then the distance is 0. Otherwise, there must be some vertex v preceding t on the shortest (s,t)-path. Then `distance(`t`)` $=$ `distance(`v`)` $+ \ell(v,t)$. But which in-neighbor v? Who knows. Let us recursively compute `distance(`v`)` over all incoming edges (v,t) and choose the one minimizing `distance(`v`)` $+ \ell(v,t)$. All put together we have the following recursive

[1]The infimum is defined as the greatest lower bound: the greatest value x that is less than all values in the corresponding set. It is usually the same as minimum but allows for $-\infty$ when the values are not bounded below and $+\infty$ when the set is empty.

[2]More formally: the distance is $-\infty$ if for all $L \in \mathbb{R}$, there exists an (s,t) walk w of total length $\bar{\ell}(w) \leq L$

algorithm.

$$\texttt{distance}(t) = \begin{cases} 0 & \text{if } t = s, \\ +\infty & \text{if } t \text{ has no incoming edges,} \\ \min_{(v,t)\in\delta^-(t)} \texttt{distance}(v) + \ell(v,t) & \text{otherwise.} \end{cases}$$

Why is the algorithm correct? Why does it avoid recursive loops? What is the underlying inductive hypothesis?

Let $v_1, \ldots, v_n$ list the vertices in topological order, and consider the recursive spec again in light of this ordering.

`distance`(v_i): the length of the shortest path from s to v_i.

We claim that the recursive algorithm above implements the recursive spec by induction on i.

In the base case, $i = 1$, and v_1 is a source. Either $v_1 = s$ and $d(s, v_1) = 0$, or $v_1 \neq s$ and v_1 is unreachable from s, hence $d(s, v_1) = +\infty$. This agrees with the code.

In the general case, $i > 1$. By induction, $\texttt{distance}(v_j) = d(s, v_j)$ for all $j < i$. If $v_i = s$ then $\texttt{distance}(v_i) = 0 = d(s, v_i)$. Otherwise, $v_i \neq s$, and $d(s, v_i)$ is the infimum length over all walks from s to v_i. For all arcs (v_j, v_i) with head v_i, we have $j < i$, hence $\texttt{distance}(v_j) = d(s, v_j)$ by induction. Thus

$$\texttt{distance}(v_i) = \inf_{(v_j,v_i)} \texttt{distance}(v_j) + \ell(v_j, v_i) = \inf_{(v_j,v_i)} d(s, v_j) + \ell(v_j, v_i).$$

By the triangle inequality,

$$d(s, v_i) \leq d(s, v_j) + \ell(v_j, v_i)$$

for all arcs (v_j, v_i), hence

$$\texttt{distance}(v_i) \geq d(s, v_i).$$

In particular, if $d(s, v_i) = +\infty$, then $\texttt{distance}(v_i) = +\infty$.

If $d(s, v_i)$ is finite, then let (v_j, v_i) be the last edge on the shortest walk from s to v_i. We have

$$\texttt{distance}(v_i) = \inf_{(v_k,v_i)} d(s, v_k) + \ell(v_k, v_i) \leq d(s, v_j) + \ell(v_j, v_i) = d(s, v_i).$$

Thus $\texttt{distance}(v_i) = d(s, v_i)$, as desired.

Running time. The initial recursive algorithm may not be fast, but we can make it efficient by caching the answers to each $d(v_i)$. With dynamic programming, the running time becomes the sum, over all v_i, of the number of incoming edges to v_i (plus a constant). This gives a $O(m+n)$ running time in total.

7.3 Unweighted shortest paths

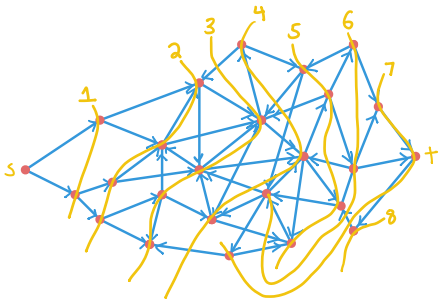
We now consider shortest paths in general directed graphs. To ease into it we first focus on unweighted graphs, where the length of a walk is the number of edges.

When working with a DAG above, we could rely on the topological ordering to provide an inductive structure. General graphs do not have a topological order and we do not know *a priori* in which order to calculate the vertex distances. However, we do know:

0. *The vertices at distance* 0*:* This is just $\{s\}$.
1. *The vertices at distance* 1*:* These are just the vertices v with an arc (s, v) from s.
2. *The vertices at distance* 2*:* These are the vertices v that (a) have an arc (u, v) from some vertex u at distance 1, and (b) aren't at distance 0 or 1.
3. *The vertices at distance* 3*:* These are the vertices v that (a) have an arc (u, v) from some vertex u at distance 2, and (b) aren't at distance 0, 1, or 2.

Clearly a pattern has started to emerge. In general, for any integer k,

k. *The vertices at distance k:* the vertices v that (a) have an arc (u, v) from some vertex u at distance $k-1$, and (b) are not at distance $0, \ldots, k-1$.



Note the inherent inductive structure in our observations: we can identify the distance k vertices only once we have identified the distances of all vertices at distance $\leq k-1$. We can encode our observations into an algorithm as follows.

`breadth-first-search (BFS)`$(G = (V, E),\ s \in V)$

1. Set `distance`$(s) = 0$.
2. For $k = 0, 1, 2, \ldots$, as long as there is at least one vertex u with `distance`$(u) = k$:
   A. For all vertices u such that `distance`$(u) = k$:
      1. For all arcs (u, v) for which `distance`(v) is not yet set:
         a. Set $d(v) = k + 1$.

Proof of correctness. One can formally prove correctness by induction on k. The induction hypothesis is that for all $k \in \mathbb{Z}_{\geq 0}$,

$$\{v : d(s, v) = k\} = \{v : \texttt{distance}(v) = k\}, \tag{7.1}$$

where $d(s, v)$ refers to the true distance from s to v, and $d(v)$ is the value assigned by the algorithm. In the base case, $k = 0$, and s is the only vertex at distance 0 and the only vertex labeled 0. For $k \geq 1$, assume by induction that (7.1) holds for all distances $i < k$.

As argued above, the vertices at (true) distance k are those that (a) have an edge from a vertex at distance $k - 1$, and (b) are not at distance less than k. By induction on k, the vertices at distance less than k are labeled correctly by their distance, so (b) is the same as saying that the vertex is unlabeled after $k - 1$ iterations. Indeed, in the kth iteration, the algorithm labels all unlabeled vertices with arcs from vertices at label $k - 1$.

Running time. It is easy to see that the algorithm runs in linear time with some obvious bookkeeping (like keeping track of all the vertices at distance k for each k). Each vertex assumes the role of u once in the loop at `(2.A)`. We then traverse each outgoing edge (u, v) from u and that is the only time we examine that edge. Overall we process each vertex and each edge once, and we have a $O(m + n)$ time overall.

An implicit DAG. The algorithm above is implicitly very similar to our algorithm for DAG's. Suppose we drop from G all the edges that do not appear in shortest paths from s. Every remaining edge goes from a vertex at distance k to a vertex distance $k + 1$ for some k. (Such an edge is sometimes called a *tight edge*.) This subgraph is a DAG, and ordering the vertices by distance from s gives a topological ordering.

Queue-based implementation. There is a more compact way to implement `BFS` relying on a queue to order the vertices as they are processed. Consider the following code.

The high level idea is that the distance labels moving through the queue are increasing, and a vertex of distance label k is dequeued only after all vertices of

BFS($G = (V, E)$, $s \in V$)

/* *A queue-based implementation of BFS.* */

1. Initialize an empty queue Q.
2. Set distance(s) $= 0$ and insert s into the queue Q.
3. While the queue Q is not empty.
   A. Dequeue the first vertex u in Q.
   B. For each arc (u, v) for which distance(v) is not yet set:
      1. Set distance(v) $=$ distance(u) $+ 1$.
      2. Enqueue v into Q.
4. Return all the distance labels.

distance label $k - 1$ have been completely processed. We leave it to the reader to formally verify that the queue-based version above simulates the first version. It can also be understood as a specialization of the upcoming algorithm for weighted graphs.

7.4 Weighted shortest paths

We now move on to the weighted case. Let G have positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. Again we want to find the lengths of the shortest paths from a fixed vertex s.

In the unweighted case, we knew that all the distances were integers, and we could inductively partition the vertices by distance layers. Weighted distances are not so simple to enumerate. That said, we do know that:

1. The closest vertex from s is s, at distance $d(s, s) = 0$.
2. The second closest vertex from s is the head v_2 of the edge $e = (s, v_2)$ of minimum length $\ell(e)$; we then have $d(s, v_2) = \ell(e)$.

Things get a little more interesting when it comes to identifying the third closest vertex, which we denote v_3. v_3 might be the head of the second smallest edge (s, x) leaving s, but it does not have to be. Maybe there is an edge (v_2, y) leaving v_2 such that the combined distance $d(s, v_2) + \ell(v_2, y)$ is less than the length of any edge (s, x) leaving s. We will also consider this possibility and take the minimum of either scenario, as follows.

3. The third closest vertex v_3 is, among all vertices outside of $\{v_1 = s, v_2\}$ either (a) the head x of the edge $e = (s, x)$ of minimum length $\ell(e)$ or (b) the head x

of the edge $e = (v_2, x)$ minimizing $d(s, v_2) + \ell(e)$. In case (a), the distance to x is $\ell(s, x)$; in case (b), the distance to x is the sum $d(s, v_2) + \ell(v_2, x)$.

This sets a pattern where the next vertex is based on the distances via all preceding vertices. Inductively, for $k \in \mathbb{N}$, if we let $v_1, \ldots, v_{k-1}$ denote the $k-1$ closest vertices from s:

k. The kth closest vertex v_k is, among all vertices outside of $\{v_1, \ldots, v_{k-1}\}$, the head x of the edge $e = (v_i, x)$ minimizing $d(s, v_i) + \ell(e)$, over $i \in \{1, \ldots, k-1\}$.

Similarly to unweighted distances, an inductive structure has emerged. *A priori* it is not clear which vertex is the kth closest from s. But it is much easier after identifying the first $k-1$ vertices in distance from s.

We transfer our observations to the following algorithm sketched at a high-level below. (More concrete implementation details will be supplied momentarily). The algorithm is named after Edgar Dijkstra who published it in 1959.

`Dijkstra(`$G = (V, E)$`, `$\ell : E \to \mathbb{R}_{>0}$`, `$s \in V$`)`

/ We assume for simplicity that all vertices are reachable from s; other vertices would be ignored. */*

1. Set `distance(`s`)` $= 0$.
2. Until all vertices (reachable from s) are assigned a distance:
   A. Let $u, v \in V$ minimize `distance(`u`)` $+ \ell(u, v)$ over
      - all vertices u that have been assigned a distance,
      - all edges (u, v) leaving u, and
      - restricting to heads v that have not yet been assigned a distance.
   B. Set `distance(`v`)` $=$ `distance(`u`)` $+ \ell(u, v)$.
3. Return all the distance labels.

Figure 7.1: High-level implementation of Dijkstra's algorithm for single-source shortest-paths.

Proof of correctness. One can formally prove correctness by induction. Let $(v_1 = s), v_2, v_3, \ldots$ be the vertices processed by the algorithm, in order. We claim that for all $i \in \mathbb{N}$, v_i is the ith closest vertex to s, and the algorithm sets `distance(`v_i`)` $= d(s, v_i)$.

The base case is $i = 1$, in which case $v_1 = s$ and `distance(`s`)` $= 0 = d(s, s)$. Let $i > 1$. By induction, $v_1, \ldots, v_{i-1}$ are the $i-1$ closest vertices (in order), and each have their distance label set to their distance from s. Above we observed that the ith closest vertex is the head x of the arc $e = (v_j, x)$ minimizing $d(s, v_j) + \ell(e)$, over

$j \in \{1, \dots, i-1\}$. By induction, $d(s, v_j) = d(v_j)$ for all $j \in \{1, \dots, i-1\}$, so the ith closest vertex is also the head x of the arc $e = (v_j, x)$ minimizing $d(v_j) + \ell(e)$, over $j \in \{1, \dots, i-1\}$. The latter is exactly how the algorithm selects and labels v_i.

Remark 7.1. The proof above is unique from previous inductive proofs we've seen for the following reason: the ordering $v_1, v_2, \dots, v_n$ on which we make our induction argument is not given *a priori*, but emerges from the algorithm itself.

Running time. `Dijkstra`'s algorithm is following our inductive approach: the next distance we identify is based on the distances identified so far. As given, however, it is fairly slow. Each iteration of the outer loop produces one distance label. The minimization in `(2.A)` is implicitly another loop: over all labeled vertices u and all edges leaving u. This adds up to a loop over all the edges in the graph and takes $O(m)$ time. So we have a $O(mn)$ time algorithm for computing distances from s.

Speeding up Dijkstra's algorithm. The bottleneck is step `(2.A)`. Identifying the next closest unlabeled vertex takes $O(m)$ time, and we do this $O(n)$ times. This approach treats each search for u and v as completely independent from one iteration to the next. But of course the searches have a lot in common. They are all in the same graph. The only difference between one iteration of `(2.A)` and the next is that one more vertex has been labeled, opening up one more option for u.

For each unlabeled vertex v, consider the minimum of $\texttt{distance}(u) + \ell(u, v)$ over all incoming edges (u, v) where u is already labeled (or $+\infty$ if there is no such u). This value represents the length of the shortest path to v *so far*; let us call it the "tentative distance" to v. Each iteration we are identifying the vertex v of minimum tentative distance, and finalizing that tentative distance the real distance to v. Labeling v may then decrease the tentative distance to vertices w where there is a directed edge (v, w). But all the other tentative distances stay the same. Perhaps we could track the tentative distances in a *data structure*, that only needs to be updated when tentative distances are revised. This data structure should also be able to quickly provide the next vertex of minimum tentative distance.

Such a data structure is given by a heap or priority queue. We assume the reader is acquainted with heaps but we briefly review the key points. A *priority queue / heap* is a data structure designed to keep track of the key with the smallest value over a collection of key-value pairs. In this context the value is sometimes called a "priority". For the sake of our discussion, a heap is defined by the following three basic operations. (Here we describe a min-heap but a max-heap can be defined analogously.)

1. `insert(`k`,`p`)`: inserts an key k with priority p.
2. `decrease(`k`,`p`)`: Let k be a key in the heap. If p is less than the current priority of k, decrease the priority to p.

`Dijkstra`$(G = (V, E), \ell : E \to \mathbb{R}_{>0}, s \in V)$

/ An implementation of Dijkstra's algorithm where a priority queue keeps track of the unlabeled vertex of minimum tentative distance. */*

1. Set `tentative`$(s) = 0$ and `tentative`$(v) = +\infty$ for all $v \neq s$.
2. Let Q be a (min) priority queue / heap over V with priorities `tentative`(v).
3. While Q is not empty
   A. Remove the vertex v of minimum `tentative`(v) from Q.
   B. Set `distance`$(v) =$ `tentative`(v).
   C. For each outgoing edge (v, w):
      1. Set

$$\texttt{tentative}(w) = \min\{\texttt{tentative}(w), \texttt{distance}(v) + \ell(v, w)\}$$

      and decrease w's priority in Q accordingly.
4. Return the distance labels.

Figure 7.2: Accelerating Dijkstra's algorithm with a priority queue.

3. `remove-min()`: removes and returns the key value pair (k, p) with the minimum priority p.

A standard array-backed heap takes $O(\log n)$ time for each of the above operations, where n is the number of items in the heap.

To accelerate Dijkstra's algorithm, we can use a priority queue over the unlabeled vertices using the tentative distance as the priority. To keep the priority queue updated, whenever we label a vertex u, for each edge $(u, v) \in E$, we may revise the tentative distance to v and update the heap via the `decrease` operation. By keeping the priorities updated, we can retrieve the next vertex v with a call to `remove-min()`. See Fig. 7.2.

As mentioned above, each heap operation takes $O(\log n)$ time in an array-backed heap. Thus, when using an array-backed heap, Dijkstra's algorithm takes

$$\begin{aligned} &O(\log n) \times (\#\ \texttt{decrease-key}\text{'s}) + O(\log n) \times (\#\ \texttt{remove-min}\text{'s}) \\ &= O(m \log n) + O(n \log n) = O(m \log n). \end{aligned}$$

This is a considerable improvement on $O(mn)$. It is also a very reasonable running time; it is as if we were sorting all the edges. Still, one can do better with the following remarkable data structure due to Fredman and Tarjan [FT87].

Fact 7.2. *There exists a data structure, called a* Fibonacci heap*, that has the following guarantee. Over any sequence of I calls to* `insert-key`*, D calls to* `decrease-key`*, and R calls to* `remove-min`*, the Fibonacci heap takes*

$$O(I + D + R \log n)$$

time in total.

Note that the running time above only guarantees that each `decrease-key` takes $O(1)$ time *on average*, but a single operation may take longer in the worst-case. This "on-average" guarantee is fine for our purposes where we are only concerned in the total running time over the course of a shortest path algorithm. Such averaging types of guarantees are proven by a technique called *amortized analysis*. We will come back to amortized analysis later in the course where we may analyze Fibonacci heaps among other data structures. For the moment, we take fact 7.2 as given.

If we use Fibonacci heaps, then the running time becomes

$$\begin{aligned} &O(1) \times (\#\ \texttt{decrease-key}\text{'s}) + O(\log n) \times (\#\ \texttt{remove-min}\text{'s}) \\ &= O(m + n \log n). \end{aligned}$$

In conclusion, we have the following remarkable running time for computing shortest paths with positive edge weights.

Theorem 7.3. *Let $G = (V, E)$ be a directed graph with positive edge weights $\ell : E \to \mathbb{R}_{>0}$ and let $s \in V$. Then Dijkstra's algorithm (with Fibonacci heaps) computes the shortest path from s to every other vertex $t \in V$ in $O(m + n \log n)$ time.*

7.5 Additional notes and materials

Spring 2026 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

7.6 Exercises

Exercise 7.1. Each of the following problems describes a graph problem over an unweighted directed graph $G = (V, E)$ with m edges and n vertices. Each problem can be solved by constructing an auxiliary graph and calling a shortest path algorithm from this chapter as a black box. For each problem, (a) design an auxiliary graph and shortest path problem, (b) indicate which algorithm to apply and how, and (c) analyze the running time. No proof of correctness is required.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[3]

2. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is a multiple of 5.

3. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is either a multiple of 2, or a multiple of 3.

4. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is either a multiple 3 or a multiple of 4, but not a multiple of 12.

5. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is of the form $5x + 8y$ for $x, y \in \mathbb{Z}_{\geq 0}$.

6. Suppose G is a patriotic directed graph where all the edges are colored either red, white, or blue. An *American walk* is a walk where the edges alternate in color in the order of the American dream: red, white, blue, red, white, blue... Here the first edge could be any color and then the colors have to cycle through

[3] For this and the other problems, if there is no such walk, then your algorithm should return $+\infty$.

the American dream thereafter. Compute the length of the shortest American walk from s to t.

7. Suppose G is again a patriotic directed graph where all the edges are colored either red, white, or blue. An *un-American walk* is a walk that never cycles through the American dream; that is, a walk where no three consecutive edges in the walk have colors that form any of the following three sequences: $(\text{red}, \text{white}, \text{blue})$, $(\text{white}, \text{blue}, \text{red})$, or $(\text{blue}, \text{red}, \text{white})$. Compute the length of the shortest un-American walk from s to t.

8. Let $s, t \in V$ be two vertices. Consider the following game with two people Alice and Bob. Initially, Alice is on s, and Bob is on t. Each round, Alice and Bob are on two vertices, and they simultaneously take an outgoing edge to another vertex. The goal is to guide Alice and Bob to the same vertex with the minimum number of rounds or declare that it is impossible to have them meet. Compute the minimum number of rounds needed to have Alice and Bob meet at the same vertex.

Exercise 7.2. Let $G = (V, E)$ be a directed graph with positive edge weights, and $s, t \in V$. Recall that Dijkstra's algorithm returns the lengths of the shortest paths from s (as well as the shortest paths themselves by incorporating parent pointers). However there may be many shortest (s, t)-paths for a particular t. Suppose we want to find the shortest (weighted) (s, t)-paths with the fewest number of edges (to break ties). Consider the problem of computing both the shortest path lengths as well as the minimum number of edges in each shortest path. Design and analyze an algorithm for this problem.[4]

Exercise 7.3. Let $G = (V, E)$ be a directed graph with positive edge weights $\ell : E \to \mathbb{R}_{>0}$. Let $s, t \in V$ be distinct vertices such that s can reach t. Recall that the distance from s to t can be computed in $O(m + n \log n)$ time, and that there may be many shortest (s, t)-walks. We say a vertex v is *on the way* from s to t if v is on a shortest (s, t)-walk. (Yes, s and t themselves are both on the way from s to t. And yes, with positive edge weights, all shortest walks are shortest paths.)

Consider the problem of computing the set of all vertices v on the way from s to t. Design and analyze an algorithm for this problem.

[4]Technically, there could be as many as $n!$ paths in which case an arithmetic operation would take $O(\log(n!)) = O(n \log n)$ time. For this exercise, let us assume for simplicity that arithmetic operations take $O(1)$ time anyway.

Exercise 7.4. This problem is meant to model the following scenario. Suppose you drive to campus but you are running late for the CS580 midterm, and in this extreme circumstance you are willing to speed up to 25% over the speed limit. However, you don't want to run too high a risk of getting a ticket, so you decide you are willing to speed for a limited number k of streets. The goal is to get to the exam as fast as possible, going 25% over the speed limit for up to k streets, and obeying the speed limit everywhere else.

Formally, the input consists of a directed graph $G = (V, E)$, positive edge weights $\ell : E \to \mathbb{R}_{>0}$, vertices $s, t \in V$, and an integer $k \in \mathbb{N}$. The *k-speeding distance* from s to t is defined as the minimum total length of any (s,t)-walk, except there the k largest edges are decreased by a factor of $4/5$. For example, if $k = 3$, and you have an (s,t)-walk with edge lengths $6, 3, 4, 6, 9, 3, 6, 9, 7, 1$, then the total length including the k "speedups" becomes

$$6 + 3 + 4 + 6 + \frac{4}{5} * 9 + 4 + 6 + \frac{4}{5} * 9 + \frac{4}{5} * 7 + 1 = 50.$$

Design and analyze an algorithm computing the k-speeding distance from s to t. Your running time may depend on k. (You may assume $k < n$ since any shortest walk should be a path anyway).

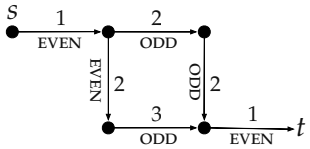
Exercise 7.5. Red light, green light! The following model is inspired by streetlights which, at a given point in time, allow traffic to enter some streets and block traffic from entering others.

Let $G = (V, E)$ be a directed graph with positive, integer edge lengths $\ell(e)$ for $e \in E$, which represents the amount of time it takes to traverse that edge. (Here time will always be an integer.) Suppose each edge is also annotated as being either "even" or "odd". The model is as follows.

We start at a vertex s at time $x = 0$, and want to get to a vertex t as soon as possible. In general, taking an edge e takes $\ell(e)$ units of time, which is added to x. However, we can only take an even edge when the time x is even, and we can only take an odd edge when the time x is odd. We can also wait at a vertex for one unit of time during which x increases by 1.

Loosely speaking, one can imagine each vertex as being like a stop light. An even edge is like a one-way street that traffic can enter only when the time x is even. An odd edge is like a one-way street that traffic can enter only when the time x is odd.

In the example on the right, the (s,t)-path along the top takes $1 + 2 + 2 + 1_{(\text{WAIT})} + 1 = 7$ units of time. (Here we annotated the 1's that correspond to waiting). The path along the bottom takes $1 + 1_{(\text{WAIT})} + 2 + 1_{(\text{WAIT})} + 3 + 1 = 9$ units of time.



For $s, t \in V$, consider the problem of computing the minimum amount of time needed to get from s to t in the time model described above. Design and analyze an algorithm for this problem.

Exercise 7.6. Suppose you are a city planner, and you are worried that it takes too long for an ambulance to get from the university to the hospital. You've decided to build one new street to improve this particular route. You and your crack team have researched and gathered a list of potential streets that could be built; the task now is to choose one street that will improve this route the most.

To model this formally, let $G = (V, E)$ be a directed graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$, and let $s, t \in V$ be two vertices. Let E be partitioned into two nonempty disjoint sets E_0 and E_1. E_0 represents the edges/streets that are "already built"; E_1 represents the edges/streets that you can add to the graph. Let $G_0 = (V, E_0)$ denote the graph of already built edges.

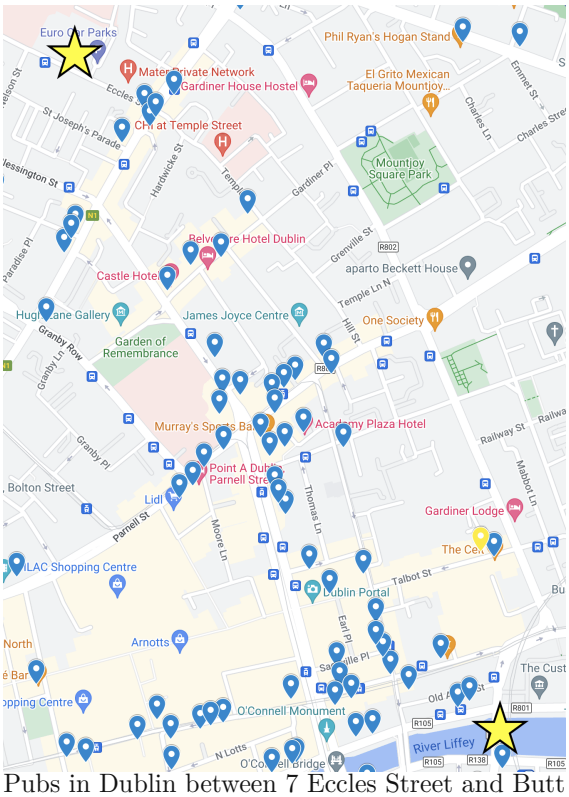
Consider the problem of identifying the single edge $e \in E_1$ so that, when you add e to G_0, the distance from s to t in the augmented graph is as short as possible. Design and analyze an algorithm for this problem.

As a follow up: What if you can add two edges? Design and analyze an algorithm for this problem.

Exercise 7.7. Bloomsday is a commemoration and celebration of the life of Irish writer James Joyce, observed annually in Dublin and elsewhere on 16 June, the day his 1922 novel Ulysses takes place on a Thursday in 1904, and named after its protagonist Leopold Bloom.

This year for Bloomsday you've decided to celebrate with your own truly epic, special kind of bar crawl. You will start the crawl at Leopold Bloom's house at 7 Eccles Street. (The house was demolished in 1967, and the site is now occupied by the Mater Private Hospital.) You will end the crawl at Butt Bridge, by the Custom House, where the book ends.

Your goal is to get from 7 Eccles Street to the Butt Bridge as fast as possible, but there's a catch: You are not allowed to walk for more than 6.5 minutes without dropping into a pub for a drink. You might have to go out of your way to keep yourself properly hydrated. Moreover, every time you drop into a pub, you have to spend 30 minutes inside. Taking these requirements into account, what was originally a 23 minute walk (according to Google maps) may take all day after all.

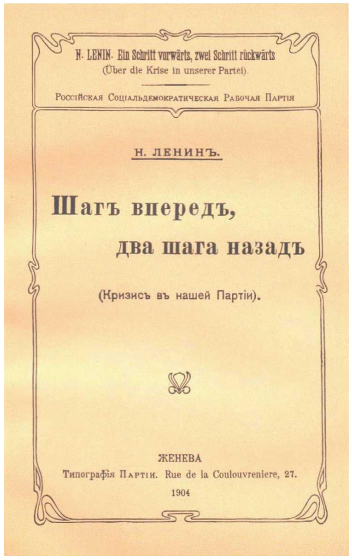


Pubs in Dublin between 7 Eccles Street and Butt Bridge

Design and analyze an algorithm (the faster the better) that computes the infimum amount of time it takes to get from 7 Eccles Street to Butt Bridge according to the requirements above. (If it is impossible to do so, the algorithm should output $+\infty$). To model the map, let $G = (V, E)$ be a directed graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. $\ell(e)$ represents the amount of time it takes to walk edge e. Let $s, t \in V$ represent 7 Eccles Street and the Butt Bridge, respectively. Let $P \subseteq V$ represent the pubs and let $k = |P|$. (Your running time may or may not depend on k, and we're focused on the regime where k is much smaller than n.) The algorithm should compute the minimum amount of time it takes to get from s to t subject to the festive requirements described above.

Exercise 7.8. *One Step Forward, Two Steps Back: The Crisis in Our Party* (Russian, romanized: *Shag vperyod, dva shaga nazad (Krizis v nashey partii)*) is a work written by Vladimir Lenin and published on May 6/19, 1904. In it Lenin defends his role in the 2nd Congress of the Russian Social Democratic Labour Party, held in Brussels and London from July 30 to August 23, 1903. Lenin examines the circumstances that

resulted in a split within the party between a Bolshevik ("majority") faction, led by himself, and a Menshevik ("minority") faction, led by Julius Martov.



Written in 1904 in response to controversies within the Social Democratic Labour Party's Second Congress regarding the status of party membership and organization, Lenin frames this conflicting factionalism within the Party in terms of dialectics. According to Lenin, there are two conflicting factions within the party: "the revolutionaries", which consists of the majority of party members (the Bolsheviks) and "the opportunists", which are the minority (the Mensheviks).

Rosa Luxemburg, a Marxist living in Germany at the time, responded to the article in *Organizational Questions of the Russian Social Democracy* (1904). She criticized Lenin's attitude towards democratic centralism and wrote about the role of "spontaneity" among the working class. However, different authors make varying claims about her precise attitude towards Lenin, the Bolshevik faction and the revolutionary situation in Russia.

Let $G = (V, E)$ be a directed graph. A *Lenin walk* is defined as a sequence of vertices $v_1, ..., v_k$ such that for each consecutive pair of vertices (v_i, v_{i+1}):

- If $i \mod 3 = 1$, then $(v_i, v_{i+1}) \in E$.
- If $i \mod 3 \in \{0, 2\}$, $(v_{i+1}, v_i) \in E$.

If we call each pair (v_i, v_{i+1}) in the Lenin walk a "step", then a Lenin walk alternates between one step going forwards along an edge and two steps each going "backwards", in the opposite direction of an edge.

Given G and $s, t \in V$, consider the problem of computing the infimum number of vertices along any Lenin walk from s to t. Design and analyze an algorithm for this problem.

Exercise 7.9. Tippecanoe County has decided to build a new hospital and call on you, seasoned and expert algorithmist, to help them figure out where to place it.

When planning hospitals, the high-level goal is to make sure that everyone in the town lives close to *some* hospital *on average*. We measure this by taking the average, over all homes in Tippecanoe Count, of the distance from the home to its nearest

hospital. Tippecanoe County already has several hospitals[5]; your goal is to place the next hospital to improve this metric as much as possible.

Your input is a directed graph $G = (V, E)$ representing all the locations and roads in Tippecanoe county. Each edge e has a label $\ell(e) > 0$ representing the length of that edge. Let $A \subseteq V$ denote the set of all current hospitals, $B \subseteq V$ the set of possible locations for new hospitals, and $C \subseteq V$ the set of all homes. Let $\alpha = |A|$, $\beta = |B|$, and $\gamma = |C|$. We assume that A, B, C are disjoint and $\alpha \leq \beta \leq \gamma$. We also assume that G is strongly connected.

1. (7 points) The first task is to evaluate the average distance (as described above) for the current set of hospitals. Consider the problem of computing the average distance from each home to its nearest hospital. Design and analyze an algorithm for this problem.

2. (3 points) Now consider the planning problem. The goal is to add one more hospital (from Y) to minimize the resulting average distance from each home to its nearest hospital. Design and analyze an algorithm for this problem.

(For either problem, if you decide to design and analyze an algorithm, then your running times may depend on $\alpha = |A|$, $\beta = |B|$, and/or $\gamma = |C|$.)

Exercise 7.10. The Economist recently reported[6] that the number of toll roads in America is increasing rapidly. It used to be hard enough to find the shortest path. Now you have to do it within a budget.

Right now, drivers pay to access 6,300 miles of Americas 160,000 miles (or so) of highway, but that number is going up. Here in Indiana, in June, the state passed a law giving the state authority to raise tolls on all its existing interstate highways. The Economist says that this trend is broadly due to a parallel trend of decreasing taxes on gasoline, which used to pay a larger share of the cost for maintaining roads and highways.

Let $G = (V, E)$ be a directed graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$ and *nonnegative integer edge costs* $\texttt{cost} : E \to \mathbb{Z}_{\geq 0}$, which model toll roads. We define the *cost* of a walk in G as the sum of edge costs (with repetition) along that walk. Design and analyze an algorithm (the faster the better) that, given $k \in \mathbb{N}$, and $s, t \in V$, computes the length of the shortest (s, t)-walk that costs at most k. Your running

[5] In reality, Tippecanoe County has five hospitals. See https://www.in.gov/health/reports/QAMIS/hosdir/ctyfac78.htm

[6] *Toll roads are spreading in America*, December 18, 2025. https://www.economist.com/united-states/2025/12/18/toll-roads-are-spreading-in-america

may depend polynomially on k (even though k only takes $\log(k)$ bits to represent). (It might help to imagine $k \leq O(n)$.)

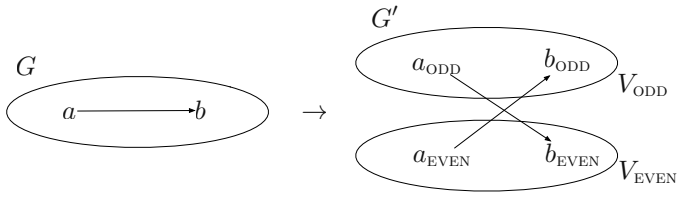
Exercise 7.11. The racetrack problem in Chapter 5 of [Eri19].

Sample solution.

Problem. (Exercise 7.1 problem 1) Let $G = (V, E)$ be an unweighted directed graph with m edges and n vertices. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[7]

Solution.
We reduce the problem to shortest paths in an appropriate auxiliary graph.
`shortest-even-walk(`$G = (V, E)$`, s, t)`

1. We build the directed graph $G' = (V', E')$ where:
   A. For each vertex v, we have two copies v_{EVEN} and v_{ODD} in V'.
   B. For every edge $(u, v) \in E$, we have two edges $(u_{\text{EVEN}}, v_{\text{ODD}})$ and $(u_{\text{ODD}}, v_{\text{EVEN}})$ in E'.
2. We run BFS algorithm in G' from s_{EVEN} to find the length of the shortest walk/path from s_{EVEN} to t_{EVEN} in G', and return this value.



The high-level idea behind the auxiliary graph is to create a "two-layer" version of G, where edges go from one layer to the other. Starting from the "even" layer, any walk to a vertex in the "even" layer has an even number of edges, and any walk to a vertex in the "odd" layer has an odd number of edges.
To prove correctness, we describe a a length-preserving bijection between even-edge walks between s and t and walks between s_{EVEN} and t_{EVEN}. Indeed, for an even-edge walk $((v_0 = s), v_1, \ldots, (v_k = t))$, we have a walk in G' with edges $(v_{i,\text{EVEN}}, v_{i+1,\text{ODD}})$ for i even and $(v_{i,\text{ODD}}, v_{i+1,\text{EVEN}})$ for i odd. Conversely, given a walk $((v_{0,\text{EVEN}} = s_{\text{EVEN}}), v_{1,\text{ODD}}, \ldots, (v_{k,\text{EVEN}} = t_{\text{EVEN}}))$, we have a walk $(v_0, v_1, \ldots, v_k)$ in G with an even number of edges.
Consequently the minimum length walk from s_{EVEN} to t_{EVEN}, computed by BFS, gives the minimum length (s, t)-walk with an even number of edges in G.
Building G' takes $O(m + n)$, and running BFS algorithm in G' takes $O(m + n)$ time.

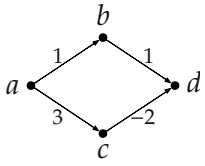
Chapter 8

Negative edge lengths and all-pairs shortest paths

Previously we discussed shortest path problems with positive edge weights, where the goal is to get from point a to point b as fast as possible. Here the points refer to vertices in a directed graph; the routes are the walks through the graph, and "fast" was based on the sum of edge lengths along the walk. Specifically, we obtained the linear time `BFS` algorithm for unweighted graphs and the nearly linear time Dijkstra's algorithm for positively weighted graphs.

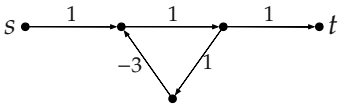
Positive edge lengths are a critical assumption for these algorithms. They permit an induction hypothesis increasing in distance; indeed, both algorithms identified and finalized distance labels in increasing distance from the source. But this argument breaks down for negative edge lengths.

Consider going from point a to point d in the weighted graph on the right. Dijkstra's algorithm would declare that point b was at distance 1, and that point d was at distance 2, before realizing that the path through c has total length 1. The shortest path does not necessarily go through closer vertices.



Let us state the problem formally. Let $G = (V, E)$ be a directed graph with real-valued (and possibly negative) edge lengths $\ell : E \to \mathbb{R}$. Fix $s, t \in V$. What is the length of the shortest walk from s to t?

The length can be smaller than any finite number, in which case we say that the distance is $-\infty$. This occurs if and only if we have a negative cycle on the way from s to t (as we will see). On the right, the negative cycle between s and t makes the distance $-\infty$.



8.1 Negative edges, shortest walks, and shortest paths

Recall the definition of distances in a directed graph on page 142. The distance from s to t is usually finite and equal to the length of the shortest walk from s to t, but

there are two non-finite exceptions. The distance is $+\infty$ if there is no walk, and the distance is $-\infty$ if for every value L, there is an (s,t)-walk of total length $< L$. That is, there is no finite lower bound on the lengths of (s,t)-walks. The $+\infty$ case is addressed by reachability. The $-\infty$ case did not arise for positive edge weights, because 0 is always a lower bound. But here, with negative edge weights, $-\infty$ can easily occur; e.g., with a negative cycle.

We address $-\infty$ head on, as well as some of the ambiguity in shortest *walks* vs shortest *paths*, with the following lemma.

Lemma 8.1. *Let $s, t \in V$ and suppose s can reach t. The distance from s to t is finite (and not $-\infty$) iff the shortest (s,t)-walk is attained by a path.*[1]

Proof. If the shortest (s,t)-walk is a path then by definition the (finite) length of this path is the (s,t)-distance.

Conversely, suppose the distance from s to t is a finite value L, which means it is attained by some (s,t)-walk. Among all (s,t)-walks of total length L, let w minimize the total number of edges along that path. We will show that w is a path. Indeed, if w is not a path, then it contains a cycle. The total length of that cycle falls into one of two cases:

1. The total length of the cycle is nonnegative, in which case removing the cycle decreases the number of edges in w without increasing the overall length – a contradiction to the choice of w.

2. The total length of the cycle is negative, in which case repeating the cycle decreases the overall length of the walk – a contradiction to the choice of w.

So w is a path. □

8.2 Shortest walks with negative edges

The problem is to find the shortest walk and there is no clear induction hypothesis. We can't rely on distances to implicitly supply an induction argument, as we did in developing BFS and Dijkstra's shortest path algorithms, when the edge lengths are negative. Any ideas for an algorithm that works around this?

[1] That is, there is an (s,t)-path with total length less than or equal to any other (s,t)-walk.

We *lift* the problem, so to speak, by inserting an additional parameter that makes it more conducive to an induction argument. For vertices $s, t \in V$ and an integer $k \in \mathbb{Z}_{\geq 0}$, we declare

> `SW(`s`,`t`,`k`)` $=$ the length of the shortest walk from s to t with at most k edges (defined as $+\infty$ if there is none).[2]

Here there is a clear induction argument on k – the k edge walks build on $k-1$ edge walks by appending an extra edge. In the code below, $\delta^-(t)$ denotes the set of edges (v,t) with head t.[3]

`SW(`s`,`t`,`k`)`

1. If $k = 0$:
   - */* The only zero edge walk is the empty walk. */*
   - A. If $s = t$ then return 0.
   - B. Otherwise return $+\infty$.
2. Return the minimum of:
   - A. `SW(`s`,`t`,`$k-1$`)`.
   - B. The minimum of `SW(`s`,`v`,`$k-1$`)` $+ \ell(v,t)$ over all edges $(v,t) \in \delta^-(t)$.

We cache the solutions to our algorithm above. With dynamic programming, for fixed s and t, and each $k \in \mathbb{Z}_{\geq 0}$, `SW(`$s$`,`$t$`,`$k$`)` takes $O(1 + d^-(t))$ time, where $d^-(v) = |\delta^-(v)|$ denotes the in-degree of v. For fixed s and k, it takes $O(m+n)$ time to compute all of the subproblems for a fixed value of k.

Lemma 8.2. *Let $G = (V, E)$ be a directed graph, with m edges, n vertices, and real-valued edge weights. Let $s \in V$ be fixed. In $O(k(m+n))$ time, one can compute* `SW(`s`,`t`,`i`)` *for all $t \in V$ and all $i \leq k$.*

Shortest paths when there are no negative cycles. Suppose all the distances from s are finite. Then by Lemma 8.1, the (s,t)-distance is attained by a path for every vertex t.

Corollary 8.3. *Let $G = (V, E)$ be a directed graph with m edges, n vertices, real-valued edge weights. Let $s \in V$ be fixed and suppose that all distances from s are finite. In $O(mn)$ time, one can compute the length of the shortest (s,t)-path for all $t \in V$.*

[2] Currently we are interested in the case where s is fixed, but later it will be convenient to have s as a parameter.

[3] More generally, for $S \subseteq V$, $\delta^-(S) \stackrel{\text{def}}{=} \{((u,v) \in E : u \notin S, v \in S)\}$.

Stationary points and $-\infty$. The remaining issue is $-\infty$. Intuitively, if $d(s,t) = -\infty$, then $\texttt{SW}(s,t,k)$ should decrease towards $-\infty$ as $k \to \infty$. That does not mean that $\texttt{SW}(s,t,k)$ decreases with *every* k; we may need several edges to go all the way around a negative cycle and decrease $\texttt{SW}(s,t,k)$. It's also not clear how long to keep iterating on $\texttt{SW}(s,t,k)$ before we can conclude that there is no lower bound. So we have to analyze $\texttt{SW}(s,t,k)$ carefully.

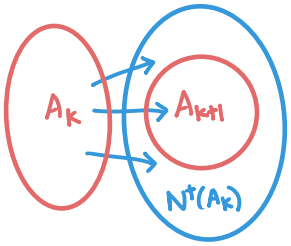
For $k \in \mathbb{N}$, let

$$A_k = \{v \in V \text{ where } \texttt{SW}(s,v,k) < \texttt{SW}(s,v,k-1)\}$$

be the set of vertices v where $\texttt{SW}(s,v,k)$ decreases compared to the previous iteration for $k-1$. Recalling the definition of $\texttt{SW}(s,v,k)$, these are the vertices where the best k-edge walk from s to v is strictly shorter than the best $(k-1)$-edge walk.

For each k, we have

$$A_{k+1} \subseteq N^+(A_k) \qquad \text{(why?)},$$



where for $U \subseteq V$, $N^+(U)$ denotes the "out-neighborhood" of U consisting of all vertices that are endpoint to some directed edge starting in U:

$$N^+(U) = \{v \text{ where } (u,v) \in E \text{ for some } u \in U\}.$$

Continuing on, we also have

$$\begin{aligned} A_{k+2} &\subseteq N^+\Big(N^+(A_k)\Big), \\ A_{k+3} &\subseteq N^+\Big(N^+\Big(N^+(A_k)\Big)\Big), \\ A_{k+4} &\subseteq N^+\Big(N^+\Big(N^+\Big(N^+(A_k)\Big)\Big)\Big), \end{aligned}$$

and so forth. In general, for any $\ell \geq k$, all the vertices in A_ℓ must be reachable from A_k. Conversely, for all vertices t that are *not* reachable from A_k, we know that $\texttt{SW}(s,t,\ell)$ will never change. They are forever *fixed points* in this process. Consequently $d(s,t) = \texttt{SW}(s,t,k-1)$ for any such t.

Consider A_n. These are the vertices v where the length of the shortest (s,v)-walk with at most n edges is less than the length of the shortest (s,v)-walk with at most $n-1$ edges. In particular, since any path has at most $n-1$ edges, the distance from (s,v) is *not* attained by a path. By Lemma 8.1, the distance from s to v must be $-\infty$.

So all vertices $v \in A_n$ have $d(s,v) = -\infty$. Additionally, any vertex v reachable from A_n has distance $-\infty$. (Why?) Meanwhile, if a vertex v is not reachable from A_n,

then (since $\texttt{SW}(s,v,\ell)$ would never change for $\ell \geq n$) we have $\texttt{SW}(s,v,n) = d(s,v)$. After running DFS from A_n, we'll have all the finite distances from s as well as all the vertices with distance $-\infty$. Funny enough, we churn $\texttt{SW}(s,t,n-1)$ for *just one more iteration* to get all the distances, finite or not.

Theorem 8.4. *In $O(mn)$ time, we can compute the (s,t)-distance for all vertices $v \in V$ (with the corresponding shortest path whenever the distance is finite).*

This algorithm is commonly referred to as the *Bellman-Ford algorithm* [Bel58; For56], although it was also described by Shimbel [Shi55]; an alternative, more descriptive name might be "that dynamic programming algorithm based on the number of edges for SSSP with negative edge lengths".

Detecting negative cycles. Suppose we want to decide if there is *any* negative cycle in the graph. The Bellman-Ford algorithm will detect if there is a negative cycle that is reachable from a particular source s. However there may be negative cycles that are not reachable from s. We can try all choices of s but this can be very slow. Can we do better?

We introduce an additional "super-source" vertex $s^\star$ with an edge $(s^\star, v)$ of length 0 for every $v \in V$. In particular $s^\star$ can reach every vertex directly. If we apply Theorem 8.4 from $s^\star$ (that is, compute $\texttt{SW}(s^\star,t,n+1)$ for all t), then any vertex on a negative cycle will have distance $-\infty$. In fact any vertex reachable from a negative cycle will also have distance $-\infty$.[4]

Corollary 8.5. *Given a directed graph G with real-valued edge weights, in $O(mn)$ time, one can identify all vertices reachable from a negative cycle.*

8.3 All pairs shortest paths and the doubling trick

We now consider the problem of computing the distances between *all pairs of vertices*, not just from a single source. We will focus on the case where G has no negative cycles for simplicity. Then all finite distances are obtained by shortest paths. Note that we can detect if there is a negative cycle using the Bellman-Ford algorithm (Corollary 8.5).

Recall our definition of $\texttt{SW}(s,t,k)$, which we repeat below:

[4] Don't change the algorithm, change the graph.

> `SW(`s,t,k`)` = the length of the shortest walk from s to t with at most k edges (defined as $+\infty$ if there is none).

The Bellman-Ford algorithm gave one implementation of `SW(`s,t,k`)` (on page 162). There, we had s fixed and varied t; now we are interested in all choices of $s, t \in V$. To this end, we could call `SW(`s,t,n`)` for all choices $s, t \in V$. With dynamic programming, this would take $O(mn^2)$ time. Can we do better?

Consider the case $k = 4$. When s was fixed we computed `SW(`s,t,k`)` based on the values of `SW(`$s,v,k-1$`)` for $v \in V$. Now that we're varying s as well, we could take advantage of `SW(`u,v,ℓ`)` for any $u, v \in V$ and $\ell \leq k$. In particular, any (s,t)-walk of at most 4 edges can be divided into two walks of at most 2 edges, one from s to u and another from u to t for some $u \in V$. Of course we can do this dividing in half for any even k; better yet, we should do this *only for powers of* 2. For k equal to powers of 2, consider the following alternative implementation for `SW`.

`SW(`$s,t,2^i$`)`

1. If $i = 0$, then return the minimum of:
   A. 0 when $s = t$
   B. $\ell(s,t)$ when $(s,t) \in E$.
   C. $+\infty$.
2. Return the minimum of
   A. `SW(`$s,t,2^{i-1}$`)`
   B. `SW(`$s,v,2^{i-1}$`)` + `SW(`$v,t,2^{i-1}$`)` for all $v \in V$. *// $O(n)$ time.*

The key difference between this implementation of `SW(`s,t,k`)` and Bellman-Ford is in line `(2.B)` – we are trying to concatenate two paths with (at most) 2^{i-1} edges to form an edge with at most 2^i edges.

Now, for fixed 2^i, there are $O(n^2)$ subproblems. Each subproblem has a loop of length n. Caching the recursive calls, we have the following.

Lemma 8.6. *Let $G = (V, E)$ be a directed graph with n vertices. In $O(n^3 i)$ time, one can compute `SW(`$s,t,2^j$`)` for all $s, t \in V$ and all $j \leq i$.*

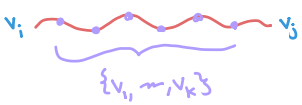
If there are no negative cycles (as we have assumed) then `SW(`$s,t,n-1$`)` (or any value greater than $n-1$) gives the distance from s to t. For $k = \lceil \log n \rceil$, Lemma 8.6 gives us the distances between all pairs of vertices.

Corollary 8.7. *Let $G = (V, E)$ be a weighted, directed graph with n vertices and no negative cycles. Then the distances between all pairs of vertices can be computed in $O(n^3 \log n)$ time.*

This running time is better than running Bellman-Ford from every s unless m is extremely close to n.

8.4 A different approach to APSP

We now consider a different approach to the all-pairs shortest-paths problem. This algorithm will only work if there are no negative cycles. In this case all distances are given by shortest paths.



Up to now, all the algorithms in this chapter were based on induction on the number of edges in the walk. Here we present a different approach, based on the subset of vertices used as intermediaries in the shortest path. Let us number the vertices $v_1, \ldots, v_n$ arbitrarily. We define:

> $\texttt{SP}(i,j,k)$ = the length of the shortest path from v_i to v_j using only $v_1, \ldots, v_k$ as intermediate vertices.

We implement the recursive spec as follows.

$\texttt{SP}(i,j,k)$

1. If $k = 0$:
   - A. If $i = j$ return 0.
   - B. If $(v_i, v_j) \in E$ then return $\ell(v_i, v_j)$.
   - C. Otherwise return $+\infty$.
2. Return the minimum of:
   - A. $\texttt{SP}(i,j,k-1)$
   - B. $\texttt{SP}(i,k,k-1) + \texttt{SP}(k,j,k-1)$

There are only $O(n^3)$ subproblems so we can apply dynamic programming and cache all the answers. Each subproblem takes constant time excluding recursive calls. So it takes $O(n^3)$ total time to compute $\texttt{SP}(i,j,k)$ for all i, j, k.

We prove the pseudocode implements the recursive spec by induction on k. The base case is $k = 0$, where there are no intermediate vertices. For fixed v_i and v_j, the only edge available is (v_i, v_j) (if present). Thus the only options for the distance is 0

when $v_i = v_j$, $\ell(v_i, v_j)$ if the edge (v_i, v_j) exists, and by default $+\infty$; the algorithm returns the minimum of these options.

Consider `SP(`i`,`j`,`k`)` for $k > 0$. The shortest (v_i, v_j)-path through $\{v_1, \ldots, v_k\}$ either uses v_k or it doesn't. If it doesn't, then by induction on k, its length is given by `SP(`i`,`j`,`$k-1$`)`. If it does, then the part from v_i to v_k is a path through $\{v_1, \ldots, v_{k-1}\}$ and the remaining part from v_k to v_j is also path through $\{v_1, \ldots, v_{k-1}\}$. By induction on k, `SP(`i`,`k`,`$k-1$`)` returns the length of the shortest path of the first type, and `SP(`k`,`j`,`$k-1$`)` returns the length of the shortest path of the second type. Their sum gives the length of the shortest (v_i, v_j) path through $\{v_1, \ldots, v_k\}$. This completes the proof of correctness.

This algorithm is commonly called the *Floyd-Warshall* algorithm although there are several relevant publications from roughly the same time [Flo62; War62; Roy59; Ing62] (see the Wikipedia article).

Theorem 8.8. *Let $G = (V, E)$ be a directed graph with real-valued edge weights and no negative cycles. Then one can compute all pairs shortest paths in $O(n^3)$ time.*

This is faster by a logarithmic factor than the running time obtained by the doubling trick in the previous section.

We should point out that when the edge weights are positive, you are better off running Dijkstra's algorithm from all $s \in V$. We point out that there is an extension of this approach for general edge lengths, called Johnson's algorithm [Joh77a], which is developed in exercise 8.1.

8.5 Additional notes and references

The content in this chapter overlaps with [Eri19, chapter 9], [KT06, sections 6.8–6.9], [DPV08, chapter 4], [CLR+09, chapter 24], and [DD11, (video) lecture 19].

Spring 2026 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Handwritten notes annotated during the re-recording.
- Re-recorded video lecture.

8.6 Exercises

Exercise 8.1. Recall the $O(n^3)$ dynamic programming algorithm for APSP. If the edge lengths were nonnegative, then we could have just run Dijkstra's algorithm from every vertex in $O(mn + n^2 \log n)$ time, which is never worse than $O(n^3)$, and significantly better when the graph is sparse. Here we will develop a classic algorithm due to Johnson [Joh77b] that builds on this observation. (Obviously you can find this algorithm in any book, but the point is that you have the background to discover it yourself.)

Let $G = (V, E)$ be a directed graph with real-valued edge weights $\ell : E \to \mathbb{R}$. For simplicity we assume there are no negative cycles in G. Let $d(s, t)$ denote the distance with respect to ℓ. We define *vertex potentials* as a set of values $\varphi : V \to \mathbb{R}$ for each vertex. For a set of vertex potentials φ, consider the modified edge weights $\ell_\varphi : E \to \mathbb{R}$ defined by

$$\ell_\varphi(u, v) = \varphi(u) + \ell(u, v) - \varphi(v).$$

Let d_φ denote distances with respect to ℓ_φ.

1. Prove that for any walk $w : s \rightsquigarrow t$,

$$\bar{\ell}_\varphi(w) = \varphi(s) + \bar{\ell}(w) - \varphi(t).$$

Since all (s, t)-walks shift by the same quantity $\varphi(s) - \varphi(t)$, the shortest paths stay the same. We also have $d_\varphi(s, t) = \varphi(s) + d(s, t) - \varphi(t)$ for all s, t.

2. Consider the potentials defined by $\varphi(v) = \min_{s \in V} d(s, v)$. Prove that for these potentials, ℓ_φ is nonnegative.

3. Clearly the potentials above can be computed in $O(mn^2)$ time by running Bellman-Ford for every choice of source s. Design and analyze an algorithm computing φ in $O(mn)$ time.

4. Finally, put everything together to obtain Johnson's $O(mn + n^2 \log n)$-time algorithm for APSP.

Exercise 8.2. Recall that in foreign exchange, different currencies can be exchanged at various exchange rates. For example, at the time of writing, one can exchange 1 US dollar for about 0.89 Euro's. *Currency arbitrage* occurs when you start with some quantity of one currency, and make a sequence of exchanges through different currencies and end up with even more of the original currency than you started with.

Suppose we have n countries and for every (ordered) pair of countries X and Y we have an exchange rate $r_{X,Y}$; meaning, one unit of currency X can be exchanged for $r_{X,Y}$ units of currency Y. Consider the problem of identifying currency arbitrage starting from a fixed currency X, or deciding that no arbitrage is possible. Design and analyze an algorithm for this problem.

Exercise 8.3. The following problem (and more elaborate extensions) appear in reinforcement learning. Let $G = (V, E)$ be a directed graph and let the edges be annotated by positive edge weights $r : E \to \mathbb{R}_{>0}$. We think of the vertices as states of some device under our control, and the edge weights $r(e)$ as rewards obtained by traversing the edge e as follows. Let $s \in V$ be a fixed starting vertex / state.

1. Given an integer $k \leq n$, the goal is to compute a walk of length at most k maximizing the sum of rewards along that walk. (If you repeat an edge, you get the same reward each time you repeat the edge.) Design and analyze an algorithm for this problem.

2. Here we also incorporate a *discount rate*. Let $\alpha \in (0, 1)$ be given. Given a walk with edges $e_1, \ldots, e_k$, the *discounted total reward* of the walk is given by

$$r(e_1) + \alpha r(e_2) + \cdots + \alpha^{k-1} r(e_k).$$

   The idea is that if we think of each edge traversal as also taking a unit of time, then the rewards attained far off in the future are perceived to be worth less than the rewards attained now and in the short term.

Given an integer $k \leq n$, the goal is to compute a walk of length at most k maximizing the discounted total reward of the walk. Design and analyze an algorithm for this problem.

Chapter 9

SAT: a different kind of search problem

9.1 Boolean formulas

A Boolean variable x takes one of two values, `true` or `false`. In computers, the Boolean values are represented by a single bit, with `true` $= 1$ and `false` $= 0$. Boolean algebra has three basic operations.

1. *Negation*, also called a *not*, is a unary operator denoted $\neg x$ for a variable x. $\neg$ maps `true` to `false` and `false` to `true`. We sometimes abbreviate $\bar{x} \stackrel{\text{def}}{=} \neg x$.

2. A *disjunction*, also called an *or*, is a binary operator denoted $x \vee y$ for two variables x and y. We have $x \vee y =$ `true` if either x or y is `true`. If both x and y are `false`, then $x \vee y =$ `false`.

3. A *conjunction*, also called an *and*, is a binary operator denoted $x \wedge y$. We have $x \wedge y =$ `true` if and only if both x and y are `true`; otherwise, $x \wedge y =$ `false`.

These operations can be combined, using parentheses to notate the order of operations. For example, the formula

$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

implements what is commonly called the "exclusive-or" of x_1 and x_2: $f(x_1, x_2) = 1$ iff $x_1 \neq x_2$.

Observe that for any $x, y, z \in \{0, 1\}$,

$$(x \vee y) \vee z = x \vee (y \vee z);$$

that is, the placement of parentheses does not matter. This is called the *associative property*, and for this reason we may simply denote the above as

$$x \vee y \vee z$$

without introducing any ambiguity. Conjunction $\wedge$ is also associative. However, mixing $\wedge$ and $\vee$ is not associative: in general,

$$(x \vee y) \wedge z \neq x \vee (y \wedge z).$$

(Exercise 9.1 asks for a counterexample.) Instead we have the following rules, called the *distributive property* of $\vee$ and $\wedge$.

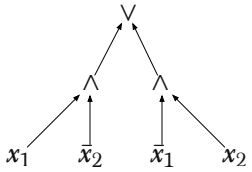
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z),$$
$$(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z).$$

$\neg$ interacts with $\wedge$ and $\vee$ by the following identity, called *De Morgan's laws*.

$$\neg(x \wedge y) = \bar{x} \vee \bar{y},$$
$$\neg(x \vee y) = \bar{x} \wedge \bar{y}.$$

We leave it to the reader to verify these identities. Lastly we point out that disjunction and conjunction are *commutative*: $x \wedge y = y \wedge x$, and $x \vee y = y \vee x$.

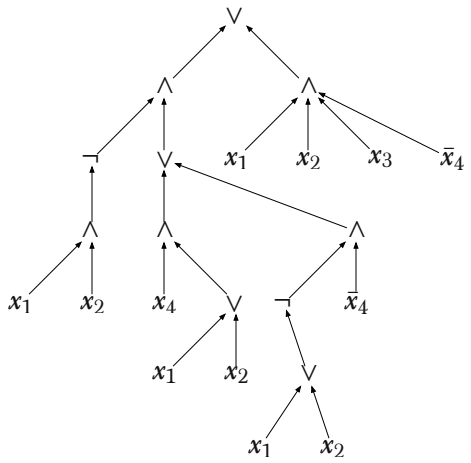
Boolean formulas have nested parentheses. The part of a formula corresponding to one pair of matching parentheses is called a *clause*. Nested parantheses encode a hierarchy of clauses, which can be visualized in a rooted tree. The following is a tree for exclusive-or:



As a perhaps more interesting example, the formula

$$f(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4) \\ \vee (\neg(x_1 \wedge x_2) \wedge ((x_1 \vee x_2) \wedge x_4) \vee (\neg(x_1 \vee x_2) \wedge \bar{x}_4))$$

simulates the addition of two bits: it is satisfied iff $x_1 + x_2 = x_3 x_4$, where $x_3 x_4$ is interpreted as a two-digit binary number. The corresponding tree is as follows.



Both the algebraic and tree-like representations of Boolean formulas emphasize a digital quality of Boolean formulas. Boolean logic is exactly the kind of thing that computers are good at. One might try to design a *search* algorithm that can solve Boolean formulas, in the following sense:

> Given a Boolean formula $f(x_1, \ldots, x_n)$, find an assignment $x_1, \ldots, x_n \in \{0, 1\}$ for which $f(x_1, \ldots, x_n) = 1$, if one exists.

One can also consider the *decision* version of the problem, as follows.

> Given a Boolean formula $f(x_1, \ldots, x_n)$, output `true` (or 1) if there exists an assignment $x_1, \ldots, x_n \in \{0, 1\}$ for which $f(x_1, \ldots, x_n) = 1$, and otherwise output `false`.

Clearly the first problem is at least as hard as the second since being able to find a solution implies you can decide if there exists a solution. But the converse also holds; an algorithm that can decide if a Boolean formula is satisfiable can be used to obtain an algorithm that *constructs* the satisfying assignment (see exercise 9.3). We refer to the above problems as that of *Boolean satisfiability*, abbreviated *SAT*. It is highly desirable to design a good algorithm for solving SAT.

To put an algorithmic discussion about satisfiability on firm ground, it is worth establishing the following parameters. Each Boolean formula $f(x_1, \ldots, x_n)$ has a well-defined number of variables, n. We also define the *size* as the number of symbols (parentheses, variables, or operators) required to write it out. Up to constant factors, this is the same as the total number of times that a variable appears in the formula, counted with repetition (and assuming no trivial redundancies). (Why?) We generally don't worry about constants so these two quantities are essentially equivalent. Note that the size of a formula (as defined above) is (basically) the input size from an algorithmic point of view.

A different kind of search problem. A fundamental feature of SAT is that, given a potentially satisfying assignment $x \in \{0,1\}^n$, it is very easy for us to verify that the answer is correct. We simply plug in the values and work through the Boolean algebra to see if the answer comes out as 0 or 1. (Here we would evaluate the innermost parentheses first, and work our way out. Or, in terms of the tree formulation, we work from the leaves up to the roots.) This can be done in linear time in the size of the formula. Now, because we can efficiently verify a solution, we have at least one algorithm at our disposal: brute force. We can enumerate all 2^n possible assignments to $x \in \{0,1\}^n$, and for each x, check if it satisfies the formula. This is not an efficient algorithm, but it is a correct algorithm nonetheless. Remarkably this is more or less the best we know how to do (in theory, in the worst case, etc.).

Zooming out, Boolean satisfiability is looking for an object that we can *recognize efficiently* when we see it. Any problem of this format is called a *search problem*. The over/under game is also a search problem. Whereas the over/under game is easy, it is challenging to come up with a polynomial time algorithm for Boolean satisfiability – even though both problems are of the same "search" metatype. Why is one so much harder than the other? Perhaps one reason is that Boolean satisfiability is so much more expressive.

For example, suppose we want to force two variables x and y to be equal. Then the following formula is true iff $x = y$, as the reader may verify:

$$(y \wedge x) \vee (\bar{y} \wedge \bar{x}).$$

Appending $\wedge((y \wedge x) \vee (\bar{y} \wedge \bar{x}))$ to an existing Boolean formula is like programming y to be the value of x. As another example, suppose we wanted to model the conditional statement, "if x then y else z". This is given by

$$(x \wedge y) \vee (\bar{x} \wedge z).$$

Through these examples, Boolean algebra starts to resemble a programming language.

The flip side of SAT being very expressive is that a polynomial time algorithm would be extremely valuable. We could effectively resolve any question that is logically posed! The consequences – practical, philosophical, and mathematical – would be profound; and alas, maybe too good to be true?

9.2 CNF

If Boolean formulas are too general to expect a good algorithm, then perhaps we can restrict ourselves to a special class of Boolean formulas that might be easier.

An important class of SAT formulas are those in *conjunctive normal form (CNF)*. These are the formulas that are a conjunction (i.e, an "and") of disjunctions ("or's"), possibly with negations on the variables inside the disjunctions. For example, the following formula is in CNF.

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) = &(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge \\ &(x_1 \vee x_4 \vee \bar{x}_5) \wedge (x_1 \vee \bar{x}_4 \vee x_5) \wedge \\ &(x_2 \vee \bar{x}_3 \vee x_4 \vee \bar{x}_5) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4 \vee x_5) \wedge \\ &(\bar{x}_2 \vee x_3 \vee x_4 \vee \bar{x}_5) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4 \vee x_5). \end{aligned}$$

One example of a satisfying solution is the bit string 01011. (Can you find another?)

The conjunctive normal form appears to clean up Boolean formulas quite a bit. As a tree, a formula in CNF would be almost flat, with one node representing the overarching conjunction, and a child for each disjunctive clause. But this is only a superficial observation. What we really want to know is whether conjunctive normal forms are meaningfully less expressive than Boolean formulas.

Above we showed how to implement assignment and conditional branching with Boolean formulas. As a test case, let us see if we can derive equivalent formulas in CNF.

1. *Assignment in CNF.* Suppose we wanted to model the constraint "$z = x$" in CNF. We have already observe that this is encoded by $(z \wedge x) \vee (\bar{z} \wedge \bar{x})$, but this is not a CNF. To this end,

$$\begin{aligned} (z \wedge x) \vee (\bar{z} \wedge \bar{x}) &\overset{\text{(a)}}{=} (z \vee (\bar{z} \wedge \bar{x})) \wedge (x \vee (\bar{z} \wedge \bar{x})) \\ &\overset{\text{(b)}}{=} (z \vee \bar{z}) \wedge (z \vee \bar{x}) \wedge (x \vee \bar{z}) \wedge (x \vee \bar{x}) \\ &\overset{\text{(c)}}{=} (z \vee \bar{x}) \wedge (x \vee \bar{z}); \end{aligned}$$

   a CNF, as desired. Here (a) and (b) are by the distributive property. (c) drops the tautologies $(x \vee \bar{x})$ and $(z \vee \bar{z})$.

2. *Conditional branching in CNF.* Suppose we want to model the Boolean statement "if x then y else z". We have

$$\begin{aligned} (x \wedge y) \vee (\bar{x} \wedge z) &\overset{\text{(d)}}{=} (x \vee \bar{x}) \wedge (y \vee \bar{x}) \wedge (x \vee z) \wedge (y \vee z) \\ &\overset{\text{(e)}}{=} (y \vee \bar{x}) \wedge (x \vee z) \wedge (y \vee z); \end{aligned}$$

   a CNF, as desired. Here (d) is by the distributive property and (e) drops the tautology $(x \vee \bar{x})$.

For at least the two cases above, CNF is just as expressive as general Boolean formulas. But these are just examples and do not constitute a complete proof. A much stronger theorem that truly establishes their (algorithmic) equivalence is as follows.

Theorem 9.1. *Given any Boolean formula $f(x_1, \dots, x_n)$ of n variables and size m, in polynomial time, one can construct a Boolean formula $g(x_1, \dots, x_n, x_{n+1}, \dots, x_p)$ with $p = O(m+n)$ variables and size $O(m+n)$ such that f is satisfiable iff g is satisfiable. Therefore, a polynomial time algorithm deciding if a Boolean formula in CNF is satisfiable implies a polynomial time algorithm for deciding if a (general) Boolean formula is satisfiable.*

The best way to see this is to prove it yourself. See exercise 9.5. (Yes, I believe you can do it.)

For (a wild and crazy) example, in the following formula, we convert the Boolean formula $f(x_1, x_2, x_3, x_4)$ for bitwise addition[1] into a CNF formula g with 15 variables.

$$g(x_1, \dots, x_{15}) = (\bar{x}_{14} \vee x_1) \wedge (\bar{x}_{14} \vee x_2) \wedge (x_{14} \vee \bar{x}_1 \vee \bar{x}_2) \wedge (\bar{x}_{15} \vee x_3) \wedge (\bar{x}_{15} \vee \bar{x}_4) \wedge (x_{15} \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_6 \vee x_1) \wedge (\bar{x}_6 \vee x_2) \wedge (x_6 \vee \bar{x}_1 \vee \bar{x}_2) \wedge (x_7 \vee \bar{x}_1) \wedge (x_7 \vee \bar{x}_2) \wedge (\bar{x}_7 \vee x_1 \vee x_2) \wedge (\bar{x}_8 \vee x_4) \wedge (\bar{x}_8 \vee x_7) \wedge (x_8 \vee \bar{x}_4 \vee \bar{x}_7) \wedge (x_9 \vee \bar{x}_1) \wedge (x_9 \vee \bar{x}_2) \wedge (\bar{x}_9 \vee x_1 \vee x_2) \wedge (\bar{x}_{10} \vee \bar{x}_9) \wedge (\bar{x}_{10} \vee \bar{x}_4) \wedge (x_{10} \vee x_9 \vee x_4) \wedge (x_{11} \vee \bar{x}_8) \wedge (x_{11} \vee \bar{x}_{10}) \wedge (\bar{x}_{11} \vee x_8 \vee x_{10}) \wedge (\bar{x}_{12} \vee \bar{x}_6) \wedge (\bar{x}_{12} \vee x_{11}) \wedge (x_{12} \vee x_6 \vee \bar{x}_{11}) \wedge (x_{13} \vee \bar{x}_5) \wedge (x_{13} \vee \bar{x}_{12}) \wedge (\bar{x}_{13} \vee x_5 \vee x_{12}).$$

The formula g above has the following guarantee: for $x_1, x_2, x_3, x_4 \in \{0, 1\}$, we have $f(x_1, x_2, x_3, x_4) = 1$ iff there exists $x_5, \dots, x_{15} \in \{0, 1\}$ such that $g(x_1, \dots, x_{15}) = 1$.

3SAT. An even more special case of CNF is where every clause has exactly k variables, for a fixed integer $k \in \mathbb{N}$; this problem is called k-SAT. Of particular interest is *3SAT*. One might hope for a better algorithm for 3SAT, but again we are thwarted by a polynomial time reduction.

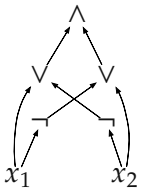
Theorem 9.2. *Given any Boolean formula $f(x_1, \dots, x_n)$ of n variables and size m, in polynomial time, one can construct a Boolean formula $g(x_1, \dots, x_n, x_{n+1}, \dots, x_p)$ with $p = O(m)$ variables and size $O(m+n)$ such that f is satisfiable iff g is satisfiable. In particular, a polynomial time algorithm deciding 3-SAT implies a polynomial time algorithm for SAT.*

The proof is left as exercise 9.6.

[1] $f(x_1, x_2, x_3, x_4) = \texttt{true}$ iff $x_1 + x_2 = x_3 x_4$.

9.3 Circuits

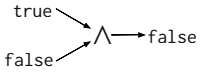
Boolean circuits are a generalization of Boolean formulas. Recall that a Boolean formula can be expressed as a rooted tree, where each internal node is associated with a logical operation that is applied to the Boolean outputs from its children in the subtree. Circuits are similar except they do not require the logic can be arranged more generally in a directed acyclic graph.



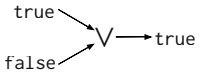
Likely the reader has encountered Boolean circuits before and we briefly review the components. The first circuit on the right describes XOR. The picture is almost self-explanatory. At a high level, Boolean circuits are composed of *logical gates* arranged (as nodes) in a *directed acyclic graph*. We first define the gates and explain their graphical arrangement afterwards.

Each gate corresponds to either one bit of input or one of the three Boolean operators. Each gate takes at most two input bits, and outputs one bit (which may be propagated to any number of other gates). The basic types of gates are:

1. An input gate, which models a single bit in the input, and outputs that bit.
2. An "and-gate" (or a "conjunction gate", or simple an "and" or a "conjunction"), takes as input the output bits from two other gates, and outputs the $\wedge$ of these two inputs.

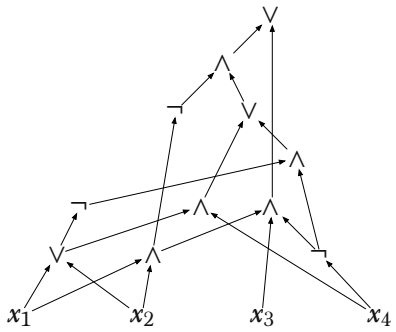


3. An "or-gate" (or a "disjunction gate", etc.) takes as input the output bits from two other gates, and outputs the $\vee$ of these two inputs.



4. A "not-gate"' (or a "negation gate", etc.) takes as input one output bit from another gate, and outputs the negation of that bit.

Physically, we imagine each circuit logically evaluating its inputs, producing a bit that is propagated to further circuits ahead. The circuit on the right simulates addition on two bits. The top gate evaluates to `true` iff $x_1 + x_2 = x_3 x_4$.



We can model the dependencies between gates by a directed graph, where each gate is a node, and we have a directed edge from one gate x to gate y when x is an input to y. The dependencies are required to be *acyclic*. Here we recall that a *cycle* in a directed graph is a sequence of edges



Figure 9.1: The addition circuit (cf. Fig. 14.2) redrawn to list the logical gates in topological order from left to right. Observe that all directed edges go from left to right.

$(x_1, x_2), (x_2, x_3), \ldots, (x_k, x_1)$ where the endpoint of one edge is the starting point of the next edge in the sequence, that starts and ends at the same vertex. A cycle would not be well-defined in a circuit: if gate x depends on y, and y depends on z, and z depends on x, then which gate is evaluated first? So to avoid these issues we disallow cyclic dependencies by definition.[2]

In the simplest model, we assume there is one gate in particular whose output we are externally interested in, as a function of the inputs; we call this the *output gate*. We will focus on a single output gate for simplicity; in general, one can have many output gates. Now we consider the problem of *circuit satisfiability*: given a Boolean circuit, is there an assignment to its input gates that makes the output gate `true`?

While a Boolean circuit is physically implementable, a first computational task would be able to simulate a Boolean circuit on a given input. A natural approach would be to try to simulate the physical process as follows.

> *Repeatedly, take any gate that has all of its input bits computed that itself has not been evaluated. Evaluate the inputs according to its logical operator, and write down that output.*

The algorithm will need to implement some bookkeeping to keep track of which gates have been evaluated, which of the unevaluated gates are ready to be evaluated, and so forth. It is fairly straightforward to sort out how to do this in linear time.

A more elegant approach applies *topological sort* to the underlying directed graph. If we topologically sort the gates of a circuit, the input gates will be towards the beginning of the list. In particular, the first gate has to be an input gate. The second gate has to be either (a) another input gate, or (b) a negation of the first gate. In general we might as well put all the input gates at the beginning of the list. Now, given the topological order, we can evaluate the gates in order: when evaluating a gate x, observe that all its inputs must be earlier in the topologically ordered list, hence already evaluated.

So much for evaluating a circuit. It is slightly more challenging to evaluate a circuit than a Boolean formula, but topological sort makes it much easier. Importantly, we have also established that circuit satisfiability is also a search problem: if, indeed, a

[2] One could consider evaluation *over time*, but this is a separate discussion.

circuit is satisfiable, then the satisfying inputs provide a proof that can be verified in polynomial time. Clearly circuit SAT is more general than Boolean satisfiability, so circuit SAT must be at least as hard boolean SAT. But is circuit SAT strictly *harder* than boolean SAT?

The reader might have realized, or at least sensed, that any circuit can be converted to a boolean SAT of roughly the same size. We leave it as exercise 9.7 to explain the construction explicitly. Such a construction implies the following theorem.

Theorem 9.3. *There is a polynomial time algorithm for boolean SAT iff there is a polynomial time algorithm for circuit SAT.*

9.4 Searching for answers

We have danced around but failed to resolve the following question.

> *Is there a polynomial time algorithm for deciding if a boolean formula is satisfiable?*

This is a huge open question, the most important question in theoretical computer science and arguably in all of mathematics. (Extra emphasis on *argue*.)

The general feeling is no, insofar as such an algorithm would be too good to be true. If the answer was yes, then we would have an efficient algorithm for *anything* that can be expressed in boolean logic. But we have already seen that boolean logic is very expressive — maybe too expressive to expect a computer to efficiently process.[3]

As daunting as the question posed above might seem, we have asked and resolved big questions before. Consider the following question posed by David Hilbert and Wilhelm Ackermann in 1928, called the *Entscheidungsproblem*. Paraphrasing, and omitting mathematical details, the question is roughly:

> *Can a computer compute (i.e., decide) any logical problem?*

For example, a computer can compute whether a boolean formula is satisfiable, by brute-forcing through all the possible solutions. (This is inefficient, but the point is that it works in a finite amount of time.) Is there anything a computer cannot compute, by brute-force or by other means? Remarkably this question has been answered. The

[3]This description oversimplifies the debate; there are other, more nuanced opinions and subtle related results.

answer to the *Entscheidungsproblem* is *no*, there are indeed *incomputable problems*, and it was proven independently by Church [Chu36] and Turing [Tur37]. We stress that the *Entscheidungsproblem* was posed *before* computers existed, purely in terms of mathematical logic. Church invented λ-calculus to lay the foundations for his proof; λ-calculus is the inspiration for `lisp` and functional programming languages since. A major contribution in Turing's approach was to define an abstract model of a computer (called a Turing machine), and then restate the *Entscheidungsproblem* in terms of this model. Turing's initial model for a computer (inspired by a typewriter) had a huge influence on modern computer design [Neu93] and is still the standard model for computation.

9.5 Additional results

While we do not have a polynomial time algorithm for SAT there are still many interesting developments that shed some insight into the character of the problem. We do not have the technical background to analyze the following topics at the moment but even at a high-level they are interesting to ponder.

9.5.1 Approximating 3SAT by randomized rounding

While 3SAT might be hard to solve exactly, it is not necessarily so hard to *approximately* solve it. Recall that in 3SAT we are given a formula in CNF with exactly three variables per clauses. Instead of trying to satisfy all the clauses, let us consider the optimization perspective where the goal is to satisfy as many clauses as possible. Of course being able to obtain the true maximum will also let you solve 3SAT. But maybe we can argue that we can obtain *relatively good* solutions. For example, is there a way to satisfy at least half as many clauses as the optimum assignment?

Let $f(x_1, \ldots, x_n)$ be a 3CNF, for which we want to satisfy as many clauses as possible in polynomial time. Imagine assigning every variable x_i *randomly* to `true` or `false`, by independently flipping a coin for every x_i. How good is this random (and completely ridiculous) assignment?

Consider a single disjunction clause; for example, consider $(x_1, \vee \bar{x}_2 \vee x_3)$. If $x_1 =$ `true`, or $x_2 =$ `false`, or $x_3 =$ `true`, then the clause is satisfied. In fact the only way the clause is *not* satisfied is if $x_1 =$ `false` *and* $x_2 =$ `true` and $x_3 =$ `false`. That exact combination occurs with probability 1/8, since each assignment is independent and uniformly random. So 7 out of 8 times the clause is satisfied!

How many clauses do we expect to satisfy? Think of the number of satisfied clauses as a *random sum*, to which each clause contributes 1 with probability 7/8, and contributes 0 otherwise. Now this random sum might seem tough to analyze because whether or not one clause is satisfied is *not* independent of whether another clause is satisfied, since they may share variables. Here we apply *linear of expectation: the average of the sum equals the sum of averages of the individual terms.* (We will introduce linear of expectation more slowly when discussing randomized algorithms in Chapter 25.) But the point is that each clause contributes 7/8th "satisfied clauses", on average; consequently, if there are m total clauses, we expect to satisfy $7m/8$ clauses. This implies, by the probabilistic method, that *every 3SAT formula (automatically) has an assignment satisfying* 7/8*-fraction of its clauses.*

It is easy to extend the above to k-SAT for any $k \in \mathbb{N}$, where each clause has exactly k variables, and obtain the following.

Theorem 9.4. *For all $k \in \mathbb{N}$, there is a randomized $(1 - 1/2^k)$-approximation algorithm for k-SAT.*

One can *derandomize* the algorithm above and obtain a deterministic algorithm. We let the reader attempt this before explaining.

Observe that it was very easy to *calculate the average* number of clauses that random coin tosses would satisfy: for 3-SAT, it was just (7/8) times the number of clauses. Even if it wasn't a k-SAT problem — if different clauses have a different number of variables — we could calculate the expected number of clauses satisfied. Indeed, a clauses with k variables is satisfied with probabilities $(1 - 2^{-k})$; the average sum is obtained as the sum of these probabilities.

We use these calculations to derandomize the coin tossing experiment. Consider x_1. We know that randomly flipping all the coins has a good average score, which we denote by μ. This means that either forcing $x_1 = \texttt{true}$ and randomly flipping the remaining $n - 1$ coins, or forcing $x_1 = \texttt{false}$ and randomly flipping the remaining $n - 1$ coins, has an average outcome of (at least) μ. At it is easy to calculate which choice of x_1 is better. We set x_2, x_3, and so on iteratively in this fashion. Each choice maintains the average over the remaining random coin flips. In conclusion we have the following.

Theorem 9.5. *For all $k \in \mathbb{N}$, there is a deterministic $(1 - 1/2^k)$-approximation algorithm for k-SAT.*

9.5.2 Hardness of approximation for 3SAT, and the PCP theorem

The above algorithm is so simple, and essentially oblivious to the input f, that it is hard to believe it is a very good algorithm. Remarkably, there is reason to believe that it is the best possible polynomial time algorithm unless $P = NP$. The *PCP theorem* states that for all constants $\epsilon > 0$, getting better than a $(7/8 + \epsilon)$-approximation to 3SAT is NP-Hard. The PCP theorem gives similar *hardness of approximation* results for many other problems besides SAT, and in general, has wide and deep consequences across theoretical computer science. The PCP theorem is far beyond the scope of this class although we did cover part of the proof in the randomized algorithms course [Qua].

9.6 Additional notes and references

SAT is discussed in [KT06, §8.2], [Eri19, §12.1–12.6], and [DPV08, §8.1], in a broader context (the theory of NP-completeness) that will emerge in upcoming chapters. The systematic study of understanding the limits of computation — of which SAT and circuits are central topics — is called *computational complexity theory*; see [Sip97; AB09; Wig19].

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

9.7 Exercises

Exercise 9.1. Show that $\wedge$ and $\vee$ are not jointly associative by finding assignments $x, y, z \in \{0, 1\}$ such that $(x \vee y) \wedge z \neq x \vee (y \wedge z)$.

Exercise 9.2. Recall that a *palindrome* is a string spelled the same forward or backwards, such as "racecar". Likewise we say that a bit string $x \in \{0,1\}^n$ if $x_i = x_{n+1-i}$ for all indices $i = 1, \ldots, n$. Describe a SAT formula in CNF of size $O(n)$ for a Boolean formula $f : \{0,1\}^n \to \{0,1\}$ that accepts palindromes (i.e., $f(x) = 1$ iff x is a palindrome).

Exercise 9.3. Suppose one had access to a polynomial time algorithm that solves the decision version of Boolean satisfiability (outputting `true` or `false` depending on whether there *exists* a satisfying assignment.). Show how to use this algorithm to obtain a polynomial time algorithm for the constructive version of Boolean satisfiability, outputting a satisfying assignment if one exists.

Exercise 9.4. Show that any Boolean formula with n variables and total size m can be converted to a Boolean formula with n variables and total size $O(m)$ such that every clause contains at most two sub-clauses or variables. That is, in the tree representation of the formula, each $\wedge$ and $\vee$ would have two children. For example, the formula for XOR, $f(x_1, x_2) = (\bar{x}_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2)$, is in this form.

Exercise 9.5. Show that any boolean formula with n variables and total size m can be converted to a boolean formula in CNF with $O(m + n)$ variables, $O(m)$ clauses, and total size $O(m)$.[4][5]

Exercise 9.6. Show that any CNF with n variables and total size m can be converted into a 3-SAT problem with $O(m)$ clauses and $O(m + n)$ variables.

[4] It might simplify things to first apply the preceding exercise.

[5] As a second (appropriately vague) hint, how do you express "$a = b \vee c$" and "$a = b \wedge c$" in CNF?

Exercise 9.7. Show that any boolean circuit with n gates and m edges can be converted into a boolean formula in CNF with $O(n)$ variables and total size $O(m)$.

Exercise 9.8. Let $k \geq 3$. Recall that the k-SAT problem is the special case of SAT where $f(x_1, \ldots, x_n)$ is a CNF with exactly k distinct variables per clause.

1. Show that a polynomial time algorithm for k-SAT implies a polynomial time algorithm for $(k+1)$-SAT.
2. Show that a polynomial time algorithm for $(k+1)$-SAT implies a polynomial time algorithm for k-sat.

Exercise 9.9. Consider the following special case of SAT, which we will call *k-occurrence-SAT* for a fixed parameter $k \in \mathbb{N}$. The input consists of a SAT formula $f(x_1, \ldots, x_n)$ in CNF such that every variable x_i appears (as is, or negated) in at most k clauses. The problem is to decide whether there is a satisfying assignment. For $k = 3$, either (a) design and analyze a polynomial time algorithm, or (b) show that a polynomial time algorithm for k-occurrence-SAT implies a polynomial time algorithm for (CNF-)SAT. [6]

Exercise 9.10. Consider the following variation of SAT, which we will call *opposite-SAT*. The input consists of a collection of m disjunction clauses over n variables $x_1, \ldots, x_n$. The goal is to decide if there is an assignment such that every disjunction evaluates to `false`. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) show that a polynomial time algorithm implies a polynomial time algorithm for SAT.

Exercise 9.11. In the *1-in-3-SAT* problem, one is given a 3-CNF $f(x_1, \ldots, x_n)$. The goal is to find a satisfying assignment such that every disjunctive clause (with three variables) is satisfied by *exactly* one of the variables.

Either (a) show that a polynomial time algorithm for 1-in-3-SAT implies a polynomial time algorithm for SAT, or (b) design and analyze a polynomial time algorithm for 1-in-3-SAT.

[6] As a warmup, it might be helpful to first consider the case $k = 5$. If you figure out 5-occurrence SAT, but don't figure out 3-occurrence SAT, we will give partial credit for a solution to 5-occurrence SAT.

Exercise 9.12. A Boolean formula is in *disjunctive normal form (DNF)* if it is a disjunction of conjunctions. For example, the formula

$$f(x, y, z) = (x \wedge y) \vee (\bar{x} \wedge z),$$

which models "if x then y else z", is in disjunctive normal form. Note that by the distributive property,

$$(a \vee b) \wedge (c \vee d) = (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d),$$

so any CNF formula can be converted to a DNF formula. Let us define *DNF-SAT* as the problem of deciding whether a boolean formula in DNF is satisfiable.

1. Is there a polynomial time algorithm for DNF SAT? If so, describe and analyze such an algorithm.
2. Would a polynomial time algorithm for DNF SAT imply a polynomial time algorithm for CNF SAT? Explain your reasoning.

Exercise 9.13. Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function on n variables. (Here no formula for f is given, and we treat f as a black box.) Prove that there is a CNF $\varphi(x_1, \ldots, x_n)$ of finite size such that $f = \varphi$.

Exercise 9.14. Let $k \in \mathbb{N}$ be fixed. Prove that for $n \geq k$, there exists a $O(n)$-size boolean formula for the n-variable function

$$f(x_1, \ldots, x_n) = \begin{cases} \texttt{true} & \text{if at least } k \text{ of the } x_i\text{'s are } \texttt{true} \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Exercise 9.15. Here we consider a special case of SAT that we call *ordered-k-overlapping-3-SAT* for a fixed parameter k. We take as input a 3-CNF formula $f(x_1, \ldots, x_n)$ with m clauses listed in order as to satisfy the following property. For every index i, there are at most k variables x_j with index $j < i$ that share a clause with the any variable x_j with index $j \geq i$. For this special case of 3-SAT, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

9.A Proof of Theorem 9.1

We left the proof of Theorem 9.1 as exercise 9.5 (which you should really try, if you haven't). However the result is so fundamental that we should provide a proof (or at least a proof sketch) anyway. The following also suggests some ideas that may help you prove Theorem 9.2 (which was left as exercise 9.6). (All these reductions are well-known and can also be found in standard textbooks.)

Here we describe a construction based on the "assignment" CNF gadget in the lecture notes. The idea is to prune all the clauses/subtrees by introducing new variables assigned to be equal to each clause. We hope the following construction is conceptually clear.

Let $f(x_1, \dots, x_n)$ be a Boolean formula with n variables and total size m. Without loss of generality (by the preceding exercise) we may assume that f is "binary", in the sense that every clause has at most 2 sub-clauses or variables.

For each clause C_i, we introduce a new variable y_i. The idea is to replace each clause C_i by assigning its output to y_i.

To this end, observe that the expression "$x_1 = x_2 \wedge x_3$" can be expressed as the CNF

$$(\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3), \tag{9.1}$$

and the expression "$x_1 = x_2 \vee x_3$" can be expressed as the CNF

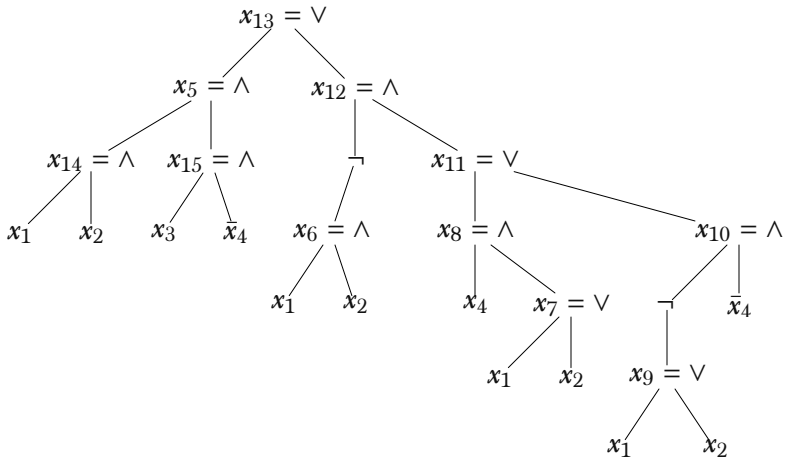
$$(x_1 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3). \tag{9.2}$$

We process all the clauses in the tree one by one starting from the bottom (and including the clause at the root). For each clause C_i, we replace C_i in the tree with a variable y_i. Because C_i is at the bottom of the tree, it is the conjunction or disjunction of at most two literals (involving either the x_i's or previously processed y_j's). We write the CNF corresponding to the assignment $y_i = C_i$ based on (9.1) or (9.2) or above. Note that this CNF has constant size.

All the CNF's we output are concatenated in an overarching conjunction. In addition we have one more clause containing the root variable, which means the whole formula must evaluate to true. We produce a constant size CNF per clause in f, so the overall size is the same up to constant factors. We introduce a variable for each clause which gives $O(m + n)$ variables in all.

Below, we apply the construction to the bit-addition formula from the lecture notes. We have annotated the nodes in the tree with the variables introduced an assigned to them. (Here, rather than use y_i's, we added x_j's with larger indices j.)

Below the tree, we list the CNF's that are produced, with one line per clause in the original formula, plus one more clause (just x_{13}) for the root. The final CNF is obtained by combining all the lines with conjunctions ("$\wedge$'s").



$$(\bar{x}_{14} \vee x_1) \wedge (\bar{x}_{14} \vee x_2) \wedge (x_{14} \vee \bar{x}_1 \vee \bar{x}_2)$$
$$(\bar{x}_{15} \vee x_3) \wedge (\bar{x}_{15} \vee \bar{x}_4) \wedge (x_{15} \vee \bar{x}_3 \vee x_4)$$
$$(\bar{x}_5 \vee x_{14}) \wedge (\bar{x}_5 \vee x_{15}) \wedge (x_5 \vee \bar{x}_{14} \vee \bar{x}_{15})$$
$$(\bar{x}_6 \vee x_1) \wedge (\bar{x}_6 \vee x_2) \wedge (x_6 \vee \bar{x}_1 \vee \bar{x}_2)$$
$$(x_7 \vee \bar{x}_1) \wedge (x_7 \vee \bar{x}_2) \wedge (\bar{x}_7 \vee x_1 \vee x_2)$$
$$(\bar{x}_8 \vee x_4) \wedge (\bar{x}_8 \vee x_7) \wedge (x_8 \vee \bar{x}_4 \vee \bar{x}_7)$$
$$(x_9 \vee \bar{x}_1) \wedge (x_9 \vee \bar{x}_2) \wedge (\bar{x}_9 \vee x_1 \vee x_2)$$
$$(\bar{x}_{10} \vee \bar{x}_9) \wedge (\bar{x}_{10} \vee \bar{x}_4) \wedge (x_{10} \vee x_9 \vee x_4)$$
$$(x_{11} \vee \bar{x}_8) \wedge (x_{11} \vee \bar{x}_{10}) \wedge (\bar{x}_{11} \vee x_8 \vee x_{10})$$
$$(\bar{x}_{12} \vee \bar{x}_6) \wedge (\bar{x}_{12} \vee x_{11}) \wedge (x_{12} \vee x_6 \vee \bar{x}_{11})$$
$$(x_{13} \vee \bar{x}_5) \wedge (x_{13} \vee \bar{x}_{12}) \wedge (\bar{x}_{13} \vee x_5 \vee x_{12})$$
$$x_{13}$$

Chapter 10

2-SAT and Max 2-SAT

Chapter 9 introduced various forms of *satisfiability problems*, such as Boolean SAT, Circuit SAT, and CNF SAT. (CNF SAT is often just called *SAT*.) Each takes as input some kind of function $f(x_1, \ldots, x_n)$ of n boolean variables $x_1, \ldots, x_n$ and asks if there is an assignent $x_1, \ldots, x_n \in \{\texttt{true}, \texttt{false}\}$ that satisfies f; i.e., such that $f(x_1, \ldots, x_n) = \texttt{true}$. We were not able to obtain polynomial time algorithms for any of these problems. However, we were able to derive polynomial time reductions between them. Consequently, there is a polynomial algorithm for any one of them iff there is a polynomial time algorithm for all of them.

The most restrictive version of SAT we discussed was *3-SAT*. This is where the boolean formula is expressed in *conjunctive normal form* with exactly three (distinct) literals per clause. Surprisingly, this restricted version turns out to be just as hard as boolean SAT and circuit SAT.

Here we go even further, to 2-SAT. *2-SAT* is the special case of (CNF-)SAT where each clause has exactly two literals. *Max 2-SAT* is like 2-SAT except rather than decide if the formula is satisfiable, the goal is to find an assignment of variables that satisfies as many clauses as possible. We are interested in the following questions:

1. Is there a polynomial time algorithm for 2-SAT, and/or does a polynomial time algorithm for 2-SAT imply a polynomial time algorithm for (CNF-)SAT?

2. Is there a polynomial time algorithm for Max 2-SAT, and/or does a polynomial time algorithm for Max 2-SAT imply a polynomial time algorithm for SAT?

10.1 2-SAT

We first consider 2-SAT. Formally, the input consists of a function $f(x_1, \ldots, x_n)$ in CNF form and exactly two variables per clause. For example,

$$f(x_1, x_2) = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2).$$

The goal is to find a satisfying assignment for f, or declare that none exists.

We want to either (a) design a polynomial time algorithm for this problem, or (b) relate 2-SAT to a problem we already believe is hard, like 3-SAT, via a polynomial time reduction.

2-SAT, unlike 3-SAT, has the following "forcing" characteristic. Consider the example above, and in particular the first clause,

$$(x_1 \vee x_2).$$

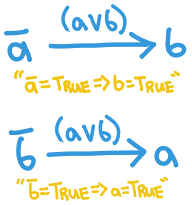
To satisfy f, either x_1 or x_2 have to be true. If we set $x_1 = \texttt{false}$, then we have to set $x_2 = \texttt{true}$. If we set $x_2 = \texttt{false}$, then we have to set $x_1 = \texttt{true}$.

For the formula $f(x_1, x_2) = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$, we can iterate this logic as follows.

Suppose $x_1 = \texttt{false}$. Then the first clause, $(x_1 \vee x_2)$, requires $x_2 = \texttt{true}$. But if $x_2 = \texttt{true}$, then the third clause, $(x_1 \vee \bar{x}_2)$, requires $x_1 = \texttt{true}$. So it is impossible to satisfy f if $x_1 = \texttt{false}$.

Similarly, if $x_1 = \texttt{true}$, then the second clause forces $x_2 = \texttt{true}$. If $x_2 = \texttt{true}$, then the last clause forces $x_1 = \texttt{false}$. So it is impossible to satisfy all clauses if $x_1 = \texttt{false}$. We conclude that f is unsatisfiable.

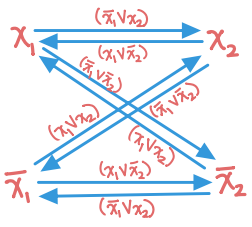
We encode these "implications" in the following *implication graph*.

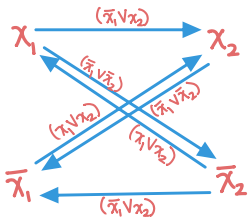


- For each variable x_i, we create two vertices for the two literals x_i and $\bar{x}_i$.
- For each clause $(a \vee b)$, where a and b are literals (such as x_1 or $\bar{x}_2$), we add arcs $(\bar{a}, b)$ and $(\bar{b}, a)$.

We call this graph the *implication* graph for the following reason: for any satisfying assignment, and any arc (a, b), if $a = \texttt{true}$, then $b = \texttt{true}$.

On the right, we draw the implication graph for the unsatisfiable formula $f(x_1, x_2) = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$. Each arc is labeled by the clause that produced it. Observe that x_1 and $\bar{x}_1$ both lie in the same directed cycle $(x_1, x_2, \bar{x}_1, \bar{x}_2, x_1)$. This cycle is a certificate that it is impossible to satisfy f.



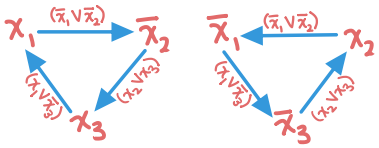


Next we draw the implication graph for the formula $f(x_1, x_2) = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$. (This formula omits the third clause from the previous example.) Observe that this time, there is no directed cycle containing both literals x_i and $\bar{x}_i$ for the same variable x_i. f is also satisfiable, with a satisfying assignment given by $x_1 = \texttt{false}$ and $x_2 = \texttt{true}$.

Is it always the case that the formula with no "bad" cycle in the implication graph is satisfiable? We claim it is necessary. That is, we claim that a 2-SAT formula is satisfiable iff for any variable x_i, the literals x_i and $\bar{x}_i$ are not strongly connected in the implication graph.

One direction is easier: if x_i and $\bar{x}_i$ are in the same directed cycle, then setting $x_i = \texttt{true}$ implies that $\bar{x}_i = \texttt{true}$ (i.e., $x_i = \texttt{false}$), and setting $\bar{x}_i = \texttt{true}$ implies that $x_i = \texttt{true}$. Since x_i is assigned exactly one of $\texttt{true}$ or $\texttt{false}$, we conclude that the formula is unsatisfiable.

Conversely, suppose that the literals x_i and $\bar{x}_i$ are not strongly connected for any variable x_i. We describe an algorithm that constructs a satisfying assignment. The algorithm is based on strongly connected components and leverages a certain *complementary* structure among the SCC's.



For a set of literals S, let $\bar{S} = \{\bar{u} : u \in S\}$ be the set of literals obtained by negating the variables in S. Observe that for every arc (u, v), we have the arc $(\bar{v}, \bar{u})$. Consequently, for every strongly connected component C, $\bar{C}$ is also a strongly connected component. We call $\bar{C}$ the complementary SCC of C.

Consider an SCC C of the implication graph. In a satisfying assignment, all literals in C must be set to $\texttt{true}$, or to $\texttt{false}$, simultaneously. Likewise, all literals in the complementary SCC $\bar{C}$ must have the same assignment, and the opposite assignment as (all the literals in) C.

We now describe the algorithm. Given a 2-SAT formula f, we construct the implication graph and compute the strongly connected components. If any strongly connected component contains both literals of the same variable, we return $\texttt{false}$. Otherwise we need to construct a satisfying assignment.



We process the SCCs in reverse topological order. The first component C is a sink component. We mark all literals in C as $\texttt{true}$ and mark all the literals in $\bar{C}$ as $\texttt{false}$. For each subsequent component C in reverse topological order, if the literals in C are not yet marked, we mark all the literals in C as $\texttt{true}$, and all the literals in $\bar{C}$ as $\texttt{false}$.

`2-SAT-solver(`2-SAT formula $f(x_1, \dots, x_n)$`)`

1. Construct the implication graph for f.
2. Compute the strongly connected components (SCCs) of the implication graph.
3. If some SCC C contains both literals x_i and $\bar{x}_i$ of some variable x_i, then return C as a certificate that f is unsatisfiable.
4. For each SCC C in reverse topological order:
   A. If C is unassigned, then assign (all literals in) C to `true` and $\bar{C}$ to `false`.
5. Return the assignment to $x_1, \dots, x_n$ computed above.

Figure 10.1: Solving 2-SAT in linear time via strongly connected components.

We need to argue that this produces a satisfying assignment. We say that a `true`/`false` labeling of the SCCs is *valid* if for every arc (C, D) between SCCs C and D, if $C =$ `true`, then $D =$ `true`. A valid labeling of the SCCs implies a satisfying assignment to the SAT formula.

Suppose we label an SCC C as `true` and $\bar{C}$ as `false`. We want to show that the label is valid for all the arcs leaving C and all the arcs entering $\bar{C}$.

Consider an arc (C, D). D succeeds C in the topological order, so it is labeled before C. Moreover, the arc (C, D) implies that D was not a source among the unmarked SCCs when it was labeled. Thus D must have been a sink, and labeled `true`, as desired.

Now consider an arc $(B, \bar{C})$. By complementarity, $(C, \bar{B})$ is also an arc. As argued above, $\bar{B}$ is labeled `true`, hence B is labeled `false`, as desired.

As for the running time, building the graph and computing the SCCs takes $O(m + n)$ time. The iterative marking process marks each vertex once, and takes $O(n)$ time. So the algorithm takes $O(m + n)$ time overall.

Altogether we have established the following.

Theorem 10.1. *Given a 2-SAT instance $f(x_1, \dots, x_n)$ with n variables and m clauses, in $O(m + n)$ time, once can compute either a satisfying assignment or a certificate of infeasibility.*

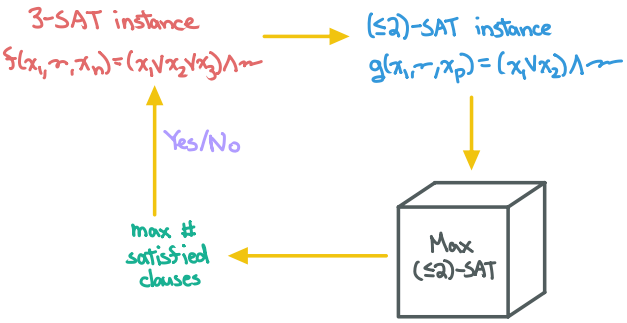
10.2 Max 2-SAT

Thus 2-SAT is polynomial time solvable. The next problem to discuss is max 2-SAT. Recall that max 2-SAT is like 2-SAT, except the goal is to satisfy the maximum number of clauses. In particular the input formula need not be satisfiable.

Of course max 2-SAT solves 2-SAT, while 2-SAT does not solve max 2-SAT. So the polynomial time algorithm for 2-SAT does not imply a polynomial time algorithm for max 2-SAT.

Max ($\leq$ 2)-SAT. It is easier to first consider a closely related problem we call *max at-most-2-SAT*, or max($\leq$ 2)-*SAT*. The problem is the same as max 2-SAT except each clause has *at most* 2 literals; in particular, we may have singleton clauses.

Theorem 10.2. *A polynomial time algorithm for max* ($\leq$ 2)*-SAT implies a polynomial time algorithm for SAT.*



We reduce from 3-SAT. The setup is as follows. We take as input a formula $f(x_1, \dots, x_n)$ in CNF with exactly three literals per clause. We want to decide if f is satisfiable.

We assume access to a black box that can solve max ($\leq$ 2)-SAT. To leverage this black box, we want to produce a formula $g(y_1, \dots, y_p)$ in CNF with at most 2 literals per clause, such that the maximum number of satisfying clauses in g can somehow tell us if f is satisfiable or not.

The main challenge is converting a 3-literal disjunction to clauses with only 1 or 2 variables. Note that our "assignment" trick from Chapter 9 does not work as that gadget produces disjunctions with 3 variables. We apply a different gadget, described in the following lemma.

Lemma 10.3. *Given a disjunction of 3 literals,* $C = (a \vee b \vee c)$*, one can compute a family* $\mathcal{F}_C$ *of 10 disjunctive clauses over* a, b, c *and an auxiliary variable* y*, each with at most 2 literals, such that for any assignment to* a, b, c*:*

(a) At most 7 clauses in $\mathcal{F}_C$ *can be satisfied.*
(b) If $(a \vee b \vee c) =$ `true`*, then there is an assignment* $y \in \{$`true`, `false`$\}$ *that satisfies 7 of the clauses in* $\mathcal{F}_C$*.*
(c) If $(a \vee b \vee c) =$ `false`*, then any choice of* $y \in \{$`true`, `false`$\}$ *satisfies at most 6 of the clauses in* $\mathcal{F}_C$*.*

Proof. Suppose we have a clause $(a \vee b \vee c)$, where a, b, c are literals. Let $y \in \{$`true`, `false`$\}$ be a new variable, and the following 10 clauses:

$$(a),\ (b),\ (c),\ (\bar{a} \vee \bar{b}),\ (\bar{b} \vee \bar{c}),\ (\bar{c} \vee \bar{a}),\ (y),\ (a \vee \bar{y}),\ (b \vee \bar{y}),\ (c \vee \bar{y})$$

$(\mathbf{a}, \mathbf{b}, \mathbf{c}) =$	$(\mathtt{f}, \mathtt{f}, \mathtt{f})$	$(\mathtt{t}, \mathtt{f}, \mathtt{f})$	$(\mathtt{t}, \mathtt{t}, \mathtt{f})$	$(\mathtt{t}, \mathtt{t}, \mathtt{t})$
$(a), (b), (c)$	0	1	2	3
$(\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a})$	3	3	2	0
$\max_{y \in \{\mathtt{t},\mathtt{f}\}} (y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$	3	3	3	4
total	6	7	7	7

Figure 10.2: Number of satisfied gadget clauses for $(a \vee b \vee c)$. The columns are based on the number of literals among a, b, c that are satisfied.

An assignment satisfies $(a \vee b \vee c)$ iff there is some choice of $y \in \{\mathtt{true}, \mathtt{false}\}$ that 7 of the 10 clauses in the gadget for $(a \vee b \vee c)$. If the assignment does not satisfy $(a \vee b \vee c)$ then any choice of y leads to satisfying at most 6 of the clauses. See Fig. 10.2 above. □

We apply this to the 3-CNF formula f as follows. Suppose we have m clauses $C_1, \ldots, C_m$, For each i, we apply Lemma 10.3 to C_i. Let $\mathcal{F}_i$ denote the corresponding set of 10 disjunctions, and let y_i be the new auxiliary variable, produced by Lemma 10.3 when applied to C_i. Finally, let $g(x_1, \ldots, x_n, y_1, \ldots, y_m)$ be the formula taking the conjunction of all the clauses in all the $\mathcal{F}_i$'s. g is in CNF, and has $m+n$ variables, $10m$ clauses, and at most 2 literals in every clause clause. We claim that f is satisfiable iff one can satisfy at least $7m$ clauses of g.

If f has a satisfying assignment $x_1, \ldots, x_n$, then we can choose $y_1, \ldots, y_m$ so that the combined assignment $(x_1, \ldots, x_n, y_1, \ldots, y_m)$ satisfies 7 clauses in each $\mathcal{F}_i$. Summing over $i \in [m]$ gives $7m$ satisfied clauses, total.

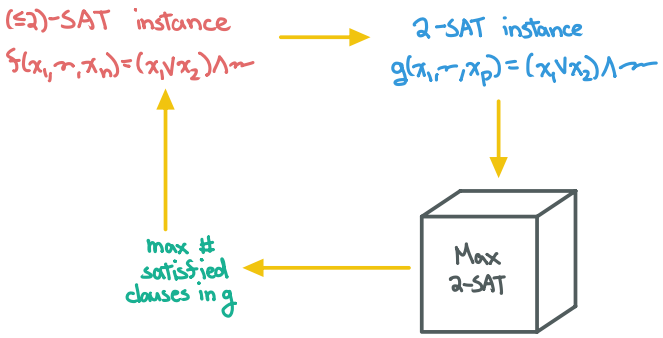
Conversely suppose f does not have a satisfying assignment. Consider any assignment $x_1, \ldots, x_n$. $x_1, \ldots, x_n$ does not satisfy some clause C_i, and consequently, any assignment of y_i leads to at most 6 clauses in $\mathcal{F}_i$ being satisfied. Since we can satisfy at most 7 clauses in any other $\mathcal{F}_j$, we conclude that at most $7m - 1$ clauses can be satisfied in g.

To recap, given f, we construct the formula g by applying Lemma 10.3 to each clause of f. We apply a black box to g that computes the maximum number of satisfiable clauses.

From max ($\leq$ 2)-SAT to max 2-SAT. We now show that max 2-SAT is hard by reducing from max ($\leq$ 2)-SAT.

Theorem 10.4. *A polynomial time algorithm for max 2-SAT implies a polynomial time algorithm for SAT.*

We give a reduction from max (≤ 2)-SAT to max 2-SAT. The only difference between the two problems is that (≤ 2)-SAT allows for singleton clauses of the form (a), where a is a literal. We first describe a clause to transform a singleton clause to a constant number of 2-literal disjunctions.



Consider a singleton clause a. Let $y \in \{\texttt{true}, \texttt{false}\}$ be a new variable, and consider the clauses

$$(a \vee y) \text{ and } (a \vee \bar{y}).$$

If $a = \texttt{true}$, then both clauses are satisfied. If $a = \texttt{false}$, then any choice of $y \in \{\texttt{true}, \texttt{false}\}$ satisfies 1 of the 2 clauses.

Now suppose we have an instance of Max (≤ 2)-SAT. The input consists of a CNF formula $f(x_1, \ldots, x_n)$ with k singleton clauses $C_1, \ldots, C_k$ and ℓ clauses $D_1, \ldots, D_\ell$ with 2 clauses each. For each singleton clause $C_i = (a_i)$ we introduce a new variable y_i and create the 2 clauses

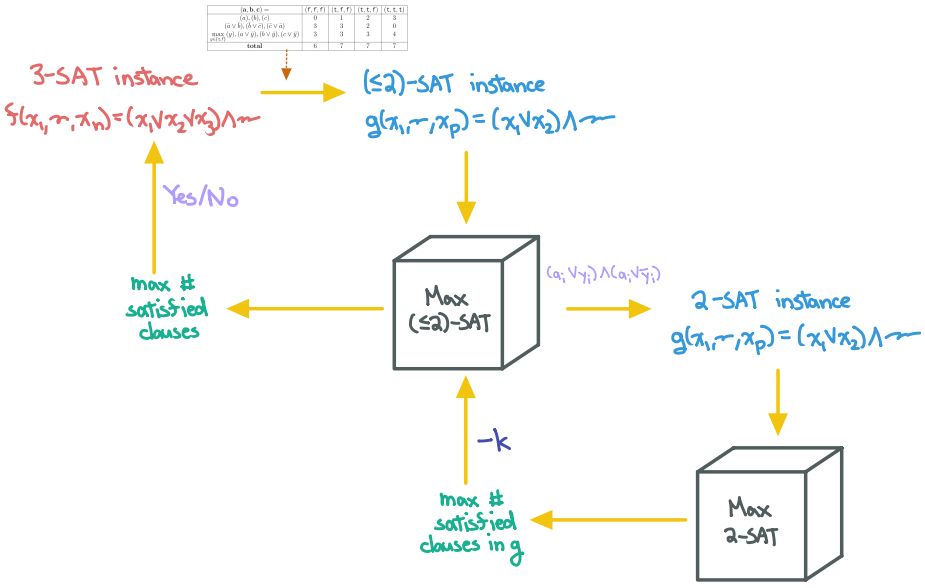
$$(a_i \vee y_i) \text{ and } (a_i \vee \bar{y}_i).$$

Consider the 2-sat formula $g(x_1, \ldots, x_n, y_1, \ldots, y_k)$ where we replace the k singleton clauses in f with the $2k$ auxiliary clauses above.

Given an assignment to $x_1, \ldots, x_n$ that satisfies p of the k singleton clauses, and q of the non-singleton clauses, any choice of $y_1, \ldots, y_k$ satisfies exactly $p + k$ of the auxiliary clauses replacing the singleton clauses. Thus the maximum number of satisfiable clauses in g is exactly k more than the maximum number of satisfiable clauses in f.

Given a black box for max 2-SAT, and a max (≤ 2)-SAT instance $f(x_1, \ldots, x_n)$, we construct the 2-SAT formula g as described above, pass it to the black box, and subtract k from the value returned by the black box.

This completes the proof of Theorem 10.4.



10.3 Additional notes and materials

Max 2-SAT was shown to be NP-Hard by [GJS76]. The linear time algorithm for 2-SAT is from [APT79]. Previously a polynomial time algorithm was given by [Kro67] and a different linear time algorithm was given by [EIS76].

Spring 2026 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

10.4 Exercises

Exercise 10.1. An undirected graph $G = (V, E)$ is *bipartite* if one can partition V into two sets V_1, V_2 such that every edge $e \in E$ has exactly one endpoint in V_1 and one endpoint in V_2. Consider the problem of deciding if an undirected graph G is bipartite. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 10.2. Given a 2-SAT formula, we say that a clause C is "pure" if either both literals are positive (e.g., $(x_1 \vee x_2)$) or both literals are negative. Otherwise we say C is "mixed". We say that a 2-SAT formula f is pure if every clause is pure.

For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. Pure 2-SAT: Given a pure 2-SAT formula decide if there is a satisfying assignment.

2. Max pure 2-SAT: Given a pure 2-SAT formula, compute the maximum number of clauses that can satisfied by any assignment.

Exercise 10.3. Recall that the exclusive-or (XOR) of two boolean variables x and y, denoted $x \otimes y$, is `true` iff exactly one of x or y is `true`.

In XOR-satisfiabilty, you are given a formula $f(x_1, \ldots, x_n)$ which is a conjuction of m (binary) XOR's from the n boolean variables and their negations; e.g.,

$$f(x_1, x_2, x_3, x_4) = (x_1 \otimes x_2) \wedge (x_2 \otimes \bar{x}_3) \wedge (x_3 \otimes x_4) \wedge (\bar{x}_1 \otimes \bar{x}_4);$$

and the goal is to compute a satisfying assignment for f. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 10.4. After your glorious app PikPok hit number 1 in the app store, you're preparing for version 2. Obviously, it needs to be great.

You've gathered a list of k features $F_1, ..., F_k$ that you could potentially add to version 2. However, there are complicated dependencies and requirements among them so you don't necessarily want to add all of them. There are 3 types of specifications defined over pairs of features F_i and F_j:

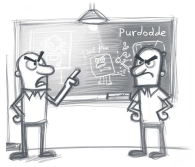
1. Requirements: Your app must include either F_i or F_j.
2. Conflicts: You cannot include both F_i and F_j.
3. Dependencies: If you include F_i, then you must include F_j.

Collectively, we call requirements, conflicts, and dependencies the *feature specifications*. The feature specifications are given in list form. The high-level task is to decide which of the features to implement, based on the given feature specifications. We have two versions of the problem:

1. In the idealistic feature selection problem, the task is to decide if there is a subset of features that satisfies all the feature specifications.
2. In the realistic feature selection problem, the task is to choose a subset of features that satisfies the maximum number of feature specifications.

For each of the problems above, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 10.5. As CEO of the fast growing tech company Purdoodle, you've decided to put together an internal conference on various topics in the field. There are m different topics $T_1, \ldots, T_m$, and n employees $E_1, \ldots, E_n$. Each employee has filled out a survey where they each select (exactly) 2 topics they are interested in and your goal is to assign each of them to a topic.



However, some pairs of employees are known to have had heated arguments and conflicts over particular topics. In these cases, you don't want to assign both of these employees to that same troublesome topic. The high-level goal is to assign employees to topics as to minimize, or even eliminate, these conflicts.

For the sake of concreteness, you can assume your input consists of:

- The list of survey responses of the form (E_j, T_h, T_i), meaning that employee E_j is interested in topics T_h and T_i. (Each E_j appears in one entry, and for each (E_j, T_h, T_i), T_h and T_i are different.)
- A list of p conflicts of the form (E_j, E_k, T_i), meaning that E_j and E_k have previously conflicted over topic T_i, so you don't want to assign both of them to topic T_i. You can assume that T_i was listed as a topic of interest by both E_j and E_k. (It is easy to filter out other conflicts in a preprocessing step.)

For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. Suppose you want to have zero conflicts. The goal is to assign each employee to 1 of their 2 chosen topics such that there are no conflicts in any of the topic sessions, or declare that it is impossible to schedule the conference without any conflict.

2. (Harder) If having no conflicts is unrealistic, then we can at least try to minimize the number of conflicts. The problem is to assign each employee to 1 of their 2 chosen topics as to minimize the total number of conflicts.

Exercise 10.6. Kenjaku has initiated the *Culling Games*, a brutal competition designed to enforce his twisted vision of survival of the strongest. In the game, there are m sorcerers and n "cursed spirits", where n is even. Kenjaku assigns a set of spirits to eliminate each sorcerer. A spirit may be assigned to multiple sorcerers.

Each sorcerer can only defend themselves against a limited number of spirits. (This number may vary by sorcerer.) The cunning Kenjaku strategically assigned exactly one more spirit to each sorcerer than they can handle.

As Satoru Gojo's successor, your goal is to save the overwhelmed sorcerers. To this end, you've partitioned the spirits into $n/2$ disjoint pairs, and you can destroy one spirit out of every pair. Your task is to choose one spirit from each pair to destroy so that each sorcerer can withstand the rest of the cursed spirits assigned to them.

Below are four different variations of the Culling Games varying in certain details. The input always consists of a list S_i of spirits assigned to each sorcerer i, where the length of a list is always one more than the sorcerer can handle, and the partition of spirits into $n/2$ pairs (as a list of pairs). For each, determine whether there exists a polynomial-time algorithm, or whether such an algorithm would imply a polynomial-time algorithm for (Boolean) SAT.[1]

1. In the most general setting, each sorcerer can handle some given number spirits, and this number varies by sorcerer. Determine whether there exists a choice of one cursed spirit to destroy from each pair such that every sorcerer can defend themselves against the rest of the spirits.

[1] I (Kent) have no idea what's going on here, and I have never felt so old. I refer you to the teaching assistants for more background.

2. Suppose each sorcerer can fend off at most 1 spirit. (So Kenjaku assigns two spirits to each sorcerer.) Again, determine whether there exists a choice of one cursed spirit to destroy from each pair such that every sorcerer can defend themselves against the rest of the spirits.

3. Suppose again that each sorcerer can fend off at most 1 spirit, but you realized you cannot save all the sorcerers. (E.g., via the previous problem.) Consider the problem of choosing one spirit to destroy from each pair to save as many sorcerers as possible.

4. Suppose again that each sorcerer can handle at most 1 spirit (and therefore has two spirits assigned to attack them), but also needs to fight a spirit to justify their life choices. Determine whether there exists a choice of one spirit to destroy per pair such that every sorcerer has exactly one spirit remaining to fight.

Exercise 10.7. Let $G = (V, E)$ be an undirected graph, with edge weights $w : E \to \mathbb{R}_{>0}$. ($G$ can have parallel edges.) A *cut* is a set of edges of the form

$$\delta(S) \stackrel{\text{def}}{=} \{\{u, v\} \in E : u \in S,\, v \notin S\}$$

for a set of vertices $S \subseteq V$. Note that $\delta(S) = \delta(V \setminus S)$. S and $V \setminus S$ are called *sides* of the cut. The *weight* of a cut $\delta(S)$ is the sum of weights over all edges $e \in \delta(S)$.

1. The weighted max cut problem is to compute the cut $\delta(S)$ of maximum total weight. The goal of this problem is to show that it is as hard as SAT to find a polynomial time algorithm, via a reduction from max 2-sat. The reduction is somewhat involved, so we will describe most of it and then ask you to fill in the remaining details and complete the proof.

   Suppose we have an instance of max 2-SAT defined by a boolean formula $f(x_1, \ldots, x_n)$ in CNF with exactly two variables per clause. Let $C_1, \ldots, C_m$ be the m clauses of f.

   We create a graph $G = (V, E)$ as follows. For the vertices V:

   - For each variable x_i, we have vertices representing the literals x_i and $\bar{x}_i$.
   - We also have two vertices representing `true` and `false`.

   For the edges E:

   - For each variable x_i, we add an edge between x_i and $\bar{x}_i$ with weight W, for some value $W > 0$ TBD.

- We also have an edge between `true` and `false` of weight W.
- For each clause C_i of the form $(a_i \vee b_i)$, where a_i and b_i are literals, we add the edges $\{a_i, b_i\}, \{a_i, \texttt{false}\}, \{b_i, \texttt{false}\}$, each with weight 1.

The high-level idea is that, for a sufficiently large value of W, any maximum weight cut $\delta(S)$ should contain all edges of weight W. In particular, (the vertices for) `true` and `false` will be on opposite sides of the cut, and the (vertices corresponding to the) literals x_i and $\bar{x}_i$ will be on opposite sides of the cut for each variable x_i. When a literal is on the same side as `true`, we interpret that as assigning the literal to be true; when a literal is on the same side of the cut as `false`, we interpret that as assigning the literal to be false.

Given the construction described above, it remains to choose W appropriately and prove that this gives a reduction from max 2-SAT to max-cut.

2. Now consider the unweighted version of the problem, where all edges have weight 1. Here G is allowed to have parallel edges. Either (a) design and analyze a polynomial time algorithm for unweighted max-cut, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT. For option (b), if you are building on / adjusting the construction from problem 1, you can explain the differences from that construction and argument, instead of rewriting the entire thing.

3. Next we consider the problem of deciding if $E = \delta(S)$ for some set $S \subseteq V$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

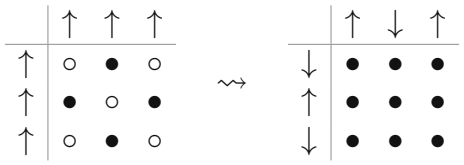
4. Finally, consider the problem of deciding if there is a cut of weight at least half of the total weight of all the edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 10.8. Let there be light. n^2 lightbulbs are arranged in an $n \times n$ grid. Each row has a switch and each column has a switch, for $2n$ switches in total. Each switch can be `up` or `down`. Flipping a row switch toggles every bulb in that row. Flipping a column switch toggles every bulb in that column.

The input is an $n \times n$ array A describing the state of each lightbulb (`on` or `off`) when every switch is `up`. (In general, some lightbulbs may be on, while others are off,

initially.) The goal is to decide if there is a configuration of switches that turns on all the lightbulbs simultaneously.

For example, consider the 3×3 instance below ($\bullet$ = `on`, $\circ$ = `off`), with row switches on the left and column switches on top. The configuration on the right turns on all the lightbulbs.



For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Chapter 11

Subset-Sum, and the secret language of numbers

Consider the following problem, called *subset sum*. You are given as input n natural numbers $x_1, \ldots, x_n \in \mathbb{N}$, and a target value $T \in \mathbb{N}$. The goal is to identify a subset of the numbers that sum to T:

$$x_{i_1} + x_{i_2} + \cdots + x_{i_k} = T,$$

for distinct indices $i_1, i_2, \ldots, i_k \in [n]$. We can also pose this as a decision problem where the goal is decide if there exists a subset that sums to T.

Subset sum is not an uncommon problem. We encounter it when we pay for groceries in cash and make exact change in quarters, dimes, nickels and pennies. It is also another example of a search problem. Given a candidate solution consisting of some numbers, we can easily verify that they are distinct numbers from the input, and that they add up to T.

11.1 Algorithms for subset sum

Let us design and analyze some algorithms for subset sum. We will focus on the decision version for simplicity but it will be easy to see how to modify our algorithms to return a feasible solution, if one exists.

A recursive algorithm. We will design a recursive algorithm. As always, we start with a recursive spec.

> `subset-sum(`$\{x_1, \ldots, x_n\}$`,`T`)` $=$ `true` if there is a subset of $\{x_1, \ldots, x_n\}$ that sums to T, and otherwise `false`.

We implement the recursive spec as follows.

`subset-sum`$(\{x_1, \ldots, x_n \in \mathbb{N}\},\ T)$

1. If $T < 0$ then return `false`.
2. If $T = 0$ then return `true`. *// The empty set sums to 0*
3. If $n = 0$ then return `false`. *// The empty set sums to 0*

/ Otherwise try either excluding or including x_1. */*

4. Return `subset-sum`$(x_2, \ldots, x_n,\ T) \vee$ `subset-sum`$(x_2, \ldots, x_n,\ T - x_1)$

It is easy to see the algorithm is correct by induction on n[1].

As per the running time, let $T(n)$ be the running time on a problem instance of n numbers. We have

$$T(0) = O(1)$$
$$T(n) = 2T(n-1) + O(1)$$

The recursion tree is a binary tree of height n, with 2^n leaves and $2^{n+1} - 1$ total nodes / subproblems. Each node / subproblem contributes $O(1)$ work, so we have

$$T(n) = O(2^n).$$

Dynamic programming. We can try to apply one of our favorite tricks: caching. In preparation for caching, let us index the arguments to our recursive function more compactly as follows. Fix an instance of subset sum, defined by $x_1, \ldots, x_n \in \mathbb{N}$ and $T \in \mathbb{N}$. For $i, S \in \mathbb{N}$, we define

> `subset-sum`(i, S) = `true` if there is a subset of $\{x_i, \ldots, x_n\}$ that sums to S, and `false` otherwise.

The following pseudocode implements `subset-sum` based on the new definition above, and is just a more compact version of our previous pseudocode.

`subset-sum`(i, S)

1. if $S = 0$ then return `true`
2. if $S < 0$ then return `false`
3. if $i > n$ then return `false`
4. return `subset-sum`$(i+1, S) \vee$ `subset-sum`$(i+1, S - x_1)$.

[1] More precisely, it is easy to see that the pseudocode implements the specification for a given problem *assuming the specification holds inductively* for all subproblems.

To solve our initial problem, we call `subset-sum(`1`,`T`)`.

After adding caching / dynamic programming to the pseudocode, each problem takes only constant time (excluding recursive calls). Thus the total running time becomes

$$(\#\text{ subproblems}) \cdot \left(\begin{array}{c} \text{time per subproblem} \\ \text{excl. recursive calls} \end{array} \right) = nT \cdot O(1) = O(nT).$$

This running time looks much better than the previous, exponential running time of $O(2^n)$. (More on this in a moment.)

Constructing a subset sum. Our algorithm above only decides if a subset sum solution exists. Of course we may also be interested in the numbers that make up the sum. One approach to extracting the numbers is to use the decision algorithm as a black box; see exercise 11.2. (SAT was similarly *self-reducible*). A more efficient approach is as follows.

In the algorithm above, each subproblem `subset-sum(`i`,`S`)` that returns `true` came to that conclusion based on some immediate choice - namely, whether or not to include x_i, which leads to one of two subproblems that also returns `true`. Let us also record, in an additional table, the choice of subproblem that leads to `true` (when that is the case). Then, after solving the decision problem and building up this table, we have a trail of of subproblems, starting from `subset-sum(`1`,`T`)` and ending at a base case that returns `true`, that lists the actual set of numbers that add up to T.

A polynomial running time? Recall that our first goal for any problem is try to establish a running time that is a polynomial function of the input size. Do we now have a polynomial time algorithm for subset sum? $O(nT)$ might look like a polynomial, but in fact it is not a polynomial in the input size.

The catch hearkens back to our very first discussion about bits: the *bit complexity* of T is only $\log(T)$. That is, we only need $\log(T)$ bits to express T. Thus $O(nT) = O\left(n2^{\log(T)}\right)$ is actually *exponential* in the input size of T. For example, if $T = 2^n$, then our $O(nT)$ running time is still exponential in n, while the input size of T is only $O(n)$. So for all the hoopla about dynamic programming, and the practical improvements for small T, we don't really have a polynomial time algorithm.

11.2 Packing in the complexity

Subset sum doesn't look complicated, but it has resisted our attempts to obtain a (truly) efficient algorithm. Let us instead see if we can find good evidence that subset sum is actually *hard*; that is, unlikely to have a polynomial time algorithm.

How can I convince you that subset sum is hard? After all, it is a very simple looking problem. Quite probably all instances you've encountered in real life (namely, making change for cash) were very easy. And *so what* if we're stumped? We have overcome problems before. Surely, all we need is a good attitude, a hot cup of coffee, and copious scratch paper!

Recall the SAT problem, where we are given a 3-SAT formula and we want to find a satisfying assignment. Now *there's* a problem. Since we can reduce any general Boolean formula to a 3-SAT formula of roughly the same size, solving 3-SAT is tantamount to solving *anything* that can be expressed in Boolean logic. There is no shame in admitting that 3-SAT is too hard to expect a polynomial time algorithm. So what if I told you that

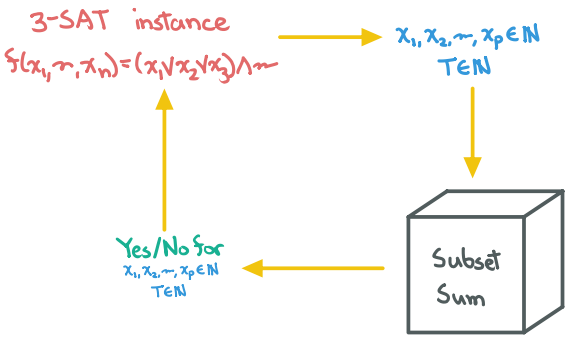
any algorithm for subset sum can be efficiently repurposed to solve 3-SAT.

That is, a polynomial time algorithm for subset-sum *would imply a polynomial time algorithm for 3-SAT*. That would be a satisfying explanation for our inability to find an efficient algorithm for subset sum: the problem is actually much more significant than we realized.

The main theorem of this section asserts precisely this, as follows.

Theorem 11.1. *If there is a polynomial time algorithm for subset sum, then there is a polynomial time algorithm for SAT.*

We have already seen that SAT and 3-SAT are polynomial time equivalent. To prove the theorem, we will describe a *polynomial time reduction* from 3-SAT to subset sum.



Given a 3-SAT formula $f(x_1,\dots,x_n)$, we will construct a subset sum problem that is feasible iff f is satisfiable. This might seem ridiculous at first. Subset sum uses only basic arithmetic, and arithmetic seems crude compared to Boolean logic.

The (simple yet profound) trick is to realize that *integers are bit strings too*: two numbers are equal iff their bit strings are equal. At a high level, we will create a subset sum problem where each number has a large number of digits. Each digit encode a different logical constraint. A feasible subset sum implies that every digit matches, which means that all logical constraints are met.

Consider an instance of 3-SAT given by a formula $f(x_1, \dots, x_n)$ with n variables and m disjunctive clauses, where each clause has three distinct variables, possibly negated. We let $C_1, \dots, C_m$ denote the m clauses, and treat them as subsets of the set of symbols $X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. For example, a clause $(x_1 \vee \bar{x}_2 \vee x_3)$ corresponds to the set $\{x_1, \bar{x}_2, x_3\}$. In this way, we can recast 3-SAT problem as a "subset" problem as follows:

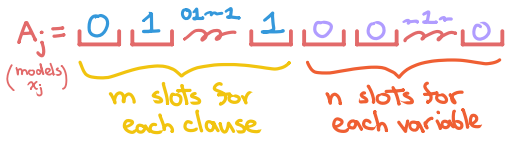
> *Find a set $S \subset X$ that satisfies the following constraints.*
>
> 1. *For each clause $C_i \subset X$, $C_i \cap S \neq \emptyset$.*
> 2. *For each Boolean variable x_j, S contains exactly one of x_j or $\bar{x}_j$.*

Above, the set $S \subset X$ indicates whether we assign `true` or `false` to each variable x_j. By item 2, for each j, S contains either x_j or $\bar{x}_j$. We interpret $x_j \in S$ as assigning $x_j =$ `true` and $\bar{x}_j \in S$ as assigning $x_j =$ `false`. With this setup, item 1 is really saying that (the assignment corresponding to) S has to satisfy every clause.

As mentioned above, we will use numbers with a large number of digits and use each digit to encode some logic. In particular, we will create one digit for each constraint on S listed above. We will have 1 digit for each clause C_i (corresponding to the constraints in item 1) and 1 digit for each variable x_j (corresponding to the constraints in item 2). Thus $m + n$ digits in all. We will also work with numbers in base 4 (without loss of generality). One can alternatively think of each $(m + n)$-digit, base-4 number as an $(m + n)$-length string composed of one of 4 letters; i.e., in $\{0, 1, 2, 3\}^{m+n}$.

We will create $2n + 2m$ numbers – one number for each x_j and each $\bar{x}_j$ in X, as well as $2m$ additional variables (two for each clause). For clarity, we will denote the numbers we create by capital letters $A, B, C, \dots$.

For each x_j, we will create a number A_j as follows. Recall that we have reserved a digit for each clause and a digit for each variable. We describe A_j in terms of the value of each digit (out of 4).



1. For each clause C_i, if $x_j \in C_i$, we set the C_ith digit (i.e., the digit corresponding to C_i) of A_j to 1. Otherwise we set that digit to 0.
2. We set the x_jth digit (corresponding to the variable x_j) to 1.
3. We set the remaining digits corresponding to other variables x_k $(k \neq j)$ to 0.

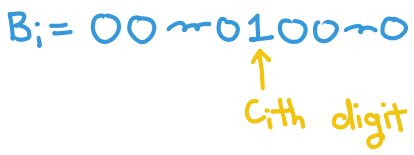
For each negated symbol $\bar{x}_j \in X$ we create a number $\bar{A}_j$ similarly and with the obvious adjustments:

1. For each clause C_i, if $\bar{x}_j \in C_i$, we set the C_ith digit of $\bar{A}_j$ to 1. Otherwise we set that digit to 0.
2. We set the x_jth digit to 1.
3. We set the remaining digits corresponding to other variables x_k $(k \neq j)$ to 0.

This creates the first $2n$ numbers and we have promised to create $2m$ more. To motivate these remaining variables let us review what we have already created. We claim that

> *A set $S \subset X$ is a satisfying assignment iff, letting*
>
> $$D = \sum_{x_j \in S} A_j + \sum_{\bar{x}_j \in S} \bar{A}_j$$
>
> *denote the sum of numbers corresponding to symbols in S, we have the following:*
>
> 1. *For every variable x_j, the x_jth digit of D is exactly 1.*
> 2. *For every clause C_i, the C_ith digit of D is at least 1.*

This is not quite a subset sum problem, but I think it is fair to say we are getting closer. We have two sets of constraints on D. The first family, corresponding to the variables x_j, is naturally suitable for subset sum, as it specifies *exactly* what the numbers have to add up to for each of each digits. However, the second set of constraints, for the digits corresponding to D, are based on inequalities rather than equalities. The issue is that, in 3-SAT, we could satisfy a clause C_i by taking one, two, or all three of the participating variables. Thus, for the corresponding digit, the values $\{1, 2, 3\}$ should all be allowed. How do we express the fact that we allow for a *range* of digits, in the native language of subset sum?



We introduce "padding": for each clause C_i, we create two copies of the same number, called B_i. B_i has value 1 in the C_ith digit and value 0 in all the other digits. The role of B_i is simply to allow us to artificially increase the C_ith digit of D. In particular, with two copies of B_i, we can add up to two units to the C_ith digit of D. Values of 1 or 2 can easily be converted to 3. Crucially, a value of 0 cannot be increased to 3.

We now select our target value T. For each clause C_i, we set the C_ith digit of T to 3. For each variable x_j, we set the x_jth digit of T to 1.



Observe that the size of our subset sum problem, $O\Big((m+n)^2\Big)$, is polynomial in the size of the 3SAT formula. It is also easy to see that our construction takes polynomial time (and maybe even linear time, with care).

In conclusion, our construction satisfies the following claim, which establishes Theorem 11.1.

> *The Boolean formula f is satisfiable iff there is a subset of the A_j's, $\bar{A}_j$'s, and (2 copies each of the) B_i's (as defined above) that adds up to T.*

11.3 Perfect is the enemy of good

Let us pause and reflect on our developments. We started with a simple and common problem: subset sum. We have some brute force algorithms, which when combined with caching (dynamic programming), leads to a reasonable running time for small target values. But this running time was not strongly polynomial; for large numbers, the algorithm can be very slow. The basic reason our algorithm did not qualify as polynomial time is that a very large integer x requires only $\lceil \log x \rceil$ many bits to represent it, and ultimately we seek running times that are polynomial in the input *size*. A $O(x)$ running time is *exponential* in the input size of x.

Our reduction from 3SAT to subset sum suggests that subset sum is much harder to solve then one might have expected. Moreover, we see explicitly in the reduction how the many bits of large numbers can be used to encode the rich complexity of a 3SAT problem.

The difficulty of subset sum lies in our insistence on an *exact* answer to the problem. By contrast, suppose we had an algorithm that satisfies the following approximation guarantee:

> *If the subset sum is feasible, then return a subset $x_{i_1}, \ldots, x_{i_k}$ whose sum lies in the range $[.99T, 1.01T]$. If the subset sum problem is infeasible, then either return a subset sum in the range $[.99T, 1.01T]$, or output that it is infeasible.*

For many real applications, where the sums represent a measurable quantity, being off by a little is not the end of the world. On the other hand, since an approximate sum in the above sense does not imply an exact solution, it does not risk deciding a SAT problem in disguise. This gives hope for a polynomial time approximation algorithm.

Indeed, there *are* polynomial time approximation algorithms that satisfy the above guarantee, even if we replace $[.99T, 1.01T]$ with $[(1-\epsilon)T, (1+\epsilon)T]$ for any fixed error parameter $\epsilon \in (0, 1)$. (See exercise 11.8.) We may discuss approximation algorithms later, but for the moment, it is good to be aware of their basic motivation.

The other side of the coin — where the hardness of subset sum is *useful* — is cryptography. Loosely speaking, we can interpret the solution to a subset sum problem as a password - it is hard to guess (as far as we know), but easy to verify. Many other problems have this characteristic (as we will see) but subset sum is particularly simple, which has some computational advantages. The Merkle-Hellman system, one of the first public-key cryptographic systems, is based on subset sum, and in particular relies on the assumption that subset sum is hard. Lattice based cryptography, a more modern cryptographic technique, is (at a high level) based on the difficulty of multi-dimensional extensions of subset sum. See (e.g.) the notes [Pei13] for more details.

11.4 Additional notes and references

This lecture overlaps with [Eri19, §3.8] and [KT06, §6.4 and §8.8].

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

11.5 Exercises

Exercise 11.1. Recall the *1-in-3-SAT* problem from exercise 9.11. Suppose instead that we had reduced from 1-in-3-SAT to subset sum. How might this have simplified our construction in Section 11.2?

Exercise 11.2. At the end of Section 11.1, we discussed how to use an additional table to also extract the subset sum solution. Here we develop an alternative approach that was briefly alluded to in Section 11.1. It is not as efficient as building a secondary table, but the technique is more general and of independent interest.

Suppose one had a black box algorithm for deciding subset sum problems. That is, given an instance of subset sum described by n integers $x_1, \ldots, x_n \in \mathbb{N}$ and an additional target value T, the algorithm returns `true` if there exists a feasible solution, and otherwise `false`. Show that one can use this decision algorithm as a subroutine to obtain an efficient constructive algorithm for subset sum.

Exercise 11.3. Consider the following variation of subset sum which allows for an additive error of 1. We'll call it *subset-sum*± 1. The input consists of n numbers $x_1, \ldots, x_n \in \mathbb{N}$ and a target $T \in \mathbb{N}$, like subset sum. The goal is to decide if there is a subset of $x_1, \ldots, x_n$ that sums to either $T - 1$, T, or $T + 1$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.4. Consider the following variation of subset-sum called *Kobe subset sum*. The input is similar to normal subset sum, with n numbers $x_1, \ldots, x_n \in \mathbb{N}$ and $T \in \mathbb{N}$, except we are promised that each number x_i lies in the range $\{8, \ldots, 24\}$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.5. Let $x_1, \ldots, x_n \in \mathbb{N}$. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.[2]

[2]You can use the solution of one subproblem to solve another, as long as there's no circular dependencies overall.

1. The *partition problem* asks if one can partition $x_1, \ldots, x_n$ into two parts such that the sums of each part are equal.
2. The 3-*partition problem* asks if one can partition $x_1, \ldots, x_n$ into 3 parts such that the sums of each part are all equal.
3. The *any-k-partition problem* asks one can partition $x_1, \ldots, x_n$ into k parts, for any integer $k \geq 2$, such that the sums of each part are all equal.
4. The *almost-partition* problem asks if one can partition $x_1, \ldots, x_n$ into two parts such that the two sums of each part differ by at most 1.
5. [3]Let n be even. The *perfect partition problem* asks if one can partition $x_1, \ldots, x_n$ into two parts such that
   (a) Each part has the same sum.
   (b) Each part contains the same number of x_i's.

Exercise 11.6. Let $A[1..m]$ be an array of integers, and let T be an additional number. We say that a sequence of numbers

$$y_1, y_2, \ldots, y_k$$

is *nondecreasing* if $y_1 \leq y_2 \leq \cdots \leq y_k$. The problem is to find the length of the longest nodecreasing subsequence of A where the numbers sum to T. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.7. Consider the following variation of subset sum. The input consists of n (positive) integers $x_1, \ldots, x_n \in \mathbb{N}$, an integer $T \in \mathbb{N}$, and an integer $k \in \{1, \ldots, n\}$. The problem is to decide if there is a subset of $x_1, \ldots, x_n$ of cardinality k that sums to T. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

[3]IMO, this one is the trickiest.

Exercise 11.8. (Approximating subset sum.) Let $\epsilon \in (0, 1)$ be fixed. Here we treat ϵ as a fixed constant (like $\epsilon = .1$, for 10% error); in particular, running times of the form $O\left(n^{O(1/\epsilon)}\right)$ count as a polynomial.

A $(1 \pm \epsilon)$-approximation algorithm for subset sum is one that (correctly) either:

1. Returns a subset whose sum lies in the range $[(1 - \epsilon)T, (1 + \epsilon)T]$.
2. Declares that there is no subset that sums to (exactly) T.

Note that such an algorithm does not solve the (exact) subset sum problem. (As usual, we assume the input consists of natural numbers.)

1. Suppose every input number x_i was "small", in the sense that $x_i \leq \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$-approximation algorithm for this setting.
2. Suppose every input number x_i was "big", in the sense that $x_i > \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$-approximation algorithm for this setting.
3. Now give a polynomial time $(1 \pm \epsilon)$-approximation algorithm for subset sum in the general setting (with both big and small inputs).

Exercise 11.9. According to a report from the World Shipping Council (WSC), 661 shipping containers (a.k.a. crates) were lost overboard in the year 2022. This is actually an improvement, as an average of 1,566 containers were lost at see each year in the fifteen year period from 2008 to 2022. (Cf. https://www.worldshipping.org/news/world-shipping-council-releases-containers-lost-at-sea-report-2023-update)



Now suppose you are running a crate shipping company. To try to minimize the number of crates that fall off the boat, you have decided to impose a strict weight limit $L \in \mathbb{N}$ on the total weight of shipping crates on any boat. Given a collection of n crates of varying weight $W_1, \ldots, W_n \in \mathbb{N}$, your goal is to ship all n crates with the minimum number of boats.

More formally, given the weight limit L and the crate weights $W_1, \ldots, W_n$, the goal is to assign each crate to one of $k \in \mathbb{N}$ boats, for k as small as possible, so that the total weight of containers assigned to any boat is at most L. (To be clear, k is not part of the input; your job includes finding the minimum possible number of boats k. You can also assume that $L \geq W_i$ for all i.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.10. This problem is about rectangular *tiling*, where the input consists of a collection of (smaller) rectangular pieces, and a (larger) rectangular container. The high-level goal is to try to "tile" the rectangular container with the smaller rectangular pieces.



More formally, you have n rectangular pieces $R_1, ..., R_n$ where each R_i is specified by a height H_i and a width W_i. (You can assume the input is given as two arrays $H[1..n]$ and $W[1..n]$). You are also given a rectangular container C specified by a height H_C and a width W_C. A "tiling" of C by $R_1, \ldots, R_n$ is a placement of (some or all of) the rectangular pieces into the container so that there are no overlaps or gaps.

(Like a jigsaw puzzle, or a tiled bathroom floor. To be clear, you can only use each piece once.) A "perfect tiling" of C by $R_1, \ldots, R_n$ is a tiling where all the rectangular pieces are used. For simplicity we assume that you cannot rotate the pieces.

1. (5 points) Consider the problem of computing a tiling of C by $R_1, \ldots, R_n$. (Not all the pieces have to be used.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. (5 points) Now consider the problem of computing a perfect tiling of C. (This time all the pieces have to be used.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.11. Here we consider a vast generalization of edit distance, as follows. Suppose the cost of an edit sequence is given by a nonnegative real-valued function $f(a, b, c) \geq 0$, where a is the number of insertions, b is the number of deletions, and c is the number of substitutions. We assume that the function is monotonically increasing in each of its arguments. That is, increasing a, b, or c increases the value of $f(a, b, c)$.

Assume that f is provided as a subroutine that takes $O(1)$ time to evaluate. Consider the problem the minimum cost edit sequence w/r/t the cost function f. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Sample solution.

Problem. Recall the subset sum problem, where the input consists of n positive integers $x_1, \ldots, x_n \in \mathbb{N}$, and a target value $T \in \mathbb{N}$. The goal is to decide if there is a subset of $x_1, \ldots, x_n$ that sums to T. Design and analyze a $O(nT)$-time algorithm for the subset-sum problem.

Solution.

1. Recursive spec / induction hypothesis. For all integers $S \leq T$ and $i \in \mathbb{Z}_{\geq 0}$, we define

> `SS(`i`,`S`)` $=$ `true` if there is a subset of $x_i, \ldots, x_n$ that sums to S, and `false` otherwise.

2. Solving the original problem. The solution to the original problem is given by `SS(1,`T`)`.

3. Recursive implementation. `SS(`i`,`S`)`

1. If $S = 0$ then return `true`.
2. If $S < 0$ then return `false`.
3. If $i \leq 0$ then return `false`.

`/*` *Otherwise* $S > 0$ *and* $i \leq n$. `*/`

4. Return the $\vee$ (or) of the following:
   A. `SS(`$i+1$`,`S`)`
   B. `SS(`$i+1$`,`$S - x_i$`)`.

4. Running time with caching / dynamic programming. After caching the (non-base case) solutions for the recursive implementation (i.e., applying dynamic programming), the running time is as follows. We have $O(nT)$ subproblems broken down as $O(n)$ choices for i and $O(T)$ choices for S, and each takes subproblem takes constant time (excluding recursive sub-calls). Overall the running time is $O(nT)$.

5. Proof. The base cases are pretty clear: if $S = 0$, then the empty set sum to S; otherwise, if $S < 0$ then no set of (positive) integers can sum to S; otherwise if $S > 0$ but $i > n$ then there are no numbers left to sum to S.
Consider `SS(`i`,`S`)` where $i \leq n$ and $S > 0$. If there is a feasible solution, then it either includes x_i or it doesn't. If it doesn't, then by induction on i, `SS(`$i+1$`,`S`)` will return `true`. If it does, then by induction on i, `SS(`$i+1$`,`$S - x_i$`)` will return `false`.
If there is no feasible solution, then there is no feasible solution to the subset sum problems described by `SS(`$i+1$`,`S`)` or `SS(`$i+1$`,`$S - x_i$`)`. By induction on i both sub-calls will return `false`, hence so will `SS(`i`,`S`)`, as desired.

Chapter 12

The longest path

Chapter 7 developed algorithms for computing shortest (s, t)-paths and walks in directed graphs. Here we consider the problem of finding the *longest* path in G. (Recall that vertices cannot repeat in a path, by definition.) One can extend the problem to edge and vertex weights though we focus on the unweighted setting for simplicity. The decision version of the problem includes in the input an integer k, and asks if there is a path with at least k vertices. (The search version reduces to the decision version: see exercise 12.1.) An important special case takes $k = n$: does the graph contain a path traversing all the vertices? Such a path is called a *Hamiltonian path*; the "Hamiltonian path problem" asks if G has a Hamiltonian path.

We will consider the longest path problem in two settings. First we will consider the special case of DAG's, which are easier to work with. Then we will consider general graphs.

12.1 The longest path in a DAG

Let $G = (V, E)$ be a directed acyclic graph (DAG). Our goal is to compute the longest path in G.

We encourage the reader to attempt the problem before proceeding.

The assumption that G has no cycles is very convenient because it implies that all walks in G are automatically paths. Indeed, vertices *cannot* repeat, since repetition implies a cycle. Our problem, then, is the same as finding the longest walk in G.

Let us develop a simple recursive algorithm, starting with a recursive spec. We want the length of the longest path/walk in G. (Getting the actual longest path itself follows easily.) Where do we start the path? Who knows – so let's throw the starting vertex in the recursive specification as a parameter. So we define:

> `longest-walk(`v`)` = the length of the longest walk starting from v (as measured by the number of edges).

Given `longest-walk(`v`)` as defined above, we will return the maximum of `longest-walk(`v`)` over all $v \in V$.

We implement `longest-walk`, as specified above, as follows.

$$\texttt{longest-walk}(v) = \begin{cases} 0 & \text{if } v \text{ is a sink} \\ 1 + \max_{(v,w) \in \delta^+(v)} \texttt{longest-walk}(w) & \text{otherwise.} \end{cases}$$

As with any recursive algorithm, we should first justify that this algorithm terminates in finite time. Here we invoke the assumption that G is a DAG. Suppose we start at a vertex v. The vertices associated with recursive sequence calls will form a walk from v, and since G is a DAG, the walk cannot circle back to v. Another way to see this is to sort the vertices in topological order, with sources towards the front and sinks towards the bottom of the list. Then all arcs are directed down the list, and all recursive calls from `longest-walk(`v`)` only go down the topological order/.

Now, directly running the recursive algorithm above has a worst case running time of $O(n!)$ (see exercise 12.2). But suppose we cache our answers and visit each vertex only once. Excluding recursive subcalls, the time to compute `longest-walk(`v`)` is proportional to the out-degree of v (denoted $d^+(v)$). Summing $d^+(v)$ over all v counts every edge exactly once, so the total running time is

$$O\left(\sum_{v \in V} 1 + d^+(v)\right) = O(m + n).$$

So the longest path problem can be solved in linear time in a DAG, by dynamic programming.

Of course this only gives the maximum length of any path. We can extract the actual path as follows. We can assume we've already computed `longest-walk(`v`)` for all v. Suppose we want to extract the longest path from a path v. Assuming v is not a sink, the longest path from v consists of v plus the longest path from the vertex w such that $(v, w) \in \delta^+(v)$ and `longest-walk(`v`)` = `longest-walk(`w`)` $+ 1$. We can used the stored `longest-walk(`w`)` values to identify w, and recursively obtain the rest of the path from v. (This technique is sometimes called "backtracking", and can be used in other dynamic programming algorithms to construct an optimal solution.)

Stepping back, we've uncovered an important algorithmic feature of DAGs: dynamic programming can be applied to DAG's, with the topological order guiding the induction.

12.2 The longest path in general graphs

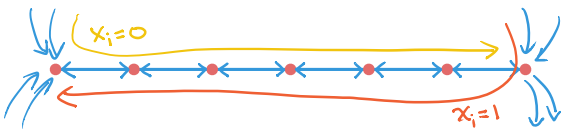
Having obtained a (rather simple) linear time algorithm for the longest path problem in DAG's, we consider the problem in a general directed graph G. In particular, G may have cycles. As remarked above, the recursive algorithm may not terminate when there are cycles. Can we somehow salvage the approach?

The following theorem indicates that the longest path problem, and even the special case of Hamiltonian path, is much harder than in DAG's.

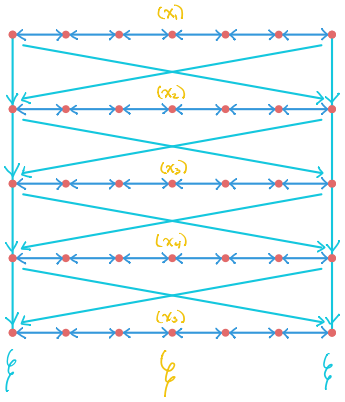
Theorem 12.1. *A polynomial time algorithm for the Hamiltonian path problem implies a polynomial time algorithm for SAT.*

The rest of this section is devoted to proving Theorem 12.1.

We take as input a SAT formula $f(x_1, \dots, x_n)$ in CNF with m clauses. Based on f, we construct a directed and unweighted graph G with M edges and N vertices, where M and N are TBD. It will have the feature that f is satisfiable iff G has a Hamiltonian path. We first sketch out the main ideas at a high level. More specific details (like exactly how many vertices of some type) will be supplied at the end once the high level ideas are in place.

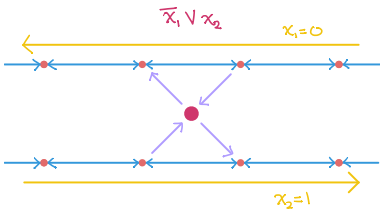


Speaking informally, for each variable x_j, we want to encode the choice of x_j (whether $x_j = \texttt{true}$ or $x_j = \texttt{false}$) as a graph. One way to do this is via a long, bidirected path for each x_j. A Hamiltonian path can only traverse the path in one of the two directions. So we can interpret one direction as setting $x_j = \texttt{true}$ and the other as setting $x_j = \texttt{false}$.



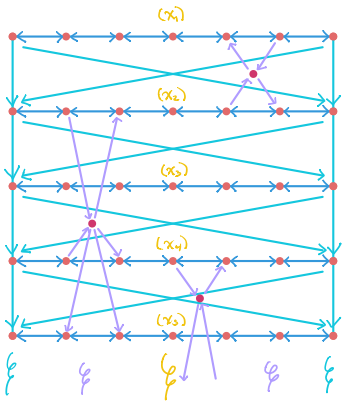
We arrange these paths, one for each x_i, in a "ladder"-like structure, strung together by edges connecting the endpoints. We add all four directed combinations from the endpoints of the path of one variable x_k to the endpoints of the path of the next variable x_{k+1}. Then any Hamiltonian path must commit to going left or right along each rung of the ladder as it works its way from the top to the bottom.

It remains to encode the clauses. To this end we have the following gadget. Let us (arbitrarily) fix, for each vertex x_j, the left to right direction as setting $x_j = \texttt{true}$, and the right to left direction as setting $x_j = \texttt{false}$. We want to design a gadget for each clause that is "free to visit" when we travel along the direction of a vertex that corresponds to satisfying the clause.



Fix a clause C_i (where $i \in [m]$). We introduce a new vertex c_i for that clause. For each variable x_j that appears in C_i as x_j (as opposed to $\bar{x}_j$), pick two consecutive (unused) nodes along x_j's path. In the left to right direction (corresponding to $x_j = \texttt{true}$), add an edge from the first of these vertices to the clause-vertex c_i, and an edge from the clause-vertex c_i to the second of these vertices. For each variable x_j that appears in C_i in negated form $\bar{x}_j$, we do something similar in the opposite direction. We pick two consecutive unused nodes along x_j's path. Moving in the right to left direction (corresponding to $x_j = \texttt{false}$), we add an edge from the first of these vertices to the clause-vertex c_i, and an edge from the clause-vertex c_i to the second of these vertices. See the picture below.

Any Hamiltonian path has to visit each clause vertex c_i. The construction is such that if a solution is already traversing the path of some x_j in the direction that satisfies C_i, then it can detour and visit that clause along the way.



The picture on the right sketches the construction which now decorates the "ladder" of paths for each x_j with clause gadgets connecting different paths. A Hamiltonian path will start from one of the two corners and weave through the ladder down to the bottom, choosing one of the two directions for each step.

This finishes the high-level description of the algorithm. We now describe the construction formally for the sake of completeness.

1. For each variable x_j, we create two variables s_j and t_j, which will be opposite ends of the path for x_j.

2. For each variable x_j (where $j \in [n]$), and each clause C_i (where $i \in [m]$), we create two variables $a_{j,i}$ and $b_{j,i}$. For each i, $a_{j,i}$ and $b_{j,i}$ are reserved for a possible gadget for clause C_i. We also have $m+1$ addition vertices $d_{j,1}$ that will act as dividers.

3. For each variable x_j, consider the sequence

$$(s_j, d_{j,0}, a_{j,1}, b_{j,1}, d_{j,1}, a_{j,2}, b_{j,2}, d_{j,2}, \ldots, d_{j,m-1}, a_{j,m}, b_{j,m}, d_{j,m}, t_j),$$

   where the $\ldots$ iterates though the vertices $a_{j,i}, b_{j,i}, d_{k,i}$ and in increasing order of i. We add directions in both directions between every consecutive pair of edges. Thus creates a "rung".

4. For $j = 1, \ldots, n-1$, we add all 4 combinations of directed edges from s_j and t_j to s_{j+1} and t_{j+1}. This strings the rungs together to form a "ladder".

5. For each clause C_i, we create a vertex c_i.

6. For each clause C_i, and each literal of the form x_j in C_i, we add directed edges from $x_{j,i-1}$ to c_i and from c_i to $x_{j,i}$. For each literal of the form $\bar{x}_j$ in C_i, we add directed edges from $x_{j,i}$ to c_i and from c_i to $x_{j,i-1}$.

A helpful fact about the construction is that to avoid revisiting any vertex, one has to traverse all the vertices corresponding to a variable x_j in one direction or the other. The dividers make it easier to prove this and we leave the proof to the reader.

We claim that f is satisfiable iff and only G has a Hamiltonian. First of all, given a satisfying assignment $x \in \{0,1\}^n$, we obtain a Hamiltonian path by first using each x to decide in which direction to traverse the path for each x_j. Each clause vertex can be visited as we traverse the path of the satisfying vertex. Conversely, given a Hamiltonian path - which, in particular, traverses each path gadget in a single direction - we obtain a satisfying assignment by mapping the directions to `true`/`false` assignments.

This completes the proof of Theorem 12.1.

12.3 Additional notes and materials

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture. (There were some issues with the audio.)

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

12.4 Exercises

Exercise 12.1. Suppose you had a polynomial time algorithm for the decision version of the Hamiltonian path problem. That is, given a directed graph G, the algorithm

returns `true` if G has a Hamiltonian path, and `false` if not. Consider the problem of constructing a Hamiltonian path: that is, if G has a Hamiltonian path, then return a Hamiltonian path. Show how to use the polynomial time algorithm for the decision problem to obtain a polynomial time algorithm for Hamiltonian path.

Exercise 12.2. Recall the recursive algorithm for longest path in DAG's.

1. Describe a DAG with n vertices where the recursive algorithm (without dynamic programming) would take $O(n!)$ time.
2. More generally, show that for any integers $k \leq n$, there is a DAG with n vertices such that
   (a) Every vertex v has out-degree $\leq k$.
   (b) The recursive algorithm would take $(n-k)^k(k-1)!$ on this input.

Exercise 12.3. A *Hamiltonian cycle* is a cycle that visits every vertex exactly once.

1. Consider the problem of deciding whether a given directed graph has a Hamiltonian cycle. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.
2. Give a polynomial time reduction from the Hamiltonian cycle problem to the Hamiltonian path problem (showing directly that a polynomial time algorithm for Hamiltonian path implies a polynomial algorithm for Hamiltonian cycle).

Exercise 12.4. 1. Consider the Hamiltonian path problem in *undirected* graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. Now consider the Hamiltonian *cycle* problem in undirected graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.5. While we showed that Hamiltonian path is unlikely to permit polynomial time algorithms, perhaps it is still interesting to try to develop faster exponential time algorithms. The simplest approach enumerates all permutations of the vertices and checks to see if they describe a Hamiltonian cycle; this takes $O(n! \operatorname{poly}(n))$ time. Can one do better? Design and analyze an algorithm that computes a Hamiltonian path in $O(2^n \operatorname{poly}(n))$ time. (The smaller the $\operatorname{poly}(n)$ term the better, but the key point is to reduce $n!$ to 2^n.)

Exercise 12.6. Let $G = (V, E)$ be a directed graph with real-valued edge lengths $\ell : E \to \mathbb{R}$, and let $s, t \in V$. Consider the problem of computing the length of the shortest path from s to t. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.7. Here we consider an extension of shortest (s, t)-walks where one has to visit a family of vertices specified by the input. The input consists of a directed graph $G = (V, E)$ with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$, as well as a list of vertices $x_1, \ldots, x_k \in V$. The high-level goal in both of the following problems is to compute the length of the shortest walk that visits all k vertices.

1. Suppose you are allowed to visit $x_1, \ldots, x_k$ in any order. Consider the problem of computing the length of the shortest walk visiting all of $x_1, \ldots, x_k$ in any order. (You may assume no vertices repeat in $x_1, \ldots, x_k$.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. Suppose you have to visit $x_1, \ldots, x_k$ in the listed order. (Here vertices may repeat, but for simplicity you may assume that any two consecutive vertices v_i and v_{i+1} are distinct.) Consider the problem of computing the length of the shortest walk visiting $x_1, \ldots, x_k$ in order. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.8. The following problems consider variations for the longest path problem that instead ask for longest walks (with varying definitions of "longest"). Here we recall that a walk may repeat vertices and edges, whereas a path cannot. Let

$G = (V, E)$ be a directed graph with m edges and n vertices. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. Compute the length of the longest walk in G.
2. Compute the maximum number of distinct vertices of any walk in G.
3. Compute the maximum number of distinct edges of any walk in G.

Exercise 12.9. Let $G = (V, E)$ be a directed graph with vertex weights $w : V \to \mathbb{R}$. We say that the *vertex weight of a walk* going through vertices $v_1, v_2, \ldots, v_k$ (in order, with vertices possible repeating) is the sum weight $w(v_1) + w(v_2) + \cdots + w(v_k)$.

Let $k \in \mathbb{N}$ be a given parameter with $k \leq n$. Consider the problem of computing the minimum vertex weight of any walk with exactly k vertices. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.10. Let $G = (V, E)$ be a directed graph with m edges and n vertices, where each vertex $v \in V$ is given an integer label $\ell(v) \in \mathbb{N}$. The goal is to find the length of the longest path[1] in G where the labels of the vertices are (strictly) increasing.

1. Suppose G is a DAG. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.
2. Consider now the problem for general graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.
3. Suppose instead we ask for the length of the longest path in G where [2] is weakly increasing if $x_1 \leq x_2 \leq \cdots \leq x_k$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.11. Let $G = (V, E)$ be a simple[3] directed graph where each edge is

[1] Recall that a path is a walk that does not repeat vertices.
[2] A sequence G is a general graph and the labels of the vertices are weakly increasing.$x_1, \ldots, x_k$
[3] i.e., G has no parallel edges.

colored `red`, `white`, or `blue`. An American path is a path where the edge colors alternate `red`, `white`, `blue`, `red`, `white`, `blue`... (The first edge may be any color.) Consider the problem of computing the length of the longest American path in the graph. For this problem in each of the following settings, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT. You may assume that one can look up the color of a path in constant time.

1. G is a DAG.
2. G is a simple directed graph.

Exercise 12.12. Consider a directed graph $G = (V, E)$ that is *transitively closed*. That is, a vertex u can reach a vertex v iff there is an edge from u to v. Consider the problem of finding a Hamiltonian path in a transitively closed graph. That is, the input consists of a transitively closed graph, and the goal is to find a Hamiltonian path in the graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.13. Let $G = (V, E)$ be a DAG, and $s, t \in V$. Consider the problem of computing the *number* of paths from s to t in G.[4] For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.14. A polygonal path is a sequence of line segments joined end-to-end; the endpoints of these line segments are called the vertices of the path. The length of a polygonal path is defined as the sum of the lengths of its segments. A polygonal path with vertices $(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)$ is *monotonically increasing* if $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for every index i – informally, each vertex of the path is above and to the right of its predecessor.

Consider the problem of computing the length of the longest monotonically increasing path among a set S of n points. Here the input consists of a set S of n points in the plane, represented as two arrays $X[1..n]$ and $Y[1..n]$ (for the x-coordinates and y-coordinates, respectively). You may assume access to a subroutine `Length(`x`,`y`,`x'`,`y'`)`

[4]Technically, there could be as many as $n!$ paths in which case each arithmetic operation would take $O(\log(n!)) = O(n \log n)$ time. For this exercise, let us assume for simplicity that arithmetic operations take $O(1)$ time anyway.

that returns the length of the segment from (x, y) to (x', y') in $O(1)$ time. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.15. It's-a me, Mario! This problem is inspired by Super Mario World, where Mario must make it from some starting point to the flag in front of a castle, collecting as many coins as possible along the way.

Let $G = (V, E)$ be a directed graph where each vertex $v \in V$ has $C_v \geq 0$ coins. For a walk in G, we say that the *number of coins collected by the walk* is the total sum of C_v over all *distinct* vertices v in the walk. (If we visit a vertex v more than once, we still only get C_v coins total.) Given $s, t \in V$, the goal is to compute the maximum number of coins collected by any (s, t)-walk.

1. (5 points) Let G be a DAG. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. (5 points) Let G be a general directed graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

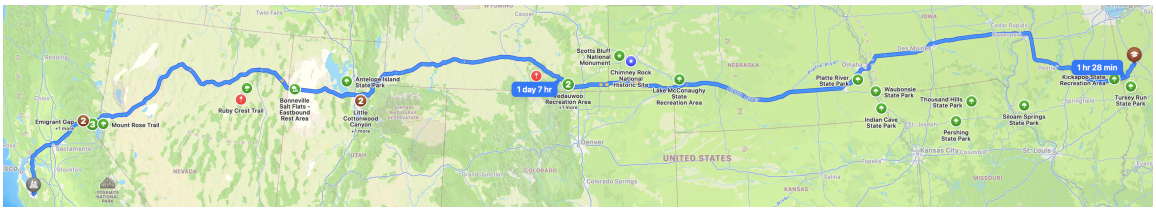
Exercise 12.16. Let $G = (V, E)$ be a directed graph with no parallel edges. We say that a walk w in G has *distinct edges* if no edge appears twice in w. Consider the problem of computing the length of the longest walk in G with distinct edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.17. Let $G = (V, E)$ be a directed graph with m edges, n vertices, and nonnegative integer edge lengths $\ell : E \to \mathbb{Z}_{\geq 0}$. (Edges may have length 0.) Given two vertices s and t, there may be many shortest paths (with respect to the edge lengths ℓ) from s to t. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. (5 points) Compute the minimum number of edges of any shortest (s, t)-path (with respect to ℓ) from s to t.

2. (5 points) Compute the maximum number of edges of any shortest (s, t)-path (with respect to ℓ) from s to t.

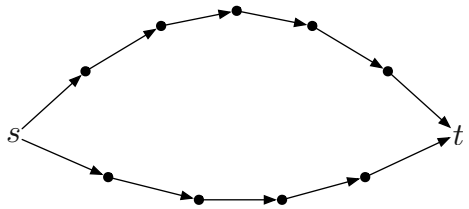
Exercise 12.18. Imagine you are planning a long road trip to San Jose, CA.



This time you haven't fixed your route and you're not so obsessed with gas stations. You decided you want to do some sightseeing along the way and you've marked out some points of interest across America. Your goal is to visit as many of these points as possible, but you are very strict about the phrase "along the way". You should never go backwards – you should always make progress towards your destination, even if its not the fastest way to San Jose.

We model this problem with a directed graph $G = (V, E)$ with positive real edge lengths $\ell : E \to \mathbb{R}_{>0}$. You have a vertex s marking the start and a vertex t marking the end. You are given a set $A \subseteq V$ of size $p = |A|$, which define the tourist attractions you would like to see. The goal is to compute the maximum number of attractions that you can visit along any walk $(v_0 = s), v_1, \ldots, (v_k = t)$ from s to t such that for $i = 1, \ldots, k$, the distance from v_i to t is (strictly) less than the distance from v_{i-1} to t (with respect to ℓ). For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.19. Let $G = (V, E)$ be a directed graph and $s, t \in V$. We say that two (s, t)-paths $p, q : s \rightsquigarrow t$ are *internally disjoint* if they are vertex disjoint except at their endpoints s and t.



Consider the problem of computing the maximum number of edges of any two internally disjoint (s,t)-paths (for fixed s and t). For example, the picture above gives two internally disjoint (s,t)-paths with a total of 11 edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.20. Recall that a sequence of numbers $x_1, \dots, x_k$ is *(weakly) convex* if $x_{i+1} - x_i \geq x_i - x_{i-1}$ for $i = 2, \dots, k-1$. The sequence $x_1, \dots, x_k$ is *strictly convex* if $x_{i+1} - x_i > x_i - x_{i-1}$ for $i = 2, \dots, k-1$.

Let $G = (V, E)$ be an unweighted directed graph with m edges and n vertices, where each vertex $v \in V$ is given a positive label $\ell(v) \in \mathbb{R}_{>0}$.

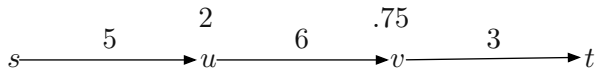
1. (5 points) Consider the problem of computing the length of the longest path in G where the vertex labels are weakly convex. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. (5 points) Now consider the problem of computing the length of the longest walk in G where the vertex labels are strictly convex. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.21. In various games like Mario Kart, not only are you trying to get from start to finish as fast as possible, but there are various items along the way that speed you up or slow you down. These items inspire the "multipliers" in the following variation of shortest paths.

Let $G = (V, E)$ be a DAG and $s, t \in V$. Let the edges have positive labels $\ell : E \to \mathbb{R}_{>0}$ representing the amount of time it takes to traverse the edge at unit speed. For each vertex $v \in V$, let $\alpha(v) > 0$ be a "speed multiplier". For example, if a vertex v has multiplier $\alpha(v) = 2$, then your speed doubles as you pass through v; if $\alpha(v) = 1/2$, then your speed decreases by a factor of 2 as you pass through v.

You start at s in a vehicle at unit speed. Your goal is to get to t as fast as possible. Along the way your speed increases or decreases multiplicatively based on the speed multipliers $\alpha(v)$ of each vertex you traverse. (For simplicity we assume $\alpha(s) = \alpha(t) = 1$.)



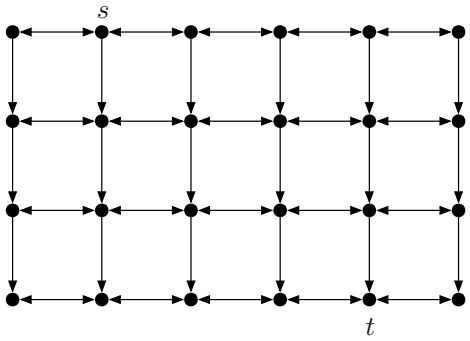
For example, in the path from s to t above, the edges are labeled by the time at unit speed, and the vertices are labeled by the speed multipliers. Going from s to t, the first edge is traversed at unit speed and takes 5 units of time, the second edge is traversed at 2x speed and takes 3 units of time, and the last edge is traversed at 1.5x speed (since $2 \times .75 = 1.5$) and takes 2 units of time, for 10 units of time total.

Consider the problem of computing the minimum amount of time needed to get from s to t, taking into account the speed multipliers of each vertex. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

(Extra credit / food for thought: what if G was a general graph?)

Exercise 12.22. Let $G = (V, E)$ be a directed graph with real-valued (and possibly negative) edge lengths $\ell : E \to \mathbb{R}$, and $s, t \in V$. We are interest in the length of the shortest (s, t)-path for a special class of "downwards grid graphs" G.

A "downward grid graph" G is parametrized by a height $m \in \mathbb{N}$ and a width $n \in \mathbb{N}$. All the vertices are arranged in an $m \times n$ grid. We let (i, j) denote the vertex in row i and column j. There are edges between every adjacent pair of edges except going up. More specifically, for $i \in [m]$ and $j \in [n]$, there is an arc



- (Down) to $(i + 1, j)$ when $i < m$;
- (Right) to $(i, j + 1)$ when $j < n$; and
- (Left) to $(i, j - 1)$, when $j > 1$.

In particular, there are no edges going "up" from (i, j) to $(i - 1, j)$. Note that G has $O(mn)$ vertices and $O(mn)$ edges.

Consider the problem of computing the length of the shortest (s, t)-path in a downward grid graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.23. Recall that the Economist recently reported[5] that the number of toll roads in America is increasing rapidly. It used to be hard enough to find the shortest path. Now you have to do it within a budget.

Right now, drivers pay to access 6,300 miles of Americas 160,000 miles (or so) of highway, but that number is going up. Here in Indiana, in June, the state passed a law giving the state authority to raise tolls on all its existing interstate highways. The Economist says that this trend is broadly due to a parallel trend of decreasing taxes on gasoline, which used to pay a larger share of the cost for maintaining roads and highways.

Let $G = (V, E)$ be a directed graph with nonnegative integer edge lengths $\ell : E \to \mathbb{Z}_{\geq 0}$ and *nonnegative integer edge costs* $\texttt{cost} : E \to \mathbb{Z}_{\geq 0}$, which model toll roads. We define the *cost* of a walk in G as the sum of edge costs (with repetition) along that walk. Let $k \in \mathbb{N}$ be a given budget. Consider the problem of computing the length of the shortest (s, t)-walk that costs at most k. (Unlike the midterm, we do not assume that k is polynomially bounded by m and n.) For this problem, either design and analyze a polynomial time algorithm, or prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 12.24. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two directed graphs with $n_i = |V_i|$ vertices and $m_i = |E_i|$ edges each. Suppose we have a labeling $\ell : V_1 \cup V_2 \to \mathcal{A}$, where $\mathcal{A}$ is an alphabet. (Different vertices may share the same letter.) A *common path* is defined as a path p_1 in G_1 and a path p_2 in G_2 such that the letters along the paths are the exact same. (The first vertices of p_1 and p_2 have the same letter, the second vertices have the same letter, etc. p_1 and p_2 must have the same length.) Consider the problem of computing the length of a longest common path in G_1 and G_2.

1. (5 points) Suppose G_1 and G_2 are both DAGs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. (5 points) Now suppose G_1 and G_2 are general directed graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

[5] *Toll roads are spreading in America*, December 18, 2025. https://www.economist.com/united-states/2025/12/18/toll-roads-are-spreading-in-america

Sample solution.

Problem. Let $G = (V, E)$ be a DAG. Consider the problem of computing the length of (i.e., the number of edges in) the longest path in G. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Solution.

1. Recursive spec / induction hypothesis. We first observe that all walks are paths in a DAG, so it is equivalent to compute the length of the longest walk. For a vertex $v \in V$, we define

$\texttt{LW}(v)$ = the length of the longest walk in G.

2. Recursive implementation. $\underline{\texttt{LW}(v)}$

1. If v is a sink then return 0.
2. Otherwise return the maximum of $1 + \texttt{LW}(w)$ over all edges (v, w) leaving v.

3. Solving the original problem. We return the maximum of $\texttt{LW}(v)$ over all $v \in V$.

4. Running time with caching / dynamic programming. Each $\texttt{LW}(v)$ takes time proportional to the out-degree of v (plus a constant). With dynamic programming, and summing over all v counts each edge once, so we obtain a $O(m + n)$ running time overall.

5. Proof. Fix v and consider $\texttt{LW}(v)$. If v is a sink then any walk starting at w is just w, hence has length 0, as returned by $\texttt{LW}(v)$.
Otherwise the LW from v has at least one edge (v, w) to an out-neighbor w of v. The length of the LW from v is one plus the length of a walk from w. By induction on the topological ordering of V, $\texttt{LW}(w)$ returns the length of the longest walk from w. So $1 + \texttt{LW}(w)$ is the length of the longest walk from v, and this value is returned in $\texttt{LW}(v)$.

Chapter 13

Optimization in graphs

This lecture continues to investigate the subtle distinctions between problems that are polynomial time solvable, and those that resist our best algorithmic efforts. This time, all our problems are posed in undirected graphs. We will also consider special cases of undirected graphs to see if the additional structure changes the fundamental character of the problem. The problems introduced here are called the *independent set*, *(vertex) coloring*, and *dominating set* problems. They are all simple to define. Of these only the independent set problem will be presented completely. Vertex coloring and dominating set will be partly presented and partly delegated to exercises.

13.1 Independent set

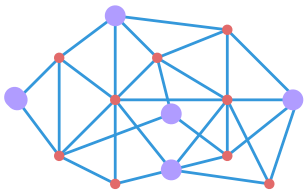
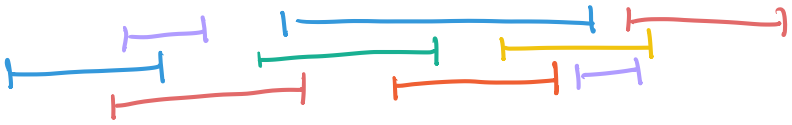


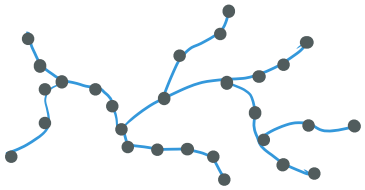
Figure 13.1: An independent set of size 5.

The independent set problem is a graph problem defined as follows. Let $G = (V, E)$ be an undirected graph. Recall that graphs consist of vertices (V) and edges (E) that connect pairs of vertices. A set of vertices $S \subset V$ is *independent* if any two vertices $a, b \in S$ are not connected by an edge. The *maximum independent set problem* asks for the maximum cardinality independent set. In the decision version of the problem, the input consists of a graph $G = (V, E)$ and an integer $k \in \mathbb{N}$, and asks if there exists an independent set S with size $|S| \geq k$. One can also consider the weighted version where we are also given positive vertex weights $w : V \to \mathbb{R}_{>0}$, and the goal is to choose the maximum weight independent set (or decide if the maximum weight is at least an input parameter T).

There are many special cases of independent set that are useful. We will discuss two of them. The first is for *intervals on a line*. The input consists of n intervals $\mathcal{I} = \{I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]\}$, with positive edge weights $w : \mathcal{I} \to \mathbb{R}_{>0}$. We say that a set of intervals $S \subset \mathcal{I}$ is independent if they are pairwise disjoint; i.e., $I \cap J = \emptyset$ for every two intervals $I, J \in S$. This is the same as the graph definition if we identify each interval as a vertex, and have an edge between every two intervals that overlap.



The second example is the special case where the graph G is a tree T. Since trees are a particularly simple class of graphs, it is interesting to investigate whether one can obtain a better algorithm for independent set in trees than in graphs in general.



13.1.1 Independent set in graphs

Let us first consider the most general formulation of maximum weight independent set in graphs. Let $G = (V, E)$ be a graph with m edges and n vertices. Let $w : V \to \mathbb{R}_{>0}$ assign positive weights to the vertices. The most general goal is to compute the maximum weight independent set $S \subset V$.

We first observe that such a set S can be found by brute force. Here is a recursive algorithm.

`MWIS(`$G = (V, E)$`,` $w : V \to \mathbb{R}_{\geq 0}$`)`

1. If $V = \emptyset$ then return 0
2. Let v be any vertex in V
3. Return the maximum of:

   `//` *all independent sets excluding* v

   A. `MWIS(G - v,w)` where $G - v$ denotes the graph obtained by removing v and all its incident edges from G.

   `//` *all independent sets including* v

   B. $w(v) +$ `MWIS(G',w)` where G' obtained by removing v and its neighbors

The recursive algorithm will take exponential time. This is slow, but it establishes a baseline. Can we do better? What about the simpler, unweighted setting?

Let us instead try to show that the problem should be hard - in particular, by proving the following.

Theorem 13.1. *A polynomial time algorithm for maximum cardinality independent set implies a polynomial time algorithm for SAT.*

We know that SAT can be reduced to 3-SAT in polynomial time, so it suffices to show that a polynomial time algorithm for maximum cardinality independent set implies a polynomial time algorithm for 3-SAT.

Let $f(x_1, \ldots, x_n)$ be a 3-SAT formula with m clauses $C_1, \ldots, C_m$ and n variables $x_1, \ldots, x_n$. We interpret each C_i as a set of three elements out of the set of symbols $X = \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}$. We want to convert f into a graph, where somehow an independent set larger than some particular size would indicate that f is satisfiable.

Clause gadgets. For each clause C, we make a triangle (i.e., three new vertices fully connected by edges). Each vertex in the triangle corresponds to a literal in the clause. For example, if $C = (x_1 \vee \bar{x}_2 \vee x_3)$, we create a triangle with three vertices, with one vertex corresponding to x_1, $\bar{x}_2$, and x_3. Thus a symbol such as x_1 or $\bar{x}_2$ may correspond to multiple vertices in the construction, with one vertex for each clause it appears in. For m clauses, we have m disjoint triangles, with $3m$ vertices and $3m$ edges in all. Any independent set in G has at most one vertex per clause/triangle.

We need to encode the fact that we can only set each variable x_i to be exactly one of true or false, and not both. Currently, an independent set can contain vertices

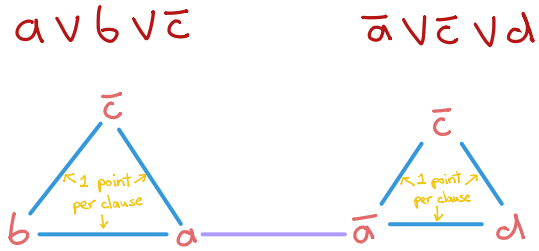


Figure 13.2: The gadget encoding disjunctions over three variables as triangles.

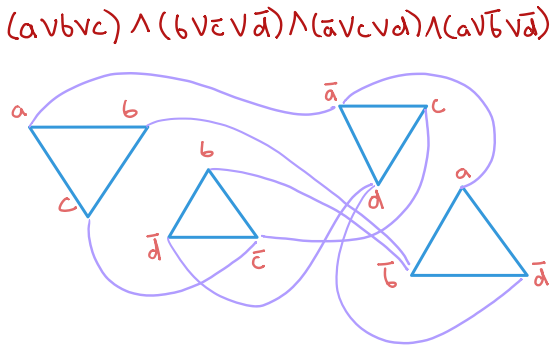


Figure 13.3: Encoding the 3-SAT formula $f(a,b,c,d) = (a \vee b \vee c) \wedge \left(b \vee \bar{c} \vee \bar{d}\right) \wedge \left(a \vee \bar{b} \vee \bar{d}\right)$ as a graph, where the maximum independent set indicates whether or not f is satisfiable.

for both x_1 and $\bar{x}_1$. To prevent this, we add an edge between every pair of vertices of the form x_i and $\bar{x}_i$, for all i.

This completes the construction. In total the graph has $O(m)$ vertices and $O(m+n)$ edges. A minimal example is given in Fig. 13.2. A more ornate example is given in Fig. 13.3.

We now claim that $f(x_1, \ldots, x_n)$ is satisfiable if and only if there is an independent set of size m in the graph G. Suppose first that there is an independent set S of size m. For each variable x_i, S may contain vertices corresponding to x_i, or vertices corresponding to $\bar{x}_i$, but not both, because of the edges we added in the second part of our construction. Consider the assignment where we set $x_i = \texttt{true}$ if any of the x_i-vertices appear in S, or $x_i = \texttt{false}$ if any of the $\bar{x}_i$-vertices appear in S. (If neither type appears in S, then we set x_i to whatever we want.) We claim that this assignment satisfies all the clauses in f. We know that (a) there are m triangles (one per clause), (b) S has m vertices, and (c) S can have at most one vertex per triangle (to be independent). So S has exactly one vertex from every clause gadget. For each clause in f, our assignment satisfies the clause via the symbol for that vertex.

Conversely, suppose f is satisfiable, and fix a satisfying assignment. For each

clause C, choose a literal that satisfies it, and add the corresponding vertex from C's gadget in our graph to a set S. Since these choices are disjoint, S will have m vertices. We claim that S is independent. Indeed, picking one vertex per clause implies we satisfy all the triangle edges. Moreover we only choose a vertex corresponding to the symbol x_i if $x_i = \texttt{true}$ in the satisfying assignment, and we only choose a vertex corresponding to the symbol $\bar{x}_i$ if $x_i = \texttt{false}$. In particular, we can't choose both. So S is independent over the second set of edges as well. Altogether, S is an independent set of size m.

To conclude, given a 3-SAT formula f with m clauses, we have built a graph G, of size polynomial in the size of f, that has an independent set of size m iff f is satisfiable. If we had a polynomial time algorithm for maximum independent set, we could run it on this graph and decide if f is satisfiable. This completes the proof.

13.1.2 Independent sets of intervals

While independent set on graphs turned out to be difficult, this does not preclude a more efficient algorithm for intervals, since this is a special case. Thus let us consider the problem for intervals with renewed optimism.

Let $\mathcal{I} = \{I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]\}$ be a set of intervals. Let $w : \mathcal{I} \to \mathbb{R}_{>0}$ assign positive weights to each interval. The goal is to compute the maximum weight independent set of intervals.

A lazy approach to this problem is to convert it to a graph by first comparing all pairs of intervals to identify all conflicting pairs. This would take quadratic time. (One can also do this faster - how?) Then one can compute brute force in the graphical setting as discussed above. But of course, this is far from efficient. More importantly, just because we can interpret the interval problem in terms of a graph, doesn't mean we should automatically treat it as a graph problem. This is because the intervals have more structure then graphs (in general), and this structure in particular makes it more difficult to reduce from SAT.

For graphs we explored brute force algorithms that recursively test one element at a time. For the sake of concreteness, let us quickly rewrite that algorithm in terms of intervals. First, the recursive spec: for a set of intervals $\mathcal{I}$ and weights $w : \mathcal{I} \to \mathbb{R}_{>0}$, we define

> `MWIS-intervals(`$\mathcal{I}$`,`w`)` = the weight of the maximum weight independent subset of intervals of $\mathcal{I}$.

We now implement `MWIS-intervals` as specified above.

`MWIS-intervals(`$\mathcal{I}$`,` $w : \mathcal{I} \to \mathbb{R}_{>0}$`)`

1. If $\mathcal{I} = \emptyset$ then return 0
2. Let $I \in \mathcal{I}$ be any interval
3. Let $\mathcal{I}' = \{J \in \mathcal{I} : J \cap I = \emptyset\}$ *// = set of intervals disjoint from I*
4. return the maximum of
   - */* The MWIS among those excluding I. */*
   - A. `MWIS-intervals(`$\mathcal{I} \setminus \{I\}$`,`$w$`)`
   - */* The MWIS among those including I. */*
   - B. $w(I) +$ `MWIS-intervals(`$\mathcal{I}'$`,` w`)`

In our brute force algorithm, we are not selective about the order in which we process the elements. In a graph, there is no obvious order to the vertices anyway. In intervals, however, there are natural ways to order the intervals and this can make a big difference.

Let us sort $\mathcal{I}$ in increasing order of left endpoint (a_i) in $O(n \log n)$ time, and henceforth assume that

$$\mathcal{I} = \{I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]\}$$

lists the intervals in increasing order of a_i. Let us adjust the brute force algorithm again so that it processes the intervals in increasing order of left endpoint. The revised pseudocode is as follows.

`MWIS-intervals(`$\mathcal{I} = \{I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]\}$`,` $w : \mathcal{I} \to \mathbb{R}_{>0}$`)`

/ We assume $I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]$ is sorted in increasing order of a_i. */*

1. If $n = 0$ then return 0
2. Let $i > 1$ be the first index such that $a_i > b_1$

// I_i is the first interval disjoint from I_1, and $\{I_i, \ldots, I_n\}$ = set of intervals disjoint from I_1

3. return the maximum of
   - */* The MWIS among those excluding I_1 */*
   - A. `MWIS-intervals(`$\{I_2, \ldots, I_n\}$`,`w`)`
   - */* The MWIS among those including I_1 */*
   - B. $w(I_1) +$ `MWIS-intervals(`$\{I_i, \ldots, I_n\}$`,` w`)`

In this ordered version of brute force, every recursive call is for a subset of intervals of the form $\{I_i = [a_i, b_i], I_{i+1} = [a_{i+1}, b_{i+1}], \ldots, I_n = [a_n, b_n]\}$ for some index i; that is, a suffix of our *ordered* sequence of intervals. There are only $n + 1$ suffixes (including the empty set), an ideal situation for caching.

We now develop a dynamic programming algorithm for the maximum weight independent set of intervals. Fix an input consisting of n intervals $\mathcal{I} = \{I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]\}$ and weights $w : \mathcal{I} \to \mathbb{R}_{>0}$. Sorting the intervals $\mathcal{I}$ in $O(n \log n)$, we assume the intervals are indexed in increasing order of left endpoint a_i. For $i \in \mathbb{N}$, we define

> `MWIS-intervals(`i`)` = the maximum weight of any independent set of intervals among $\{I_i, I_{i+1}, \ldots, I_n\}$.

We implement the spec with the following pseudocode.

$$\texttt{MWIS-intervals}(i) = \begin{cases} 0 & \text{if } i > n, \\ \max\begin{Bmatrix} \texttt{MWIS-intervals}(i+1), \\ w(I_i) + \texttt{MWIS-intervals}(j(i)) \end{Bmatrix} & \text{otherwise,} \end{cases}$$

where $j(i)$ is the first index j such that $a_j > b_i$. We can identify $j(i)$ by scanning the list of intervals in order of left endpoint. We apply dynamic programming to the above recursive code to make sure we never solve the same subproblem twice.

For the running time, we have n subproblems, each of which takes $O(n)$ time to compute, excluding recursive subcalls. (The $O(n)$ time comes from scanning for the next index $j(i)$.) The space usage is $O(n)$, since there are $O(n)$ many subproblems. The solution to the original problem is given by the call `MWIS-intervals(1)`. Correctness is inherited from brute force.

Can one do better? In fact, yes. For fixed i, we spend $O(n)$ time to find the first index $j(i)$ such that $a_j > b_i$. But instead one can do binary search to find $j(i)$, in $O(\log n)$ time. This improves the running time of each subproblem to $O(\log n)$, and the overall running time to $O(n \log n)$. This is our first examples where a dynamic programming algorithm is sped up by an adjustment to the implementation.

Taking a step back, we see that there is substantially more structure to intervals than a generic graph. A provably hard problem in graphs can be solved as fast as sorting in interval graphs!

13.1.3 Independent set in trees

We now consider the independent set problem in our second special case of graphs: trees. Here we are given a tree $T = (V, E)$ and (optionally) vertex weights $w : V \to \mathbb{R}$.

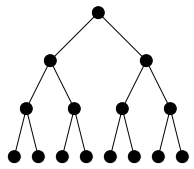
The goal is to compute the maximum weight independent set $S \subset V$. The question really is whether trees are as general as graphs, or as structured as intervals on a line. Let us make two initial observations relating trees to our previous reduction.

1. The reduction from 3SAT to independent set was full of triangles. But trees cannot have triangles, or any cycle, for that matter.

2. The graph corresponding to a set of intervals is also not a tree - any three overlapping intervals form a triangle.

There is clearly much more structure to trees than graphs. As mentioned above, trees have no cycles. If we remove an edge or a vertex, then we get two entirely disconnected parts. In fact one can always find a vertex whose removal breaks up the graph into subtrees with at most half the vertices (why?) - a remarkable property that is certainly not true for graphs.

Between this chapter and the last, we have seen multiple problems turn out to be easy when we take advantage of a clearly defined, *linear* ordering. But there is no clear notion of "left" and "right" in a tree. But while trees don't have a linear structure, they do have a *hierarchical* structure that is very useful.



Fix any node $r \in V$ to be the root, and interpret T as a rooted tree. Now we can speak of "children", "parents", "subtrees" and so forth. There is a clear notion of "top" – the root r – and 'bottom" – at the leaves. Let's use this ordering to design a recursive algorithm for the maximum weight independent set. From a hierarchical point of view, it is natural to design a recursive algorithm using subtrees to define the subproblems. Consider the following recursive spec. For a vertex $v \in V$, we define

> `MWIS(`v`)` = the maximum weight of any independent set in the subtree rooted at v.

Let us now sketch out an implementation `MWIS(`v`)`. Fix v, and suppose we want to find the maximum weight independent set in the subtree rooted at v. A natural decision is whether or not to include v in the independent set. If we don't include v, then we can recurse on each of the children to compute the maximum weight independent set in each subtree. If we do include v, then... wait. Actually, it's not so obvious. When we recurse on a child x of v, this may return (the value of) an independent set that includes x. But this is incompatible with our decision to include v in the independent set. The following pseudocode summarizes our progress and poinpoints where we get stuck.

`MWIS(`v`)`

1. Return the maximum of

   `/*` *Don't include v in the independent set* `*/`

   A. The sum of `MWIS(`x`)` over all children x of v

   `/*` *Include v in the independent set* `*/`

   B. The weight of v plus the sum of ~~`MWIS(`x`)`~~ ??? over all children x of v

In step `(1.B)`, where we got stuck, what did we really want? Not the MWIS in the subtree rooted at x, since this can include x. Rather we need the MWIS in the subtree rooted at x *among those that exclude* x. `MWIS(`x`)` simply doesn't offer this. The easiest way to address this defficiency is to strengthen the definition `MWIS(`x`)`. We introduce a boolean flag, called `exclude-root` $\in \{0, 1\}$, as an additional parameter, and define `MWIS(`v`,exclude-root)` as follows.

> `MWIS(`v`,exclude-root)`: If `exclude-root` = `true`, then the maximum weight independent set in the subtree rooted at v among those excluding v. If `exclude-root` = `false`, then the maximum weight independent set in the subtree rooted at v (which may or may not include v).

This stronger definition makes the implementation effortless.

`MWIS(`v`,exclude-root)`

1. If `exclude-root`

   `/*` *Don't include v in the independent set* `*/`

   A. Return the sum of `MWIS(`x`,false)` over all children x of v

2. Otherwise return the maximum of:

   `/*` *Don't include v in the independent set* `*/`

   A. The sum of `MWIS(`x`,false)` over all children x of v.

   `/*` *Include v in the independent set* `*/`

   B. The weight of v plus the sum of `MWIS(`x`,true)` over all children x of v

Good. It was a little trickier, but we have now established a recursive algorithm for computing the maximum weight independent set. The overall answer is given by `MWIS(`r`,false)`. But our algorithm is not yet efficient. To this end – as usual – we

apply caching. Let us save the answers to each `MWIS(`v`,true)` and `MWIS(`v`,false)` in a table[1] of size $2n$. Then we never compute the same answer twice.

Consider the running time. We have

$$(\#\text{ subproblems})(\text{time per subproblem}) = n \times O(n) = O\Big(n^2\Big).$$

Here we upperbound the "time per problem" by $O(n)$ because a vertex may have up to n children. But this analysis is too conservative. The total number of children over all nodes is n. A more refined analysis first observes that

$$(\text{time spent on } \texttt{MWIS(}v\texttt{, } \cdots\texttt{)}) = O(\#\text{ children of } v).$$

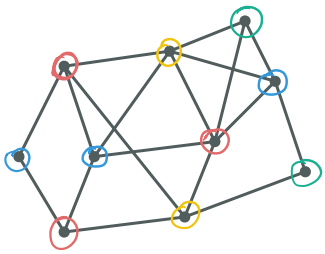
Summing over all v, we have

$$\sum_v (\text{time spent on } \texttt{MWIS(}v\texttt{, } \cdots\texttt{)}) = O\left(\sum_v \#\text{ children of } v\right) = O(n).$$

So actually the running time is $O(n)$. We also mention that the space usage is $O(n)$ because there are $O(n)$ subproblems, each requiring constant space. Thus we have a $O(n)$ time algorithm for maximum weight independent set in trees. *Which is faster than for intervals! Faster than sorting!*

13.2 Graph coloring

Graph coloring problems are an important topic in graph theory. Generally speaking, vertex coloring problems are problems where you want to color the vertices of an undirected graph so that any two adjacent vertices have different colors. More formally, we can define a *proper vertex coloring* as an assignment $\pi : V \to [k]$, for some integer k, such that for any edge $\{u, v\} \in E$, $\pi(u) \neq \pi(v)$. The goal is to find a vertex coloring $\pi : V \to [k]$ for k as small as possible.



[1]We assume the vertices are number 1 through n. Otherwise one can number them in a preprocessing step.

Besides being a useful problem in practice (e.g., assigning non-overlapping frequency ranges to nearby cell-phone towers), it is an important problem in combinatorics, where graph theorists have discovered remarkable structural theorems such as the following famous theorem.

Theorem 13.2 (Brook's theorem, [Bro41])**.** *Let G be an undirected graph with maximum degree Δ. Then G can be colored in Δ colors unless G is either (a) a complete graph or (b) an odd cycle.*

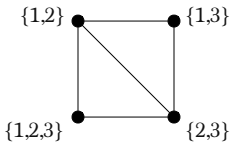
We briefly point out there are many related graph problems of interest. For example, in edge coloring, the goal is to color the edges of an undirected graph so that no two adjacent edges[2] have the same color. For edge coloring, *Vizing's theorem* asserts that every graph with maximum degree Δ can be edge-colored with at most Δ colors.

Let us return to vertex coloring. Here we are interested in the optimization problem rather than absolute bounds such as Brook's theorem. Given an undirected graph $G = (V, E)$, the goal is to color G with as few colors as possible. Or, as a decision problem: given an undirected graph $G = (V, E)$, and an integer $k \in \mathbb{N}$, can G be colored in k or fewer colors? There is also the k-coloring problem where k is fixed, but otherwise asks the same question: Given as input a graph $G = (V, E)$, can G be colored in k or fewer colors? Note that for k-coloring, an $n^{f(k)}$ running time, for any function f, counts as polynomial time.

This problem is yet another search problem, since it is easy to check if an assignment is a proper coloring with at most k colors. So in particular we have an exponential time algorithm by brute force. Can one do better? What about the special case of $k = 3$ – 3-coloring?

Theorem 13.3. *A polynomial time algorithm for vertex 3-coloring implies a polynomial time algorithm for SAT.*

To prove the above it is convenient to introduce the following more general variant of k-coloring. In the *subset k-coloring problem*, each vertex v in the graph is annotated with a subset $S_v \subset [k]$ of colors available to that vertex. The goal is to properly color graph the graph $\pi : V \to [k]$ such that for each k, $\pi(v) \in S_v$.

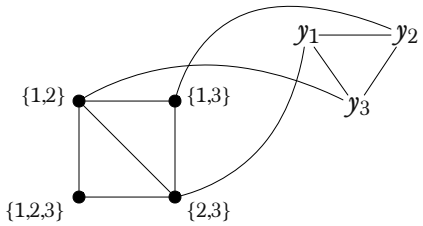


[2]Two edges are adjacent if they have a common endpoint.

Lemma 13.4. *Let $k \in \mathbb{N}$ be fixed. A polynomial time algorithm for k-coloring implies a polynomial time algorithm for subset-k coloring.*

Proof. Assume an instance of subset-k coloring defined by a graph $G = (V, E)$ with m edges and n vertices and, for each vertex v, a set $S_v \subset [k]$ of colors available to v.

To reduce to k-coloring, we will construct a new graph H that extends the original graph G. Starting from G, we first introduce a clique K_k of k colors. That is, we create k new vertices $y_1, \dots, y_k$ and add an edge between every pair of new vertices. Observe that any k-coloring of this clique requires every y_i to receive a distinct color, and this uses up all k colors. (Conceptually we interpret the color assigned to y_i as the ith color.) Now, for each vertex v, and each color i omitted from S_v, we add an edge between y_v and i. (Conceptually, this forbids v from taking i in a k-coloring.) Below we draw the construction for $k = 3$.



We claim that G is k-subset-colorable with respect to the sets $\{S_v : v \in V\}$ iff H is k-colorable. First suppose G is k-subset-colorable. Given any subset coloring π of G, we can extend π to a k-coloring on H by coloring $\pi(y_i) = i$ for each i. This does not create any monochromatic edges between the y_i's, nor does it decrease any monochromatic between a y_i and a vertex $v \in V_G$ because $i \notin S_v$ for any edge $\{y_i, v\}$.

Conversely, consider any proper coloring of $\pi : V_H \to [k]$ of H. Without loss of generality we may assume that $\pi(y_i) = i$ for each i.[3] We need to show that $\pi(v) \in S_v$ for all v. Indeed, for any color $i \notin S_v$, the edge $\{y_i, v\}$ implies that $\pi(v) \neq \pi(y_i) = i$. □

To prove Theorem 13.3, it now suffices to show that subset 3-coloring can be used to solve SAT.

Theorem 13.5. *A polynomial time algorithm subset 3-coloring implies a polynomial time algorithm for 3-SAT.*

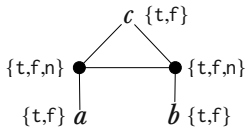
[3]This is because we only care if two colors have the same labels, which is invariant to permutation of the labels.

Let $f(x_1, \ldots, x_n)$ be a 3-SAT formula with n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$. We want to create an instance of subset 3-coloring (of roughly the same size) that is feasible iff f is feasible.

We point out that the names of the colors in a coloring problem are arbitrary. For this reduction, it is convenient to name the colors `true`, `false`, and `nonsense`; or `t`, `f`, and `n` for short.

We first create vertices corresponding to the variables. For each variable x_j, we make two vertices A_j and $\bar{A}_j$ each constrained to have colors from the set $\{\texttt{t}, \texttt{f}\}$. We also add an edge between A_j and $\bar{A}_j$, which forces exactly one of A_j or $\bar{A}_j$ to be colored `true`, and the other to be colored `false`.

Lemma 13.6. *Consider the following instance of subset 3-coloring with five vertices.*

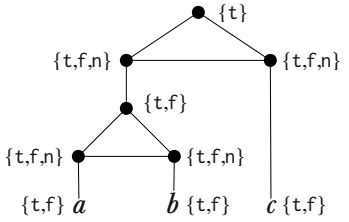


In any proper subset coloring, $c =$ `true` only if $a \vee b =$ `true`.

Proof. By inspection. □

Now we can construct a gadget for 3-CNF clauses. Recall that $(a \vee b \vee c) = ((a \vee b) \vee c)$. Likewise we can compose two OR gadgets as follows.

Lemma 13.7. *Consider the following instance of subset 3-coloring with 8 vertices.*



Then in any proper subset coloring of the above, $a \vee b \vee c =$ `true`.

For each clause C_i, we identify the three vertices corresponding to the literals in that clause. (For example, $x_1 \vee \bar{x}_2 \vee x_3$ corresponds to the vertices A_1, $\bar{A}_2$, and A_3.) We build the 3CNF gadget, per Lemma 13.7 over these three vertices. Each clause generates 6 new variables.

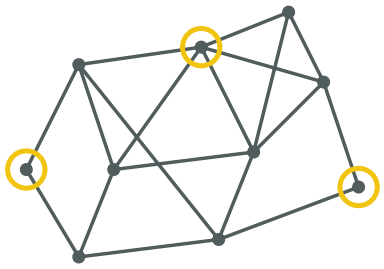
Between the two vertices A_j and $\bar{A}_j$ for each variable x_j, and the constant-size gadgets built over these vertices for each C_i, we obtain an instance of subset 3-coloring that is satisfiable iff f is satisfiable.

Indeed, given a satisfying assignment x, we color $A_j = \texttt{true}$ and $\bar{A}_j = \texttt{false}$ if $x_j = \texttt{true}$, and reverse the colors if $x_j = \texttt{false}$. One can then color each clause gadget appropriately. Conversely, given any proper coloring of the clauses, one constructs an assignment x by setting x_j to be the color of A_j. One can verify that, since the coloring was able to color the clause-gadgets properly, each clause in the formula is satisfied by x. This completes the proof.

Coloring with more than 3 colors. We have not shown that k-coloring is hard for $k > 3$. This is left to the reader in exercise 13.1.

Coloring in interval graphs and trees. As with the independent set problem, the reduction from SAT to 3-coloring does not forbid polynomial time algorithms for trees or intervals. This is left to the reader in exercise 13.3.

13.3 Dominating set



Let $G = (V, E)$ be an undirected graph. A set of vertices $S \subseteq V$ is a *dominating set* if every vertex $v \in V$ is either in S or the neighbor of a vertex in S. The minimum dominating set problem is to compute the minimum cardinality dominating set. The decision version additionally specifies an integer k and the question is whether there is a dominating set of size k (or greater). One can consider weighted versions of the problem where we have weights $w : V \to \mathbb{R}_{>0}$ and the goal is to compute the dominating set of minimum weight.

Perhaps it is not so surprising, after discussing independent set and coloring, that dominating set is also as hard as SAT.

Theorem 13.8. *A polynomial time algorithm for dominating set implies a polynomial time algorithm for SAT.*

Let $f(x_1, \ldots, x_n)$ be a Boolean formula in CNF with n variables, m clauses, and total size p. We will construct an undirected graph G such that G has a dominating set of size n iff f is satisfiable. Here are the first steps in making G.

1. For every variable x_i, we create two vertices denoted v_i and $\bar{v}_i$. v_i is associated with the symbol x_i and the assignment $x_i = \texttt{true}$; $\bar{v}_i$ is associated with the symbol $\bar{x}_i$ and the assignment $x_i = \texttt{false}$.
2. For every clause C_j, we create a vertex w_j.

3. For every clause C_j and every variable x_i appearing as x_i (without negation) in C_j, we add the edge $\{v_i, w_j\}$. For every variable x_i appearing in C_j negated as $\bar{x}_i$, we add an edge from.
4. For every variable x_i, we add an edge between v_i and $\bar{v}_i$.

Let us try to explain the intuition of the construction so far. We have vertices for each assignment to each variable. We also have vertices for the clauses. We have edges between (vertices representing) clauses and the (vertices representing the) single-variable assignments that satisfy the clauses. The total number of variables is $O(m+n)$, and the number of edges is $O(p)$.

Now consider any satisfying assignment for f. We will construct a dominating set S with n vertices as follows. If the satisfying assignment has $x_i = \texttt{true}$, then we include the corresponding vertex v_i. If the satisfying assignment has $x_i = \texttt{false}$, then we include the corresponding vertex $\bar{v}_i$. So S has n vertices, with each vertex corresponding to a variable's assignment. To see that this is a dominating set, we observe that:

1. For every clause-vertex w_j corresponding to clause C_j, the vertex corresponding to a variable satisfying C_j will dominate w_j.

2. For each variable-vertex v_i or $\bar{v}_i$, the fact that S takes either v_i or $\bar{v}_i$ ensures that both v_i and $\bar{v}_i$ are dominated.

Conversely, suppose our graph has a dominating set S of size n. Ideally S would contain exactly one of the vertices, v_i or $\bar{v}_i$, for each variable x_i, and form a satisfying assignment accordingly. However S could contain neither v_i or $\bar{v}_i$, or both v_i and $\bar{v}_i$. So it is not clear that a dominating set of size n implies a satisfying assignment (and the reader might think of some simple counterexamples).

So to summarize the current state of affairs: our construction is such that if f has a satisfying set, then the graph has a dominating set with (at most) n vertices. However, a dominating set with n vertices does *not* imply a satisfying assignment for f. We ask the reader how to improve the construction to complete the proof.

We would like to force S to take at least one of v_i or $\bar{v}_i$ for each variable x_i. If so, and S also has size (at most) n, then it must take exactly one of these vertices for each x_i. How can we force every dominating set to take at least one of v_i or $\bar{v}_i$? We can introduce another auxiliary variable, which we denote u_i, that can only be dominated by v_i or $\bar{v}_i$ (or itself). So we add the following steps to our construction.

5. For very variable x_i, we create a (third) vertex u_i.
6. We add the edges $\{u_i, \bar{v}_i\}$ and $\{u_i, \bar{v}_i\}$.

The introduction of u_i means that we have to have either u_i, v_i or $\bar{v}_i$ in any dominating set. Moreover, if a dominating set S has u_i, it can always be replaced with v_i and $\bar{v}_i$ while keeping S a dominating set.

Now, we can complete the proof. As before, a satisfying assignment to f maps to a dominating set of size n. Now, suppose S is a dominating set with at most n vertices. We know that S includes at least one of v_i, $\bar{v}_i$, or u_i for each i, in order to dominate u_i. Whenever S contains some u_i, we replace it with v_i or $\bar{v}_i$ (either one works); now we know that S includes at least one of v_i or $\bar{v}_i$. On the other hand, $|S| \leq n/2$ implies that S contains exactly one of v_i or $\bar{v}_i$. Consider the assignment $x_1, \ldots, x_n$ corresponding to the vertices in S. For each clause C_j, the corresponding variable w_j is dominating by some v_i or $\bar{v}_i$ in S; the corresponding assignment ($x_i =$ `true` or $x_i =$ `false`) satisfies C_j. Thus we have a satisfying assignment.

To summarize, we have a dominating set of size n iff f is satisfiable. Our auxiliary graph is polynomial in the size of f, and the construction takes polynomial time. A polynomial time algorithm that computes the minimum dominating set can, in particular, decide if there is a dominating set of size n in the auxiliary graph of a given CNF $f(x_1, \ldots, x_n)$. We would obtain a polynomial time algorithm for CNF SAT by first converting the algorithm to the graph described above and then computing a dominating set.

Dominating set for intervals and trees. What about dominating set in interval graphs, or in trees? We leave these questions to the reader in exercise 13.6.

13.4 Additional notes and materials

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture 13 materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.

- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture 15 materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

13.5 Exercises

[Eri19, Chapter 3] also has several exercises about dynamic programming on trees.

Exercise 13.1. Theorem 13.3 proved that a polynomial time algorithm for 3-coloring implies a polynomial time algorithm for SAT. Prove that for any $k > 3$, a polynomial time algorithm for k-coloring implies a polynomial algorithm for SAT.[4]

Exercise 13.2. Let k be fixed. Suppose you had access to a polynomial time algorithm for the decision version of k-coloring. Use this algorithm to obtain a polynomial time algorithm that obtains a k-coloring of a graph (if one exists).

Exercise 13.3. Recall that the k-coloring problem is to decide if there is a vertex coloring with at most k-colors. The subset k-coloring problem also specifies for each vertex v a subset $S_v \subseteq [k]$, and the k-coloring is restricted to have each vertex v colored by a color in S_v. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. The k-coloring problem in trees.

[4] It might be helpful to focus on $k = 4$ to develop your ideas.

2. The k-coloring problem for intervals.

3. The subset k-coloring problem in trees, for fixed k.

4. The subset k-coloring problem for intervals, for fixed k.

Exercise 13.4. Let $G = (V, E)$ be an undirected graph. We say that a set of vertices is *almost independent* if each vertex $v \in S$ has at most one neighbor in S.[5] Consider the problem of computing the maximum cardinality of any almost independent set of vertices. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 13.5. [6]Let $G = (V, E)$ be an undirected graph with m edges and n vertices, and $k \in \mathbb{N}$. Recall that a *vertex k-coloring* (a.k.a. a *graph k-coloring*) is an assignment of colors (out of $\{1, \dots, k\}$) to each $v \in V$ so that no two adjacent vertices have the same color. When such an assignment exists, G is said to be *k-colorable*. Now suppose that for each color $i = 1, \dots, k$, and each vertex $v \in V$, there is a positive *assignment cost* $c(i, v) > 0$ for assigning color i to vertex v.

The following problem is called the *minimum cost graph coloring (MCGC)* problem. Given G and k as input, where $k \geq 3$, the goal is to find the minimum total assignment cost of any vertex k-coloring of V (which is $+\infty$ if no such coloring exists).[7] We consider the problem both for general graphs and trees.

1. (5 points) Suppose $G = (V, E)$ is a graph with m edges and n vertices. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. (5 points) Suppose $G = (V, E)$ is a tree with n vertices. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

[5]Two vertices u and v are neighbors if they are connected by an edge.

[6]You may assume as fact that a polynomial time algorithm for k-coloring implies a polynomial time algorithm for SAT; this fact is otherwise left to the reader as exercise 13.1.

[7]Technically this would be the "infimum" of all assignment costs of (valid) vertex k-colorings.

Exercise 13.6. Recall the dominating set problem from Section 13.3. Here we will consider the weighted version where the vertices are given positive weights, and the goal is to compute the minimum weight dominating set. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

1. The minimum weight dominating set problem for intervals, *with the additional assumption that* no two intervals are nested.

   To state it more precisely: the input consists of n weighted intervals $\mathcal{I}$. The non-nested assumptions means that for any two intervals $I, J \in \mathcal{I}$, we never have I contained in J or J contained in I.

   The goal is to compute the minimum weight subset $S \subseteq \mathcal{I}$ of intervals such that every interval in $\mathcal{I}$ is either in S or overlaps some interval in S.

   - (For 1 pt. extra credit) Extend your algorithm to general intervals.[8]

2. The minimum weight dominating set problem in trees.

Exercise 13.7. Let $I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]$ be a collection of n intervals on the real line, and let $p_1, \ldots, p_m \in \mathbb{R}$ be a collection of m points on the line with positive weights $w(p_i) > 0$. A point p_i *hits* an interval I_j if $p_i \in I_j$. A subset of points $p_{i_1}, \ldots, p_{i_k}$ is a *hitting set* for $I_1, \ldots, I_n$ if every interval I_j is hit by some point p_{i_h} in the subset. We assume that $p_1, \ldots, p_m$ itself is a hitting set. We also assume for simplicity that all the points p_i are distinct. The goal is to compute the minimum weight of any hitting set of points. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 13.8. Given a graph $G = (V, E)$, a *clique* is a set of vertices $S \subseteq V$ such that every pair of vertices in S is connected by an edge. (This is sort of like the opposite of an independent set.) When the vertices are weighted by $w : V \to \mathbb{R}_{>0}$, one can ask for the maximum weight clique in the graph.[9]

[8] Of course, anyone who has already solved the general case automatically solves the special case where no two intervals are nested.

[9] Note: we updated the problems so that one only needs to return the maximum weight of any clique, as opposed the actual clique itself, for the sake of simplicity.

1. Consider the problem of finding the largest weight of any clique from a set of weighted intervals. (Here a clique of intervals is a set of intervals where every two intervals are overlapping.) The input consists of n intervals $I_1 = [a_1, b_1], \ldots, I_n = [a_n, b_n]$, and you can assume that you can look up the weight of an interval in constant time. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. Consider the problem of finding the largest weight of any clique in a tree. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

3. Consider the problem of finding the largest weight of any clique in a graph G. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.[10]

Exercise 13.9. Here we consider a special case of independent set that we will call *k-neighbors independent-set*, for a fixed parameter $k \in \mathbb{N}$. (e.g., $k = 5$.) The input is an undirected graph $G = (V, E)$ with m edges and n vertices, with the following property: every set $S \subseteq V$ has at most k neighbors of S outside of S.[11] The goal is to compute the maximum size independent set. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 13.10. Let $I_1, \ldots, I_n$ be a collection of n intervals on the real line, and let $p_1, \ldots, p_m \in \mathbb{R}$ be a collection of m points on the line. Let each interval I_j be assigned a positive weight $w(I_j) > 0$.

An interval I_j *covers* a point p_i if $p_i \in I_j$. A collection of intervals $I_{j_1}, \ldots, I_{j_k}$ is an *interval cover* of $p_1, \ldots, p_m$ if every point p_i is covered by at least one interval I_{j_h} in the collection. We assume that the collection of all the intervals $I_1, \ldots, I_n$ is an interval cover of $p_1, \ldots, p_m$. We also assume that all the points are distinct.

Consider the problem of computing the minimum weight of any interval cover of $p_1, \ldots, p_m$. For this problem, either (a) design and analyze a polynomial time

[10] Hint: Consider the "complement graph" $G' = (V, E')$ where there is an edge $\{u, v\}$ iff $\{u, v\}$ *is not* an edge in G.

[11] Here a vertex v is a *neighbor* of S if it is connected by an edge.

algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 13.11. Can't we all just get along? You want to throw a big party to celebrate finishing midterm 2, but not all of your friends get along. So you resolve to invite as many friends as possible while making sure there are no problems within the party.

Formally, we assume that you have n friends indexed by $1, \dots, n$. Your input consists of a list L of length m of all pairs of friends who don't get along. The goal is to compute the largest cardinality (of any) subset of friends, $S \subseteq [n]$, that does not contain any pair of friends in the list L.

1. (5 points) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

2. (5 points) Now suppose there are no "cycles" among the pairs of enemies: that is, there is no sequence of friends $f_0, \dots, f_{k-1}$, where $k \geq 3$, such that f_i does not get along with $f_{i+1 \mod k}$ for all i. For this setting, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 13.12. Fast forward 10 years and your LLM company GNAGI[12] is taking off. Your company's main product is an API where customers submit text generation tasks, but due to the Great GPU Panic you only have 2 machines and can't service all of the requests. So you've set up a bidding system, where each customer submits a job (with time specifications described below) along with a bid. The high-level task is to choose which of these jobs to schedule on your machines to maximize the total amount of money you make.

The input consists a sequence of n jobs $J_1, \dots, J_n$. Each job i consists of a

- Time of arrival $T_i \geq 0$, a
- Duration $D_i > 0$, and a
- Bid $B_i > 0$.

For the sake of concreteness, we'll say that your input is given as three arrays $T[1..n]$, $D[1..n]$, and $B[1..n]$, listing the job arrival times, durations, and bids, respectively.

You get to choose which of the jobs to take on, and which machines to schedule them on. As mentioned above, you have 2 machines. Each machine can only work on

[12] GNAGI stands for GNAGI Not AGI.

one job at any given time. If you assign job i on machine M_j (where $j \in \{1, 2\}$), then the job occupies M_j for D_i units of time starting from time T_i, and you make a profit of B_i dollars.

Consider the problem of computing the maximum profit (i.e., sum of bids) from scheduling jobs onto the 2 machines. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Sample solution.

Problem. Let $T = (V, E)$ be an undirected tree with positive vertex weights $w : V \to \mathbb{R}_{>0}$. Consider the problem of computing the weight of the maximum weight independent set in T. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Solution. We root T arbitrarily and henceforth refer to the root, children, subtrees, etc., accordingly.

1. *Recursive spec.* For a vertex $v \in V$, we define

   `MWIS(`v`)` = the weight of the maximum weight independent set in the subtree rooted at v.

   `MWIS-no-root(`v`)` = the weight of the maximum weight independent set in the subtree rooted at v among those that exclude v.

2. *Recursive implementation.*

   `MWIS(`v`)`

   1. Return the maximum of:

      `/* The maximum weight independent set containing v. */`

      A. $w(v) + \sum_{\text{children } w \text{ of } v}$ `MWIS-no-root(`v`)`

      `/* The maximum weight independent set excluding v. */`

      B. $\sum_{\text{children } w \text{ of } v}$ `MWIS(v))`

   `MWIS-no-root(`v`)`

   1. return $\sum_{\text{children } w \text{ of } v}$ `MWIS(w)`

3. *Solving the original problem.* We return `MWIS(`r`)` where r is the root of the tree.

4. *Say "caching" or "dynamic programming".* We cache the answers to the recursive algorithm above.

5. *Running time.* There are two subproblems for each vertex v. For fixed v, the running time is proportional to the number of children of v (plus a constant). Summing over all v counts each vertex as a child once, so we obtain a $O(n)$ running time overall.

6. *Brief proof / justification.* Fix v and consider `MWIS(`v`)`. The MWIS in the subtree rooted at v either includes v or it doesn't. If it does, then the restriction of the MWIS to each subtree of v is an independent excluding the root of that subtree. By induction on subtrees, for each subtree rooted by a child w, `MWIS-no-root(`w`)` returns the maximum weight of any independent set in the subtree excluding w. Line `(1.A)` then ensures that we return at least the weight of this set.

Now suppose the MWIS in the subtree rooted at v does not include v. Then the restriction of the MWIS to each subtree of v is an independent set of that subtree. By induction on subtrees, for each subtree rooted by a child w, `MWIS(`w`)` returns the maximum weight of any independent set in the subtree. Line `(1.A)` then ensures that we return at least the weight of this set.

Consider now `MWIS-no-root(`v`)`, and consider the MWIS S of the subtree rooted at v, among those excluding v. For each subtree w, the restriction of S to the subtree rooted by w is an independent set in the subtree rooted at w. By induction on subtrees, `MWIS(`w`)` returns the weight of the MWIS in this subtree. So the sum we return gives the MWIS overall.

Remark 13.9. In the lecture notes we added a flag to `MWIS`; this time I thought it might be clearer to just make two functions where the intent of the flag is declared in the name of the function.

Sample solution.

Problem. Let $\mathcal{I}$ be a family of n intervals with positive weights $w : \mathcal{I} \to \mathbb{R}_{>0}$, and consider the problem of computing the weight of the maximum weight independent (i.e., non-overlapping) subset of $\mathcal{I}$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Solution. We sort the intervals in increasing order of left end-point (in $O(n \log n)$ time) and let $I_i = [a_i, b_i]$ refer to the ith interval in increasing order of left end-point a_i.

1. *Recursive spec.* For $i = 1, \ldots, n+1$, we declare

   `MWIS(`i`)` = the maximum weight independent set of intervals $I_i, \ldots, I_n$. ($i > n$ implies the empty collection of intervals.)

2. *Recursive implementation.*

   `MWIS(`i`)`

   `1.` If $i > n$ then return 0.

   `2.` Return the maximum of

   `/* Include` I_i`. */`

   `A.` $w(I_i)$ + `MWIS(`$j+1$`)` where j is the last index of any interval I_j overlapping I_i.

   `/* Don't include` I_i`. */`

   `B.` `MWIS(`$i+1$`)`

3. *Say "caching " or "dynamic programming".* We apply dynamic programming to the above recursive implementation.

4. *Extracting the original problem.* The overall solution is given by `MWIS(1)`.

5. *Running time.* There are $O(n)$ subproblems. Naively, identifying the interval I_j in step `(2.A)` would take $O(n)$ time in a loop. However, having already sorting the intervals by left-endpoint, we can use binary search to identify I_j in $O(n \log n)$ time. Overall (with dynamic programming) the algorithm takes $O(n \log n)$ time.

6. *Proof by induction.* Consider `MWIS(`i`)`. In the base case $i > n$ in which case there are no intervals and the answer is 0. Suppose $i \leq n$.

Consider the (true) MWIS of $I_i, \ldots, I_n$. The MWIS either includes I_i or doesn't. Suppose first that the MWIS includes I_i. Since I_i overlaps I_j and the intervals are indexed in increasing order of left endpoint, the intervals in between $(I_{i+1}, \ldots, I_{j-1})$ also overlap I_i. So in particular the MWIS does not include any of these intervals either, and is an independent subset of $I_{j+1}, \ldots, I_n$. By induction on the number of intervals, `MWIS(`$j+1$`)` returns the weight of the MWIS of $I_{j+1}, \ldots, I_n$. So $w(I_i) +$ `MWIS(`$j+1$`)` is (at least) the weight of the true MWIS, and is one of the two options considered in step `(2)`.

Now suppose the MWIS does not include I_i. Then it is the MWIS of $I_{i+1}, \ldots, I_n$. By induction on the number of intervals, `MWIS(`$i+1$`)` returns this weight, hence so does `MWIS(`i`)`.

Chapter 14

Polynomial time is a flat circle

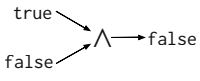
14.1 Logical circuits

Boolean circuits are a generalization of Boolean formulas. Recall that a Boolean formula can be expressed as a rooted tree, where each internal node is associated with a logical operation that is applied to the Boolean outputs from its children in the subtree. Circuits are similar except they do not the logic is more generally arranged in a directed acyclic graph.

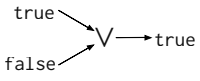
Likely the reader has encountered Boolean circuits before and we briefly review the components. Figure 14.2 gives an example of a circuit for adding two bits. The picture is almost self-explanatory but let us define circuits formally away. At a high level, Boolean circuits are composed of *logical gates* arranged (as nodes) in a *directed acyclic graph*. We first define the gates and explain their graphical arrangement afterwards.

Each gate corresponds to either one bit of input or one of the three Boolean operators. Each gate takes at most two input bits, and outputs one bit (which may be propagated to any number of other gates). The basic types of gates are:

1. An input gate, which models a single bit in the input, and outputs that bit.
2. An "and-gate" (or a "conjunction gate", or simple an "and" or a "conjunction"), takes as input the output bits from two other gates, and outputs the $\wedge$ of these two inputs.



3. An "or-gate" (or a "disjunction gate", etc.) takes as input the output bits from two other gates, and outputs the $\vee$ of these two inputs.



4. A "not-gate"' (or a "negation gate", etc.) takes as input one output bit from another gate, and outputs the negation of that bit.

Physically, we imagine each gate logically evaluating its inputs, producing a bit that is propagated to further gates ahead.

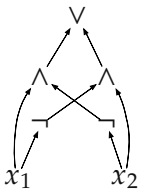


Figure 14.1: A circuit computing the exclusive-or of two bits (at the top gate).

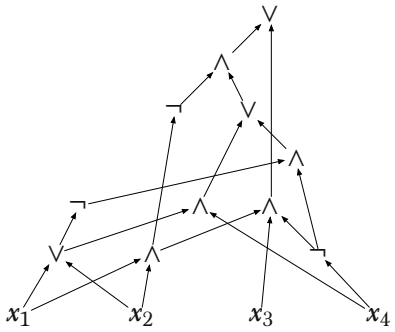


Figure 14.2: A circuit for adding two bits. Here, the gate at the top outputs true iff $x_1 + x_2 = x_3 x_4$.

We can model the dependencies between gates by a directed graph, where each gate is a node, and we have a directed edge from one gate x to gate y when x is an input to x. The dependencies are required to be *acyclic*. Here we recall that a *cycle* in a directed graph is a sequence of edges $(x_1, x_2), (x_2, x_3), \ldots, (x_k, x_1)$ where the endpoint of one edge is the starting point of the next edge in the sequence, that starts and ends at the same vertex. A cycle would not be well-defined in a circuit: if gate x depends on y, and y depends on z, and z depends on x, then which gate is to be evaluated first? So to avoid these issues we disallow cyclic dependencies by definition[1].

In the simplest model, we assume there is one gate in particular whose output we are externally interested in, as a function of the inputs; we call this the *output gate*. We will focus on a single output gate for simplicity; in general, one can have many output gates. Now we consider the problem of *circuit satisfiability*: given a Boolean circuit, is there an assignment to its input gates that makes the output gate `true`?

While a Boolean circuit is physically implementable, a first computational task would be able to simulate a Boolean circuit on a given input. We are now equipped to do so elegantly. Since a logical gates form a directed acyclic graph, we can topologically

[1]One could consider evaluation *over time*, but this is a separate discussion.

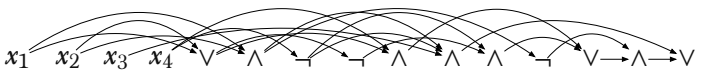


Figure 14.3: The addition circuit (cf. Fig. 14.2) redrawn to list the logical gates in topological order from left to right. Observe that all directed edges go from left to right.

sort the gates. We can then evaluate the gates in topological order. Indeed, when evaluating a particular gate in this order, all the gates it depends on, which come earlier in the order, have already been evaluated.

So much for evaluating a circuit. It is slightly more challenging to evaluate a circuit than a Boolean formula, but topological sort makes it much easier. Importantly, we have also established that circuit satisfiability is also a search problem: if, indeed, a circuit is satisfiable, then the satisfying inputs provide a proof that can be verified in polynomial time. Clearly circuit SAT is more general than Boolean satisfiability, so circuit SAT must be at least as hard boolean SAT. But is circuit SAT strictly *harder* than boolean SAT?

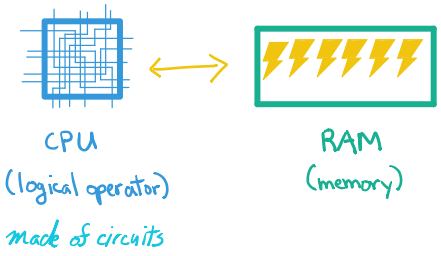
We can show the answer is no by converting circuits into boolean formulas. Actually we will convert circuits into CNF form. The construction is the same as in our previous proof converting Boolean formulas into CNF form, except applied to a DAG rather than a tree (see Section 9.A). We briefly recap the main ideas. Recall that we can express assignment (e.g., "$z = x$") in CNF; moreover, we can express phrases like "$z = x \wedge y$", "$z = x \vee y$" and "$z = \neg x$" in CNF. For each non-input gate, in topological order, we introduce a variable for that gate and "assign" it to the value obtained by evaluating its inputs, using our CNF gadgets. We combine the CNF's generated by all the gates in a conjunction, and we also write append "$\wedge z$" to the whole formula, where z represents the output gate, to require that the whole circuit is satisfied.

Theorem 14.1. *There is a polynomial time algorithm for SAT iff there is a polynomial time algorithm for circuit SAT.*

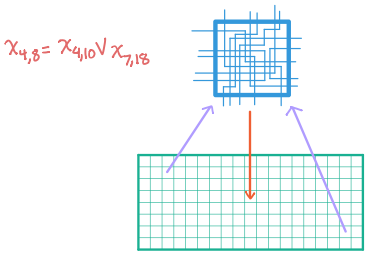
14.2 Every program is a circuit

Let us argue, at a high and informal level, that circuits are universal in the following sense: every polynomial time algorithm on a computer corresponds to a polynomial size circuit. The argument can be made rigorous but then we would want to establish a more formal model of computation than we are currently working with, which is besides the point.

For the sake of this discussion, a "computer" refers to a combination of a CPU (central processing unit) and RAM (random access memory).

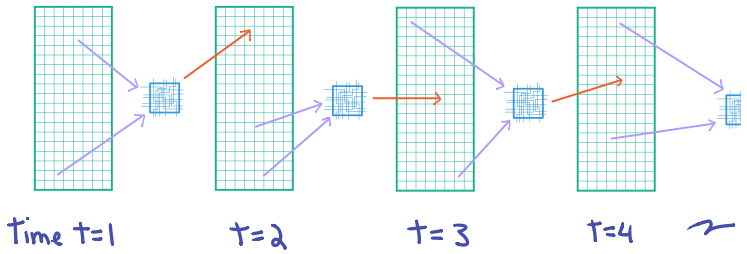


The RAM consists of an array of memory that can be used for any purpose. The CPU is a mechanical device ultimately composed of a fixed number of circuits to execute logic. Fix an algorithm that is being executed on this computer (taking its input from RAM, and writing out to RAM). Each step, the CPU reads in a finite (say, 2) number of locations from RAM, does some logical computation such as addition, and writes the data out to another location in RAM. This single step resembles a circuit, where each location in RAM corresponds to a node in a circuit, and the CPU combines the outputs of some nodes and stores the answer at another. The fact that the CPU is a combination of gates, instead of a single gate, is insignificant (in this theory), since ultimately the size is a constant.



The real difference between the CPU model and a circuit is that the CPU can write different results to the same place in memory over time. Such a dynamic can create cycles in the circuit, so to speak, which is expressly forbidden by definition of circuits. The key idea is to break these cyclic dependencies is as follows.

To break cycles, we parameterize everything by time. That is, we have nodes of the form (i, t), where i is an index in RAM, and $t \in \mathbb{N}$ is a time step. In time step t, if the CPU reads in indices $i_1$and i_2 and writes to index j, then we will have a circuit that takes as input the nodes (i_1, t) and (i_2, t) and writes the output to (j, t). We also set $(k, t) = (k + 1, t)$ for all indices $k \neq j$, since none of these locations in memory change. This parameterization – building layers of memory – is very similar to how we dealt with negative edge lengths in developing the Bellman-Ford algorithm.



Thus we transform the computation over time into a circuit by indexing by time. How large is this circuit? In this construction, it is (roughly) bounded above by

$$(\text{(size of CPU)} + \text{(size of RAM)}) \times (\#\text{ time steps}).$$

For a polynomial time algorithm, the number of time steps is polynomial in the size of the input. The CPU has constant size. We also have

$$\text{(size of RAM)} \leq \text{(input size)} + (\#\text{ time steps})$$

since we can write to only one location in RAM each iteration anyway. Thus the overall size of the circuit is bounded above by

$$O\Big((\#\text{ time steps})^2\Big).$$

In particular, for a polynomial time algorithm, the quantity above is a *polynomial in the input size.* Somewhat informally, we have shown that *every polynomial time algorithm corresponds to a polynomial size circuit.*

14.3 Cook '71, Levin '73: Every search problem is a Circuit-SAT problem

By now we have encountered many problems that look simple, but are actually very hard to find an efficient algorithm for. Some examples we have discussed include SAT, Subset Sum, Independent Set, and the Hamiltonian path problem. All of these problems are *search problems.* More formally, in the decision formulation of each problem, if the true answer is `true`, then there exists a polynomial sized proof that can be verified in polynomial time. For example:

1. In SAT, if a formula $f : \{0,1\}^n \to \{0,1\}$ in CNF is satisfiable, then a proof is given by a satisfying assignment $x \in \{0,1\}^n$. It is easy to evaluate the formula f over x and verify that indeed $f(x) = \texttt{true}$.

2. If an instance of subset sum is feasible, then there is a proof given by the subset of numbers that add up to the target. It is easy to verify that a given set of numbers is a subset of the input numbers, and that they add up to the target.

3. In the independent set problem, if there exists an independent set of size k in a given graph, then a proof is given by the satisfying set of vertices. It is easy to verify that a set of vertices is independent in polynomial time.

4. In Circuit-SAT, if a circuit is satisfiable, then a proof is given by a satisfying assignment to the inputs, $x \in \{0,1\}^n$. We can evaluate the circuit of x with topological sort and verify that the output is `true`.

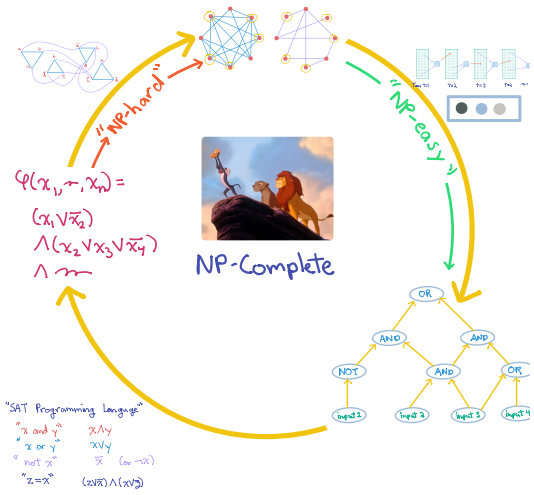
Any problem with a polynomial time prover is in *nondeterministic polynomial time*. Omitting a more rigorous definition, this is really to say that if we had an unlimited number of CPUs, we can solve the problem in polynomial time (by having each CPU guess a different solution and then verifying if it works).

Definition 14.2. *A decision problem said to be in* nondeterministic polynomial time *(abbrev.* NP*) if there is a polynomial time algorithm $f : \{0,1\}^* \to \{0,1\}$, called the* verifier *or* prover*, such that an instance of the problem is satisfiable iff $f(x) =$* `true` *for some string x of length polynomial in the size of the input. In this case, the string x is called a* certificate*.*

Consider *any* problem in nondeterministic polynomial time, which we will call P. This includes SAT, subset sum, and independent set, but here we assume nothing other than that there is a polynomial time prover, which we will call f. Now recall our earlier discussion on circuits. Any polynomial time algorithm can be converted into a polynomial size circuit. In particular, let C_f denote the polynomial size circuit corresponding to our prover f. By definition, any instance of problem P is satisfiable iff there is a polynomial size certificate x such that $f(x) =$ `true`. But there exists a certificate for f iff the circuit C_f is satisfiable. Thus, if we had a polynomial time algorithm for Circuit-SAT, then we *automatically* have a polynomial time algorithm for P - despite knowing nothing about P beyond its membership in NP. In turn, Circuit-SAT can be solved by SAT. Thus we have obtained (somewhat informally) the landmark *Cook-Levin theorem*.

Theorem 14.3 (Cook [Coo71] and Levin [Lev73])**.** *A polynomial time algorithm for SAT implies a polynomial time algorithm for any problem in nondeterministic polynomial time.*

Take a step back and absorb the theorem. We now know that SAT can solve any problem in NP. We had some hints of this when we first observed that SAT could simulate programming primitives like assignment and if-else statements. We also have a growing collection of problems for which a polynomial time algorithm for



this problem gives a polynomial time algorithm for SAT. In light of the Cook-Levin theorem, this also implies a polynomial time algorithm for any problem in NP. For this reason, any problem for which a polynomial time algorithm would imply a polynomial time algorithm for SAT is said to be *NP-Hard*.

14.4 Karp'72: The theory of NP-Completeness.

As we have seen, many basic problems - such as SAT, Independent Set, and Subset Sum - are both NP-Hard and in NP. As such, *a polynomial time algorithm for any one of them implies a polynomial time algorithm for all of them.* Problems that are both NP-Hard and in NP are called *NP-complete*.

So what does it mean for us mere mortals, who simply want to solve some useful and innocuous-looking problems? Take for example subset sum: we would really like to find a polynomial time algorithm or else rule out the possibility. The theory of NP-Completeness does not answer this definitively. But it heightens the stakes for subset sum substantially – any NP-Complete problem solves all NP-Complete problems. We have a sort of explanation.

This grand scheme was put together by [Kar72] who, building on the Cook-Levin theorem above, compiled a list of 21 NP-Complete problems and observed that they are all equivalent as far as obtaining a polynomial time algorithms. The list includes subset sum, Hamiltonian cycle, and graph coloring, among many others (cf. Chapter D). For each problem [Kar72] showed that the problem is in NP, and can solve some other NP-Hard problem by a polynomial time reduction. Quoting [Kar72], these reductions

> ...strongly suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually.

One cannot overstate the influence of [Coo71; Lev73; Kar72] on algorithm design. Karp's reductions opened a floodgate of research demonstrating problems of interest to be NP-complete. In fact it seems that a very limited set of problems can be solved exactly. We have seen some of these problems, and will see more in subsequent chapters. By 1979, Garey and Johnson [GJ79] had compiled an entire book on NP-Hard problems and reductions (organizing a burgeoning family of academic papers), and countless more problems have been shown to be NP-Hard since.

How do we move beyond NP-Completeness? These problems remain of interest, however difficult. One way is via *approximation algorithms*, where algorithms do obtain solutions that are in a rigorous sense guaranteed to be close to the optimum. Even this has some limits, though: the PCP theorem asserts that it is NP-Hard to satisfy a $(7/8 + \epsilon)$-fraction of the clauses in 3SAT; note that a $(7/8)$th approximation is obtained trivially from randomly assigning variables. From here one develops *approximation preserving reductions*, where an α-approximation for some problem X implies a better-than-$(7/8)$th-approximation for 3SAT, which is NP-hard.

Another way is to try to add more structure to the problem. The additional structure makes the problem more narrow and possibly excludes reductions from SAT. We have seen many problems, like coloring or independent set or Hamiltonian path, that are NP-hard in general graphs, but have polynomial algorithms for limited classes of graphs like trees, DAGS, and interval graphs.

We should also point out that it does not violate NP-Complete theory to solve a specific instance of an NP-Complete problem. Many instances of NP-Complete problems are solved with heuristics and care in practice. NP-Completeness is only concerned with the prospects of a *general* algorithm for an NP-Complete problem that is guaranteed to solve any instance in polynomial time.

Again, the theory of NP-Completeness does not entirely rule out a polynomial time algorithm for NP-complete problems. Whether we can solve NP-Complete problems - equivalently, whether we can solve SAT - remains an outstanding open question and a humbling reminder of our limited understanding of computing.

14.5 Recap

Perhaps the biggest point of emphasis in the lectures to this point has been in obtaining polynomial running times for algorithmic problems. It may be surprising to find out that there are simple looking problems, like Hamiltonian path, that may never have polynomial time algorithms. Funny enough, given the overwhelming number

of NP-Hard problems uncovered in the last 50+ years (e.g., [GJ79]), it is arguably surprising that we can solve *any* combinatorial problems (exactly) in polynomial time.

As mentioned above, we do not have a compact description of why some problems are tractable and some are not. But we can still speculate and grasp at some patterns based on our own experience. For seemingly any search problem, one can propose a recursive, essentially brute-force type of algorithm that exhausts all the combinations. These recursive algorithms do work, with proofs given by induction, but such an approach is often exponentially slow, at least initially. In *some cases*, the induction-hypothesis / recursive-spec breaks down the original problem into only *polynomially many subproblems*. The latter allows us to leverage memory — saving all the answers — and obtain a polynomial upper bound on the running time that multiplies the polynomial time per problem with the polynomial number of subproblems.

In some cases it is easy to identify good subproblems. One heuristic is to prepend "sub-" to the structure at hand, like with *subsequences* or *subtrees*. Another cue is the word "order": like *ordering intervals* by left endpoint, or the *topological ordering* of vertices in a DAG. Of course there are many subtleties too. Obtaining a truly polynomial number of subproblems can be tricky (e.g., subset-sum). Sometimes the induction hypothesis / underlying DAG is more implicit; e.g., the DAG of strongly connected components for some reachability problems, or the DAG of tight edges for the problem of single-source shortest paths with positive edge weights.

Dynamic programming has solved many problems for us, to the extent that the technique nearly characterizes everything that is tractable (based on what we've seen so far). But the theory of NP-Completeness crushes the dream of conquering all of our problems with dynamic programming. Given this opposition, it is ironic that the key idea of the proof of the Cook-Levin theorem – looking at the "layered circuit" induced by a (model) computer *over time* – so closely resembles the "layered-graph" trick from Bellman-Ford, a prototypical dynamic programming algorithm. Karp's reductions – the other nail in the coffin – cruelly exploits our other main algorithmic hammer: *don't change the algorithm, change the input!* We can change a SAT formula into a Hamiltonian path problem, or into a subset sum problem, or into an independent set problem. It's unbelievable.

14.6 Additional notes and materials

As mentioned previously, we refer to [AB09; Sip97; Wig19] for more on P vs NP question. Recently, to mark the 50th anniversay of the Cook-Levin theorem, Cook, Karp, and Levin gave a wonderful (virtual) presentation reflecting on these developments [CKL+21].

The content in this chapter can also be found in [Eri19, Chapter 12], [DPV08, §8.1], [KT06, Chapter8], and [Sch, Chapter 6]. We also recommend the video lectures by Sipser on NP-Completeness and the Cook-Levin theorem [Sip20b; Sip20a] for those who are interested. Sipser's proof of the Cook-Levin is more rigorous than the one presented here.

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture. (There are some issues with the audio.)

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

14.7 Exercises

Exercise 14.1. In *Minesweeper 2: On Graphs*, we have an undirected graph $G = (V, E)$ whose vertices are partitioned into *open* vertices V_{OPEN} and *hidden* vertices V_{HIDDEN}. Hidden vertices may contain mines. Each open vertex tells you how many of its hidden neighbors are mines.

Each open vertex $v \in V_{\text{OPEN}}$ has a label $\ell(v) \in \mathbb{Z}_{\geq 0}$. A *mine placement* is a function $M : V_{\text{HIDDEN}} \to \{0, 1\}$. A mine placement is *consistent* if for every open vertex v,

$$\ell(v) = \sum_{\substack{w \in V_{\text{HIDDEN}} \\ \{v,w\} \in E}} M(w).$$

That is, each label equals the number of adjacent hidden vertices containing a mine.

Given $G = (V_{\text{OPEN}} \sqcup V_{\text{HIDDEN}}, E)$ and $\ell : V_{\text{OPEN}} \to \mathbb{Z}_{\geq 0}$, consider the problem of determining whether there is a consistent mine placement. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 14.2. Least romantic walk ever. You and your significant other have decided to go to a local nature preserve for a romantic walk. Together you picked a handful of trails that you would particularly like to traverse. But you're also a little pressed for the time because you need to practice exercises for the upcoming CS 381 final exam. Your goal is to plan for the shortest walk that covers all of your selected trails.

We model this with an undirected graph $G = (V, E)$ with positive edge lengths $\ell : E \to \mathbb{R}^+$, a starting vertex s, and a set of required edges $R \subseteq E$. Each edge models a trail, and R is the set of trails you want to visit. Call a walk *romantic* if it includes every edge in R. Consider the problem of computing the length of the shortest romantic walk starting and ending at s. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Chapter 15

Polynomial lower bounds

At the moment, we do not have any running time for SAT faster than $O(2^n \operatorname{poly}(m, n))$. The *strong exponential time hypothesis* (SETH) conjectures that it is impossible to solve SAT in $O\Big(2^{(1-\epsilon)n} \operatorname{poly}(m)\Big)$ time for any fixed $\epsilon > 0$. Of course we do not know if this conjecture is true. But giving it a name makes a landmark in the theoretical landscape around which other ideas can organize and new connections are established, as we will see.

15.1 Disjoint sets

Let $\mathcal{A} = \{x_1, \ldots, x_n\} \subset \{0,1\}^m$ and $\mathcal{B} = \{y_1, \ldots, y_n\} \subset \{0,1\}^m$ be two families of bit strings of length m. We can interpret each x_i and y_j as a subset of m, depending on which bits are set to 1. Alternatively we can think of them as vectors in $\mathbb{R}^m$. In this section we are interested in the following simple problem:

> Is there a disjoint pair of sets $x_i \in \mathcal{A}$ and $y_j \in \mathcal{B}$?

Equivalently, in terms of vectors:

> Is there an orthogonal pair of vectors $x_i \in \mathcal{A}$ and $y_j \in \mathcal{B}$?

We remind the reader that the two vectors x, y are *orthogonal* if their dot product, denoted $\langle x, y \rangle = \sum_{i=1}^{m} x_i y_i$, is 0. This problem is called the *orthogonal vectors problem* in the literature though the reader might find it easier to think in terms of disjoint sets.

The obvious algorithm is to compare every pair x_i and y_j in $O(m)$ time per comparison, giving a total running time of $O(n^2 m)$. Can one do better?

On one hand, there's not much structure to take advantage of. On the other hand, it's always hard to rule out the possibility of a clever idea.

We give the following *conditional* lower bound, which states that a substantially better dependency on n would improve the running time for SAT.

Theorem 15.1. *For all $\epsilon > 0$, a $O(n^{2-\epsilon}\operatorname{poly}(m))$ time algorithm for the orthogonal vectors problem implies a $O\Big(2^{(1-\epsilon/2)n}\operatorname{poly}(m)\Big)$-time algorithm for SAT.*

Proof. Let f be a CNF with n variables and m clauses. We assume[1] that n is even and split the variables into two equal halves. For ease of notation, let us name the variables

$$f\Big(x_1, \dots, x_{n/2}, y_1, \dots, y_{n/2}\Big).$$

Alternatively we write $f(x, y)$ where $x \in \{0,1\}^{n/2}$ and $y \in \{0,1\}^{n/2}$.

Fix $x, y \in \{0,1\}^{n/2}$. Then the combined assignment (x, y) satisfies a clause C_i iff some literal among the x_j's, or some literal among the y_k's, makes it true. We say that x satisfies C_i in the former case, and that y satisfies C_i in the latter. (Of course, both can occur simultaneously.) For example, the clause

$$(x_j \vee \bar{y}_k)$$

is satisfied by any $x \in \{0,1\}^{n/2}$ such that $x_j = \texttt{true}$, or by any $y \in \{0,1\}^{n/2}$ such that $y_k = \texttt{false}$. We can now rephrase the satisfiability question as finding $x, y \in \{0,1\}^{n/2}$ such that every clause C_i is satisfied by either x or y.

We create two families of subsets of the clauses, $\mathcal{A}$ and $\mathcal{B}$, as follows. For each $x \in \{0,1\}^{n/2}$, we define a set

$$A_x = \{\text{clauses } C_i \text{ not satisfied by } x\} \subseteq \{C_1, \dots, C_m\}.$$

Similarly, for each $y \in \{0,1\}^{n/2}$, we define a set

$$B_y = \{\text{clauses } C_i \text{ not satisfied by } y\} \subseteq \{C_1, \dots, C_m\}.$$

We define

$$\mathcal{A} = \Big\{A_x : x \in \{0,1\}^{n/2}\Big\} \text{ and } \mathcal{B} = \Big\{B_y : y \in \{0,1\}^{n/2}\Big\}.$$

Two sets $A_x \in \mathcal{A}$ and $B_y \in \mathcal{B}$ are disjoint iff every clause is satisfied by either x or y; that is, iff $f(x, y) = \texttt{true}$.

We have $|\mathcal{A}| = |\mathcal{B}| = 2^{n/2}$. Thus a $O(n^{2-\epsilon}\operatorname{poly}(d))$ algorithm for orthogonal vectors (with n sets in dimension d) implies a

$$O\bigg(\Big(2^{n/2}\Big)^{2-\epsilon}\operatorname{poly}(m)\bigg) = O\Big(2^{((1-\epsilon/2)n)}\operatorname{poly}(m)\Big)$$

time algorithm for SAT. □

[1] for simplicity; otherwise, add a dummy variable.

15.2 Furthest pair and diameter

In the *furthest pair* problem, you are given a graph $G = (V, E)$ and two sets $S, T \subset V$. The goal is to find $s \in S$ and $t \in T$ maximizing the distance from s to t. One can consider the different combinations of unweighted and weighted graphs, and undirected and directed graphs.

For simplicity we assume that $|S| = |T| = k$ for some $k \in \mathbb{N}$, as it will be clear in hindsight how to generalize to uneven cardinalities. Run Dijkstra from every $s \in S$. This takes $O(k(m + n \log n))$ time, or $O(km)$ in unweighted graphs.

Theorem 15.2. *Suppose there is an algorithm for (S, T)-furthest pair in unweighted, undirected graphs that runs in $O(k^{1-\epsilon}m)$ time, for $\epsilon > 0$. Then the orthogonal vectors problem, with two sets of n vectors in d dimensions, can be solved in $O(n^{2-\epsilon}d)$ time.*

Proof. Let $A, B \subset \{0, 1\}^d$ be two sets of n bit strings of length d. The goal is to find $a \in A$ and $b \in B$ such that $\langle a, b \rangle = 0$. We will reduce the problem to (S, T)-furthest pair as follows. We make a layered graph with three layers.

1. The first layer has one vertex v_a for each vector $a \in A$.
2. The third layer has one vertex v_b for each vector $b \in B$.
3. In the middle layer has one vertex v_i for each index $i \in [d]$.

For each vector $a \in A$, and each index $i \in [d]$ such that $a_i = 1$, we add an edge from v_a to v_i. Similarly, for each vector $b \in B$, and each index $i \in [d]$ such that $b_i = 1$, we add an edge from v_i to v_b. For $a \in A$ and $b \in B$, if $\langle a, b \rangle \neq 0$, then there is a path of length 2 from v_a to v_b via any common index in their support. If $\langle a, b \rangle = 0$, then any path from v_a to v_b must have length at least 4. Thus, for $S = \{v_a : a \in A\}$ and $T = \{v_b : b \in B\}$, the farthest (S, T) pair gives an orthogonal pair of vectors, if one exists. □

The *diameter* of a graph $G = (V, E)$ is the maximum distance $d(s, t)$ over all pairs $s, t \in V$. It is a special case of furthest pair where $S = T = V$. One can also show obtain lower bounds for computing the diameter of a graph, which we leave as an exercise.

Question 15.3. How does this extend to negative edge weights? Are there lower bounds for Bellman-Ford?

15.3 Closest pair

In the *(bichromatic) closest pair problem*, we have two point sets $A, B \subseteq \mathbb{R}^m$ of size $|A| = |B| = n$, and a fixed norm $\|\cdot\|$, and the goal is to compute a pair $a \in A$ and $b \in B$ minimizing $\|a - b\|$.

Theorem 15.4. *If bichromatic closest pair in Euclidean norm $\|\cdot\|_2$ can be solved in $T(n, m)$ time, then orthogonal vectors can be solved in $O((m+1)^2 T(n, m) + mn)$ time. In particular, under the SETH, there is no $O(n^{2-\epsilon} \operatorname{poly}(m))$ time algorithm for all fixed $\epsilon > 0$.*

Proof. Let $A, B \subseteq \{0,1\}^m$ each of size n. For $x \in \{0,1\}^m$, $\|x\|_2^2 = \|x\|_1$, an integer between 0 and m. The polarization identity gives

$$\|a - b\|_2^2 = \|a\|_2^2 + \|b\|_2^2 - 2\langle a, b\rangle.$$

If we knew $\|a\|_2^2$ and $\|b\|_2^2$, minimizing $\langle a, b\rangle$ would be the same as maximizing $\|a - b\|_2^2$. We pin the norms by bucketing: for $i \in \{0, \dots, m\}$, let

$$A_i = \{a \in A : \|a\|_1 = i\} \quad \text{and} \quad B_i = \{b \in B : \|b\|_1 = i\}.$$

Closest pair returns a minimum, not a maximum. Flip the sign by negating B: let $B'_j = \{-b : b \in B_j\}$. For $a \in A_i$ and $b \in B_j$,

$$\|a - (-b)\|_2^2 = \|a + b\|_2^2 = i + j + 2\langle a, b\rangle,$$

so closest pair on A_i and B'_j finds the pair minimizing $\langle a, b\rangle$, and the minimum is zero iff the squared distance equals $i + j$.

Run closest pair on each of the $(m+1)^2$ bucket pairs (A_i, B'_j), padding short buckets with duplicates so each instance has n points. An orthogonal pair exists iff some bucket pair attains squared distance $i + j$. Bucketing takes $O(mn)$ time, for a total of $O((m+1)^2 T(n, m) + mn)$. □

15.4 Longest common subsequence

Recall the longest common subsequence problem. We are given two strings x and y from some finite alphabet. A *common subsequence* is a string z that appears (non-consecutively) as a subsequence of both x and y. We denote the length of the longest common subsequence of x and y by $\mathtt{LCS}(x, y)$. The longest common subsequence problem, which is very similar to the edit distance problem, can be solved with

dynamic programming in $O(mn)$ time if m is the length of x and n is the length of y. Can one do better than the straight forward dynamic program?

For simplicity we will consider longest common subsequence for bit strings $x, y \in \{0,1\}^m$ of the same length m.

Theorem 15.5. *Suppose that the longest common subsequence between two strings of length n can be solved in $O(n^{2-\epsilon})$ time for some fixed $\epsilon > 0$. Then the orthogonal vectors problem, with two sets of n vectors in m dimensions, can be solved in $O(\text{poly}(m)n^{2-\epsilon})$ time.*

Proof overview. The orthogonal vectors problem presents two sets of vectors $A = \{x_1, \ldots, x_n\}$ and $B = \{y_1, \ldots, y_n\}$, and asks for single pair $x_i \in A$ and $y_j \in B$ such that $\langle x_i, y_j \rangle = 0$. The first step of the proof is to somehow simulate the inner product $\langle x_i, y_j \rangle$ with an LCS computation. This is addressed by our first main lemma which describes a LCS-based gadget for verifying orthogonality.

Lemma 15.6. *Let $m \in \mathbb{N}$ be fixed. There are functions $h_1, h_2 : \{0,1\}^m \to \{1,2,3,4\}^M$ and an integer $T \in \mathbb{N}$ with the following properties.*

1. *Both $M, T \leq \text{poly}(m)$.*
2. *h_1 and h_2 both take $\text{poly}(m)$ time to evaluate.*
3. *For any two vectors $x, y \in \{0,1\}^m$,*

$$\mathsf{LCS}(h_1(x), h_2(y)) = \begin{cases} T+1 & \text{if } \langle x, y \rangle = 0 \\ T & \text{if } \langle x, y \rangle > 0 \end{cases}$$

We prove this lemma below in Section 15.4.1.

Lemma 15.6 lets us represent orthogonality with LCS. But the real challenge of orthogonal vectors is not computing the dot-product, but identifying a pair of vectors without an all-to-all comparison. The next lemma describes how to simulate an all-to-all-comparison with a single edit distance function.

Lemma 15.7. *Let $A_1, \ldots, A_n \in \{1,2,3,4\}^M$ and $B_1, \ldots, B_n \in \{1,2,3,4\}^M$ be two sets of n bit strings of length M, and T an integer, such that for all A_i and all B_j,*

$$T \leq \mathsf{LCS}(A_i, B_j) \leq T + 1.$$

Then in $O(Mn)$ time, one can construct strings $\bar{A}, \bar{B} \in \{1,2,3,4,5\}^{M'}$ of length $M' = O(Mn)$, and select an integer U, such that

$$\mathsf{LCS}\left(\bar{A}, \bar{B}\right) > U \text{ iff } \mathsf{LCS}(A_i, B_j) = T + 1 \text{ for some } A_i \text{ and } B_j.$$

We prove this lemma below in Section 15.4.2.

Assuming the two lemmas above to be true, then we have the ingredient to prove Theorem 15.5. Let us do so now, and address each lemma separately afterwards.

Proof of Theorem 15.5. Given two sets A and B of n bit strings in m dimensions, by combining the above two lemmas, we obtain two strings $\bar{A}$ and $\bar{B}$ of length $O(n \operatorname{poly}(m))$ such that $\mathsf{LCS}\left(\bar{A}, \bar{B}\right)$ indicates whether there is an orthogonal pair. A subquadratic algorithm for LCS, applied to $\bar{A}$ and $\bar{B}$, would imply a subquadratic (in n) algorithm for orthogonal vectors. □

15.4.1 The LCS Orthogonality Gadget

We build up the gadget described in Lemma 15.6 in three stages.

1. Simulate the product xy for single bits $x, y \in \{0, 1\}$ via LCS.
2. Simulate the dot product $\langle x, y \rangle$ for single bits $x, y \in \{0, 1\}$ via LCS.
3. Normalize step 2 so that it assigns the same value to all non-orthogonal pairs.

Encoding bits

Lemma 15.8. *There are functions $f_1, f_2 : \{0, 1\} \to \{1, 2\}^2$ such that for any two bits $x_1, x_2 \in \{0, 1\}$,*

$$\mathsf{LCS}(f_1(x_1), f_2(x_2)) = 1 - x_1 x_2.$$

Proof. Consider the following table computing the LCS between particular two bit strings.

LCS	12	22
21	1	1
11	1	0

Consequently we define

$$f_1(0) = 21, \qquad f_1(1) = 11 \qquad f_2(0) = 12 \qquad f_2(1) = 22,$$

which satisfies the claim. □

Bit strings

Lemma 15.9. *There are functions $g_1, g_2 : \{0,1\}^m \to \{1,2,3\}^M$, for $M = O(m^2)$, with the following properties.*

1. *g_1 and g_2 both take $O(M)$ to compute.*
2. *For all $x_1, x_2 \in \{0,1\}^m$,*

$$\mathsf{LCS}(g_1(x_1), g_2(x_2)) = 2m^2 + 3m - \langle x_1, x_2 \rangle.$$

Proof. By Lemma 15.8, we already have a good solution for the simplest case of a single bit, $m = 1$. The claim is that we can obtain something similar for longer strings of length m.

Let $m' \in \mathbb{N}$ be a parameter TBD. Let $C = 3^{m'}$; that is, C is the string

$$C = 3^{m'} = 333\cdots3$$

where 3 is repeated m' times. We use C as a sort of divider between coordinates. We define g_1 and g_2 by

$$g_1(x_1) = C \cdot f_1(x_1) \cdot C \cdot f_1(x_2) \cdot C \cdots C \cdot f_1(x_m) \cdot C$$
$$g_2(x_2) = C \cdot f_2(y_1) \cdot C \cdot f_2(y_2) \cdot C \cdots C \cdot f_2(y_m) \cdot C,$$

where $\cdot$ denote concatenation. These strings have length

$$(m+1)|C| + 2m = (m+1)m' + 2m.$$

Claim. *The length of the longest common subsequence is at least*

$$\mathsf{LCS}(y_1, y_2) \geq (m+1)m' + \sum_{i=1}^{m} \mathsf{LCS}(f_1(x_i), f_2(y_i)).$$

Indeed, the claimed length is obtained by matching up all the C's, and taking the longest common subsequence of $f_1(x_i)$ and $f_2(y_i)$ for each index i. □

Claim. *If a common subsequence between $g_1(x_1)$ and $g_2(x_2)$ matches up a character from $f_1(x_i)$ with a character from $f_2(y_j)$ for different indices $i \neq j$, then it has length at most $(m'+2)m$.*

Indeed, suppose an edit sequence matched some characters between A_i and B_j for $i \neq j$. Suppose $i < j$. Then there is at least one more substring C to the right of A_i then B_j, which implies that at least m' of the 3's on the right of A_i will not be

matched. Similarly if $i > j$, then at least m' of the 3's on the left of A_i will not be matched. Otherwise each of the A_i's can contribute at most 2. □

Let $m' = 2m$. Then we have $(m+1)m' \geq (m'+2)m$. $(m+1)m'$ is attainable simply by matching up all the C's. Thus the longest common subsequence can always be attained without matching up a character from $f_1(x_i)$ to $f_2(y_j)$ for $i \neq j$. This forces the solution to be bounded above by

$$\mathsf{LCS}(g_1(x), g_2(y)) \leq (m+1)m' + \sum_{i=1}^{m} \mathsf{LCS}(f_1(x_i), f_2(y_i)).$$

Combined with the lower bound in the first claim, we obtain the equality

$$\mathsf{LCS}(g_1(x), g_2(y)) = (m+1)m' + \sum_{i=1}^{m} \mathsf{LCS}(f_1(x_i), f_2(y_i)).$$

Moreover, we have $\mathsf{LCS}(f_1(x_i), f_2(y_i)) = 1 - x_i y_i$ for all i by Lemma 15.8. Substituting into the above completes the proof of the lemma. □

Normalization

Lemma 15.6. *Let $m \in \mathbb{N}$ be fixed. There are functions $h_1, h_2 : \{0,1\}^m \to \{1,2,3,4\}^M$ and an integer $T \in \mathbb{N}$ with the following properties.*

1. *Both $M, T \leq \text{poly}(m)$.*
2. *h_1 and h_2 both take $\text{poly}(m)$ time to evaluate.*
3. *For any two vectors $x, y \in \{0,1\}^m$,*

$$\mathsf{LCS}(h_1(x), h_2(y)) = \begin{cases} T+1 & \text{if } \langle x, y \rangle = 0 \\ T & \text{if } \langle x, y \rangle > 0 \end{cases}$$

Proof. Let $g_1, g_2 : \{1,2\}^m \to \{1,2,3\}^M$ be given by Lemma 15.9, where M is a polynomial in m. Recall that we have

$$\mathsf{LCS}(g_1(x), g_2(y)) = 2m^2 + 3m - \langle x, y \rangle$$

for any two $x, y \in \{0,1\}^m$.

Let $T = 2m^2 + 3m - 1$. Consider the function $h_1, h_2 : \{0,1\}^m \to \{\mathsf{1},\mathsf{2},\mathsf{3},\mathsf{4}\}^{M+T}$ defined by

$$h_1(x) = \mathsf{4}^T g_1(x)$$
$$h_2(y) = g_2(y)\mathsf{4}^T,$$

where $\mathsf{4}^T$ denotes the string of T consecutive 4's. Observe that the LCS of $h_1(x)$ and $h_2(y)$ can only have matches between the $\mathsf{4}$'s, or between $g_1(x)$ and $g_2(y)$, but not both. As such, we have

$$\mathsf{LCS}(h_1(x), h_2(y)) = \max\{T, \mathsf{LCS}(g_1(x), g_2(y))\} = \begin{cases} 2m^2 + 3m = T + 1 & \text{if } \langle x, y\rangle = 0 \\ T & \text{otherwise,} \end{cases}$$

as desired. □

15.4.2 The LCS ∨-gadget

Our goal in this section is to prove the second main lemma, which we first restate for convenience.

Lemma 15.7. *Let $A_1, \ldots, A_n \in \{\mathsf{1},\mathsf{2},\mathsf{3},\mathsf{4}\}^M$ and $B_1, \ldots, B_n \in \{\mathsf{1},\mathsf{2},\mathsf{3},\mathsf{4}\}^M$ be two sets of n bit strings of length M, and T an integer, such that for all A_i and all B_j,*

$$T \le \mathsf{LCS}(A_i, B_j) \le T + 1.$$

Then in $O(Mn)$ time, one can construct strings $\bar{A}, \bar{B} \in \{\mathsf{1},\mathsf{2},\mathsf{3},\mathsf{4},\mathsf{5}\}^{M'}$ of length $M' = O(Mn)$, and select an integer U, such that

$$\mathsf{LCS}\Big(\bar{A}, \bar{B}\Big) > U \text{ iff } \mathsf{LCS}(A_i, B_j) = T + 1 \text{ for some } A_i \text{ and } B_j.$$

Given A_i's and B_j's as described above, the goal, motivated by the reduction from orthogonal vectors, is to decide if there is a single pair of indices i, j such that $\mathsf{LCS}(A_i, B_j) = T+1$. We are particularly interested in the regime where the parameters M and T are lower-order terms and the dominant term is n. In particular, in the reduction from orthogonal vectors, a $O(n^{1.99}\,\mathrm{poly}(M,T))$-time running time identify such a pair A_i and B_j gives a $O(n^{1.999}\,\mathrm{poly}(M,T))$ time algorithm for SAT.

We will combine the A_i's into (some carefully constructed) string $\bar{A}$, and the B_j's into a string $\bar{B}$, each of same size $O(\mathrm{poly}(M,T)n)$. We will design $\bar{A}$ and $\bar{B}$ so that $\mathsf{LCS}\Big(\bar{A}, \bar{B}\Big)$ (somehow) reveals whether or not we have $\mathsf{LCS}(A_i, B_j) = T + 1$ for any pair of indices i and j.

Recall that the A_i's and B_j's consist of characters from the set $\{\mathsf{1}, \mathsf{2}, \mathsf{3}\}$. Let $D = \mathsf{5}^S$ be the string of S consecutive $\mathsf{5}$'s, for a sufficiently large parameter $S \in \mathbb{N}$ TBD. Consider the concatenated strings

$$\bar{A} = D^n A_1 D A_2 D \cdots D A_n D^n$$
$$\bar{B} = B_1 D B_2 D \cdots D B_n D B_1 D B_2 D \cdots D B_n$$

To gain some informal intuition for the above constructions, observe that $\bar{A}$ has $3n-1$ D-blocks and $\bar{B}$ has $2n-1$ D-blocks. For S, the $\mathsf{LCS}\Big(\bar{A}, \bar{B}\Big)$ will be incentivized to match up the maximum number of D-blocks, $2n-1$. Any maximum matching between the $2n-1$ D-blocks of $\bar{B}$ to the $3n-1$ D-blocks to $\bar{A}$, means that each A_i-block above can be partially matched to at most one B_j-block below. The net effect is that the D's encourages the lengths of the LCS to have the form $(2n-1)\,\mathsf{LCS}(D, D) + \sum_{i=1}^{n} \mathsf{LCS}\Big(A_i, B_{\pi(i)}\Big)$ for some mapping $\pi : [n] \to [n]$ of A_i's to B_j. The sum is $> nT$ iff $\mathsf{LCS}\Big(A_i, B_{\pi(i)}\Big) = T + 1$ for one of these matches i.

The formal argument consists of two parts. Let

$$U = (2n-1)S + nT$$

Observe that $\mathsf{LCS}\Big(\bar{A}, \bar{B}\Big) \geq U$ by (say) matching up the A_i-blocks with the first n B-blocks (each contributing at least T), and then matching up all $(2n-1)$ D-blocks in between and afterwards (each contributing S). This lower bound of U is the threshold that will decide whether or not some $\mathsf{LCS}(A_i, B_j) > T$ for some pair of A_i and B_j.

Lemma 15.10. *Suppose* $\mathsf{LCS}(A_i, B_j) = T + 1$ *for some* i, j. *Then* $\mathsf{LCS}\Big(\bar{A}, \bar{B}\Big) \geq U + 1$.

Proof. Suppose $i \leq j$. Consider the common subsequence where we matching up A_i to the first B_j, and then match up A_{i-k} with B_{j-k} to the left and then match up A_{i+k} with B_{j+k} to the right for each relevant value of k. In between we match up the D's. In the example below, we have $n = 5$. The strings are lined up to match up A_3 with B_4, and unmatched blocks are crossed off.

$$\cancel{D D D D D} D A_1 D A_2 D A_3 D A_4 D A_5 D D D D \cancel{D}$$
$$\cancel{B_1} D B_2 D B_3 D B_4 D B_5 D B_1 D \cancel{B_2} D \cancel{B_3} D \cancel{B_4} D \cancel{B_5}$$

The total cost of the matching is at least $(n-1)T + T + 1 + (2n-1)S$, since $\mathsf{LCS}(A_i, B_j) = T + 1$, while the other A-blocks and B-blocks match up to contribute at least T to the LCS.

If $i > j$ then we match up A_i to the *second* B_j instead, and otherwise proceed similarly, arriving at the same conclusion as before. □

Lemma 15.11. *Let $S \geq M - T$. Suppose $\mathsf{LCS}(A_i, B_j) \leq T$ for all i, j. Then $\mathsf{LCS}\left(\bar{A}, \bar{B}\right) \leq U$.*

Proof. We start with the assumption that all the D-blocks of B-blocks are matched, giving us an initial score of $(2n-1)S$. We then analyze each A_i's in order and consider the net contribution from A_i to the LCS. If in the analysis of a block A_i we realize that we did not match part of a D-block in $\bar{B}$ then we charge the unmatched letters from the D-block against the contribution of A_i to the LCS. We claim that each A_i has a net contribution of at most T. For $i = 1, \ldots, n$, consider A_i.

1. If A_i is partially matched to characters of (at most) one B_j, then the maximum contribution from A_i is at most $\mathsf{LCS}(A_i, B_j) \leq T$.

2. If A_i is partially matched to characters of at least two distinct B_j's, then at least one D-block between the B_j's is forfeited. This results in an extra loss of S because we had initially assumed all of the D-blocks in $\bar{B}$ were matched. Thus the net contribution of A_i is at most $|A_i| - S = M - S \leq T$.

In either case, the net contribution each A_i is T. Overall we have

$$\mathsf{LCS}\left(\bar{A}, \bar{B}\right) = (2n-1)S + \sum_{i=1}^{n} (\text{net contrib. of } A_i) \leq (2n-1)S + nT,$$

as desired. □

15.5 Additional notes and materials

The strong exponential time hypothesis was formulated by Calabro, Impagliazzo, and Paturi [CIP09] and builds on the exponential time hypothesis formulated in [IP99]. For more results of this nature, see (e.g.) the recent classes by Bringmann and Künnemann [BK19] and Williams and Williams [WW20]. This chapter is based in part on lecture notes from these classes. The connection between k-SAT and orthogonal vectors is from [Wil05]. The lower bounds for furthest path and diameter in Section 15.2 is by Roditty and Williams [RW13]. The description here is based on a recent talk by Nicole Wein at the Michigan-Purdue Theory Seminar in January, 2021. The presentation of the lower bound for LCS is specifically based on [BK19], and the result is from [ABW15; BK15].

15.6 Exercises

Exercise 15.1. Assuming the strong exponential time hypothesis, give a lower bound on the running time for any exact algorithm for subset sum.

Exercise 15.2. 1. Prove a lower bound for computing the diameter of an unweighted and undirected graph, assuming the strong exponential time hypothesis.

2. Here and in the discussion we gave lower bounds for exact algorithms, but one can consider *approximation* algorithms as well. For $\alpha \geq 1$, let us say that an algorithm gives an α-approximation of the diameter D if it returns a value D' guaranteed to satisfy the inequality $D/\alpha \leq D' \leq D$. For example, a 2-approximation returns a value D' that is at least half the real diameter, and $\leq$ the real diameter.

   For some constant $\alpha > 1$ of your choosing[2], give a lower bound for the running time of any α-approximation for computing the diameter of an unweighted and undirected graph, assuming the strong exponential time hypothesis.

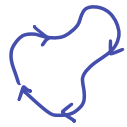
Exercise 15.3. Consider the bichromatic closest pair problem with respect to the L_1-norm, $\|x - y\|_1 = \sum_{i=1}^{m} |x_i - y_i|$. Either (a) give a $O(n^{2-\epsilon} \operatorname{poly}(m))$ time algorithm for some fixed $\epsilon > 0$, or show that such a running time contradicts SETH. Here n is the number of points in each set, and m is the dimension.

[2]The larger the better – I think that 3/2 is doable. Anything greater than 1 is already interesting.

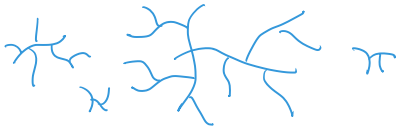
Chapter 16

Spanning trees

Let $G = (V, E)$ be an undirected graph with m edges and n vertices. Recall that a *circuit* in G is any nonempty walk that starts and ends at the same vertex. A *forest* is an edge set $F \subseteq E$ with no circuits (or equivalently, no cycles). A *tree* is a forest with one connected component.



A circuit

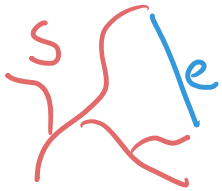


A forest.

For a set of edges $S \subseteq E$, and an edge $e \in E$, we say that S *spans* e if the endpoints of e are connected in S. We say that S is *spanning* if it spans every edge in E. We denote

$$\text{span}(S) = \{e \in E : S \text{ spans } e\}.$$

Observe that $\text{span}(\text{span}(S)) = \text{span}(S)$ for all S.



S spans e.

This chapter focuses on the following two graph problems.

1. Compute a forest $F \subseteq E$ of maximum cardinality (or weight, when the edges are weighted).

2. Compute a spanning edge set $S \subseteq E$ of minimum cardinality (or weight).

It is helpful to catalog the two problems as packing and covering problems, respectively. In the first problem, the goal is to find a set F with as many edges as possible, subject to not taking all the edges of any cycle. In general, problems where we want to take as many elements as possible subject to a family of maximum capacity constraints are called *packing problems*. In the second problem, the goal is to take a set $S \subseteq E$ with as few edges as possible, subject to making sure S spans (i.e., "covers") every

edge in E. In general, problems where we want to take as few as possible subject to a family of minimum requirements are called *covering problems*.

We will develop algorithms that solve both of the above problems. As we do so, it is worth paying attention to the *style* of our investigation. We will make a sequence of small, incremental observations about the above definitions. As we get to understand the definitions well, connections between the two problems emerge, technical issues melt away, and algorithms present themselves. In fact we will end up with several different algorithms for the above problems, and (if done right) it will be easy to see that they all work for the same reasons.

16.1 Unweighted graphs

We first focus on the unweighted versions of the problem. The two problems are to compute a *minimum cardinality spanning set* and a *maximum cardinality forest*. Throughout this section, let κ denote the number of connected components in G.

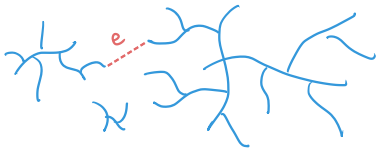
Locally optimal vs globally optimum. An easier problem is to obtain *maximal* forests and *minimal* spanning sets. A forest $F \subseteq E$ is a *maximal forest* if for every edge $e \notin F$, $F + e$ is *not* a forest. A spanning set $S \subseteq E$ is a *minimal spanning set* if for all $e \in S$, $S - e$ is not a spanning set. It is easy to find a maximal forest: start with $F = \emptyset$, and repeatedly add edges to F subject to keeping F a forest, until we are out of options. Similarly one can easily find a minimal spanning set: start with $S = E$, and repeatedly remove edges from S as long as S remains a spanning set. We will end up with a maximal forest F and a minimal spanning set S in polynomial time. These represent *locally optimal* solutions – adding or deleting a single edge either violates a constraint or hurts the cardinality. Unfortunately we do not know if they are *globally optimum* solutions – perhaps a particular sequence of additions / deletions will lead to a better solution than others.

Spanning sets are bigger than forests. Our first lemma bounds the maximum number of edges in any forest, and the minimum number of edges in any spanning set, in a connected graph. The salient point is the relationship it uncovers between forests and spanning sets: every spanning set has at least as many edges as any forest.

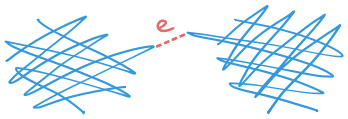
Lemma 16.1. *Let $G = (V, E)$ be connected, let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - 1 \leq |S|.$$

Proof. Consider the forest F. Imagine starting with an empty graph, and tracking the number of connected components as we add edges from F one-by-one. Initially we have n connected components each consisting of an isolated vertex.



Each additional edge e cannot introduce a cycle, so the endpoints of e must be in two different connected components of the current graph. In particular, e reduces the number of connected components by 1. After adding all the edges from F, the number of connected components is at least 1, so we have added at most $n-1$ edges from F. That is, $|F| \leq n-1$.



Now consider the spanning set S. Imagine starting with S as a graph, and tracking the number of connected components of S as we remove edges one-by-one. Initially we have 1 connected component because G is connected and S is spanning. Each time we remove an edge e, the number of connected components increases by at most 1. At the end there are no edges left and n connected components. To reach n connected components, we must have removed at least $n-1$ edges from S. □

The following lemma extends the previous ones to disconnected graphs, by treating each connected component as if it were a connected graph on its own.

Lemma 16.2. *Let $G = (V, E)$ be an undirected graph with κ connected components. Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges. Then*

$$|F| \leq n - \kappa \leq |S|.$$

Proof. Let $G_1 = (V_1, E_1), \ldots, G_\kappa = (V_\kappa, E_\kappa)$ be the connected components of G. For each i, let $F_i = F \cap E_i$ be the restriction of F to G_i, and let $S_i = S \cap E_i$ be the restriction of S to G_i. Let $n_i = |V_i|$. By Lemma 16.1, we have

$$|F_i| \leq n_i - 1 \leq |S_i|$$

for each i. Summing over $i = 1, \ldots, \kappa$ gives the claimed inequality. □

Locally optimal vs globally optimum, revisited. Lemma 16.2 establishes that for any forest F and any spanning set S, we have

$$|F| \leq |S|.$$

Our maximization problem is bounded above by our minimization problem. What is particularly nice is that the two problems can mutually certify each other to be

optimal: *if* F is a forest, and S is a spanning set, such that

$$|F| \geq |S|,$$

then we know that F and S are both globally optimum solutions for their respective problems. Better yet, if a set of edges S was both spanning and acyclic, then it is *both* a maximum cardinality forest and a minimum cardinality spanning set. In the following theorem and proof, observe how the dual relationship between forests and spanning sets plays off each other.

Theorem 16.3. *Let $F \subseteq E$ be a forest, and let $S \subseteq E$ be a spanning set of edges.*

1. *If F is a **maximal** forest, then F is a spanning forest.*
2. *If S is a **minimal** spanning set of edges, then S is a spanning forest.*

In particular, maximal forests are maximum forests, minimal spanning sets are minimum spanning sets, and in both cases they are spanning forests with $n - \kappa$ edges.

Proof. For the first claim, let $F \subseteq E$ be a maximal forest, and suppose by contradiction that F is not spanning. Any edge e not spanned by F can be added to F without introducing a cycle. But then F is not maximal, a contradiction.

For the second claim, let $S \subseteq E$ be a minimal spanning edge set and suppose by contradiction that S is not a forest. In particular, S contains a cycle. Deleting an edge from a cycle in S does not affect connectivity. But then S is not minimal, a contradiction. □

One takeaway is that all minimal spanning edge sets have the same size, and all maximal forests have the same size, which is not obvious *a priori*. It is extremely convenient that minimal equals minimum and maximal equals maximum – it suggests a sort of discrete analogue of convexity and concavity. We highlight again the way that analyzing forests and spanning sets together gives short proofs of powerful facts about forests and spanning sets taken alone. This is our first example of a more general phenomenon called *packing and covering duality*, which we will revisit in the context of paths and cuts in the next two chapters.

Theorem 16.3 proves that the following algorithms both produce a spanning forest:

1. Starting from $F = E$, repeatedly remove edges $e \in F$ as long as $F - e$ remains spanning.
2. Starting from $F = \emptyset$, repeatedly add edges $e \in E \setminus F$ to F as long as $F + e$ remains a forest.

The first algorithm terminates with a minimal spanning edge set, which by Theorem 16.3 is a forest. The second algorithm terminates with a maximal forest, which by Theorem 16.3 is also spanning.

An important special case is when a graph is connected. In this case, the spanning forest is a spanning *tree*.

Corollary 16.4. *Every connected graph has a spanning tree, with $n - 1$ edges.*

16.2 Weighted graphs

Let $G = (V, E)$ be a connected and undirected graph, with edge weights given by $w : E \to \mathbb{R}$. Consider the problem of computing the minimum weight spanning tree (MST) of G. We assume for simplicity that the edge weights are distinct. However the algorithms we develop will work for general weights as well, and we address this at the end of the section. It will also be easy to see that the algorithms generalize to unconnected graphs as well; here we compute the minimum weight spanning forest rather than a tree.

Good edges. Thus, consider the problem of finding the minimum weight spanning tree where we assume the graph is connected and the edge weights are distinct. Our algorithms will all be based on the following definition.

Definition 16.5. *An edge $e \in E$ is* good *if for every set of edges $S \subseteq E - e$ that spans e (and doesn't include e),*

$$w(e) < \max_{f \in S} w(f).$$

The following lemma explains why we call these "good" edges.

Lemma 16.6. *Let $e \in E$. Then the following conditions are equivalent.*

1. *e is good.*

2. *Every minimum weight spanning tree contains e.*

Proof. (2) implies (1). Suppose every minimum weight spanning tree T contains e. Suppose by contradiction that e is not good. Let $F \subseteq E$ be a set of edges spanning e where $w(e) > w(f)$ for all $f \in F$. In particular F contains a path P where $w(e) > w(f)$ for all $f \in P$.

Consider the forest $T - e$ obtained by removing e from T. Let $e = \{a, b\}$. $T - e$ has two connected components A and B, where A contains a and B contains b. As a

path from $a \in A$ to $b \in B$, P must have an edge f crossing from A to B. This edge f is not spanned by $T - e$, so $T' \stackrel{\text{def}}{=} T - e + f$ is a forest. Since T' has $n - 1$ edges, it is also a spanning tree. It has weight

$$\bar{w}(T') = \bar{w}(T) - w(e) + w(f) \stackrel{\text{(a)}}{<} \bar{w}(T)$$

where (a) is because $w(f) < w(e)$ by assumption. But then T is not a minimum weight spanning tree, a contradiction.

(1) implies (2). Suppose e is good. Suppose by contradiction that T is a minimum weight spanning tree excluding e. Consider the unique path $P \subseteq T$ connecting the endpoints of e. Then P spans e, so P contains an edge f with $w(e) < w(f)$.

We claim that $T' = T - f + e$ is a spanning tree. We first observe that $T - f$ does not span e. Indeed, if $T - f$ did span e, then this implies a second path P' in T spanning e distinct from P, a contradiction. Thus $T - f$ is a forest that does not span e, with $n - 2$ edges. $T - f + e$ is then a forest with $n - 1$ edges. Any forest with $n - 1$ edges is a spanning tree.

Thus $T' = T - f + e$ is a spanning tree. It has weight

$$\bar{w}(T') = \bar{w}(T) - w(f) + w(e) < \bar{w}(T),$$

which means that T is not the minimum weight spanning tree, a contradiction. □

The obvious algorithm following Lemma 16.6 would be to repeatedly try to add good edges. Consider the following abstract algorithm.

`generic-MST(`$G = (V, E)$`,`$w : E \to \mathbb{R}$`)`

1. $T \leftarrow \emptyset$
2. Until T is a spanning tree.
   A. Add a good edge $e \in E \setminus \text{span}(T)$ to T
   // Or throw an error if no such edge exists.
3. Return T.

`generic-MST` is very conservative, only adding edges that we know (by Lemma 16.6) must be in every minimum weight spanning tree. There are at least two issues with this approach (particularly in step `(2.A)`) which prevents it from being a "real" algorithm. First, it is not clear that good edges are always available. For the time being we throw an error if we fail to find a good edge. Second, it is not clear how to certify that an edge e is good.

That said, suspending the discussion of these drawbacks for a moment, we can still say the following.

Lemma 16.7. *If* `generic-MST` *terminates, then it returns the minimum spanning tree, which is unique.*

Proof. Indeed, it returns a spanning tree T where, by Lemma 16.6, every edge in T is in every spanning tree. □

It remains to show that `generic-MST` always terminates. In particular, we need to show that an edge satisfying step `(2.A)` always exists.

16.2.1 The greedy algorithm

The following lemma gives our first concrete certificate for good edges.

Lemma 16.8. *Let $H \subset E$ be any non-spanning set of edges. Let $e \in E \setminus \text{span}(H)$ be the minimum weight edge not spanned by H. Then e is a good edge.*

Proof. Let $F \subseteq E - e$ be any set of edges spanning e. Since F spans e and H does not, F is not a subset of $\text{span}(H)$. Any edge $f \in F \setminus \text{span}(H)$ must have $w(f) > w(e)$ by choice of e, as required. □

The key point to the above lemma is that the set H gives a *certificate* that e should be included in the minimum weight spanning tree. Without such a certificate, it is hard to justify taking e. The lemma leads to perhaps our simplest implementation of `generic-MST`: repeatedly add the minimum weight edge that does not introduce a cycle, until we have a spanning tree. This is called the *greedy algorithm* and is also known as Kruskal's algorithm.

`greedy-MST`$(G = (V, E), w : E \to \mathbb{R})$

/ Also known as Kruskal's algorithm. */*

1. Sort and index $E = \{e_1, \ldots, e_m\}$ in increasing order of weight.
2. Set $T \leftarrow \emptyset$.
3. For $i = 1, \ldots, m$:
   - */* Below we discuss how to implement the following predicate efficiently. */*
   - A. If $T + e_i$ is a forest:
     - */* e_i is the minimum weight edge not spanned by T */*
     - 1. Set $T \leftarrow T + e_i$.
4. Return T.

Running time and the disjoint union data structure. Correctness follows from Lemma 16.8 and Lemma 16.7. Running time is another matter and we briefly review the algorithm to set the stage. The algorithm sorts the edges by weight, and repeatedly adds edges in order as long as they do not introduce a cycle. Sorting is easy. However, checking that we do not introduce a cycle in step `(3.A)` is not so obvious. Given an edge $e_i = (u, v)$, we need to quickly figure out if the endpoints u and v are in the same components with respect to the current forest. One could do so by running a search algorithm from u, which leads to a $O(mn)$ running time.

To do better, imagine the connected components of T over the course of the algorithm. Initially, when $T = \emptyset$, every vertex is in its own singleton set. Thereafter, whenever we add an edge e_i to T, we also combine the connected components of the endpoints, replacing the two components with their *union*. Along the way, we are constantly checking two vertices are already in the same component.

These operations are facilitated by the *disjoint union* data structure, which maintains a partition of a set of elements into disjoint groups and supports the following interface.

1. `union(`u`,`v`)`: Given two elements u and v, take the union of their sets.
2. `same-set(`u`,`v`)`: Returns whether or not u and v are in the same set.
3. `new-set(`x`)`: Given a new element x, creates the singleton set $\{x\}$.

We plan to cover the disjoint union data structure later when we focus on data structures. For now we treat it as a black box with the following guarantees.

Lemma 16.9. *There is a data structure that can implement the* `union`*,* `same-set`*, and* `new-set` *operations listed above that, for any m operations over n total elements, takes $O(m + n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function.*

We refer to the wikipedia article for more on the inverse Ackerman function. Suffice it to say that it is asymptotically smaller than $\log(n)$, or $\log(\log(n))$, or $\log(\log(\log(n)))$, or $\log(\log(\log(\log(n))))$...

Now we put everything together.

Theorem 16.10. *The greedy algorithm computes the MST in $O(m \log n)$ time.*

Proof. Each edge added is good by Lemma 16.8, so correctness follows from Lemma 16.7. For the running time, we observe that we make $O(m)$ calls to the `disjoint-union` data structure to maintain disjoint sets over n elements. The total running time from these operations is $O(m + n\alpha(n))$. The remaining operations take $O(m \log n)$ time, where the bottleneck is from sorting. □

`greedy-MST`$(G = (V, E), w : E \to \mathbb{R})$

/ Also known as Kruskal's algorithm. */*

1. Initialize a new `disjoint-union` data structure U.
2. Call U`.new-set(`v`)` for all $v \in V$.
3. Sort and index $E = \{e_1, \dots, e_m\}$ in increasing order of weight. *// $O(m \log n)$*
4. Set $T \leftarrow \emptyset$.
5. For $i = 1, \dots, m$
   A. Let $e_i = \{u, v\}$. Unless U`.same-set(`u`,`v`)` is true:
      1. Set $T \leftarrow T + e_i$ and call U`.union(`u`,`v`)`.
6. Return T

Figure 16.1: The greedy algorithm for MST implemented with the disjoint union data structure.

Note that the $O(m + n\alpha(n))$ bound for the `disjoint-union` data structure was overkill, since sorting was already a bottleneck.

16.2.2 A parallel algorithm

For a set of vertices $S \subset V$, the *cut* induced by S, denoted $\delta(S)$, is the set of edges with exactly one endpoint in S:

$$\delta(S) \stackrel{\text{def}}{=} \{\{u, v\} \in E : u \in S,\ v \notin S\}$$

The name "cut" comes from the fact that removing $\delta(S)$ from G cuts off S from the rest of the graph.

Lemma 16.11. *Let $S \subset V$ be any set of vertices. Then the minimum weight edge in the cut induced by S is a good edge.*

Proof. Let $H = E - \delta(S)$. Then $E \setminus \text{span}(H) = \delta(S)$. The claim now follows from Lemma 16.8. □

This inspires the following instantiation of `generic-MST` with a parallel character: each round, for each connected component S of the current tree T, identify the smallest weight edge in $\delta(S)$. Add all of these edges to T.

parallel-MST($G = (V, E), w : E \to \mathbb{R}$)

/* *Also known as Borůvka's algorithm.* */

1. $T \leftarrow \emptyset$
2. While T has more than 1 connected component:
   A. For each component S of T, identify the smallest weight edge in $\delta(S)$.
   B. Add all of these edges to T.
3. Return T

An alternative perspective is to imagine *contracting* the edges as they are added to T. For an edge $e = \{u, v\}$, *contracting* e means we identify u and v as the same vertex. One way to implement this is to introduce a new vertex (say) z, replace u and v with z in every edge incident to u or v, and remove u and v from the graph.

We can rewrite parallel-MST as a contraction-based algorithm as follows. The contraction version immediately contracts every edge that is added to T. As we do so, every vertex in the contracted graph corresponds to a distinct connected component of T in the original graph.

parallel-MST($G = (V, E), w : E \to \mathbb{R}$)

/* *Also known as Borůvka's algorithm.* */

1. $T \leftarrow \emptyset$
2. While V has more than one vertex:
   A. For each vertex v, identify the smallest weight edge incident to v.
   B. Contract all the selected edges, and add the original, uncontracted versions of them to T.
3. Return T

Theorem 16.12. parallel-MST *computes the minimum spanning tree, and can be implemented to run in* $O(m \log(n))$.

Proof. Correctness follows from Lemma 16.7 and Lemma 16.11. For the running time, we first observe that there are at most $O(\log(n))$ iterations of the outer loop. This is because each iteration reduces the number of components by at least a factor of 2.

Next we claim that each iteration can be implemented in $O(m)$ time. We first analyze the first version that does not contract edges. To build some intuition, we point out that this is very easy in the first round: each vertex scans its list of incident

edges to identify the smallest weight edge incident to that vertex. In subsequent iterations, some additional effort is required to do this on a per-component basis, but it is not too difficult either. By running BFS or DFS, we identify the connected component of each vertex, labeling the vertex with some identifier for the component. (e.g., a unique integer.) We then run through the edges and relabel the endpoints by their component. Then, similar to the first iteration, it is easy to find the smallest edge cut by each component.

The contraction version is implicitly very similar and we analyze it for the sake of completeness. Each iteration, observe that contracting all the selected edges amounts to building a new (smaller) graph. Each round, one should label the connected components formed by the selected edges, and contract each component all at once. Then the whole process takes $O(m)$ time. So again we spend $O(m)$ time per iteration.

In either version, a single iteration takes $O(m)$ time, and there are $O(\log n)$ iterations. This gives the claimed running time. □

Beyond theoretical and sequential running times. Borůvka's algorithm is clearly very simple, and although it is not the fastest theoretical running time for MST we will see, the simplicity seems to lend itself to very good performance in practice[1]. Another important property of Borůvka's algorithm is that it is inherently *parallel*. Each vertex in the contracted graph can pick its lightest weight edge independently. Borůvka's algorithm takes $O(\log n)$ time (up to lower order terms) on most natural distributed and parallel models of computation.

16.2.3 The search algorithm

Our final algorithm, like the previous one, also certifies its decisions by only taking the minimum weight edge of cuts induced by components of the spanning tree. Rather than adding many edges in parallel, this algorithm focuses on one component (which is initially one vertex) and keeps adding the minimum weight edge that grows the component. Finding the next edge can be implemented very efficiently with a Fibonacci heap.

[1]We also point out that the theoretically fastest algorithms [KKT95; Cha00], not covered here, are based on Borůvka's algorithm. See also exercise 16.7.

search-MST($G = (V, E), w : E \to \mathbb{R}$)

/* *Also known as Prim's algorithm.* */

1. Set $T \leftarrow \emptyset$ and $S \leftarrow \{v\}$ for an arbitrary vertex v.

/* *We maintain the invariant that S is the connected component of v in T.* */

2. While T is not a spanning tree:

   /* *We can accelerate the following steps with a Fibonacci heap.* */

   A. Let e be the minimum weight edge in $\delta(S)$.

   B. Set $T \leftarrow T + e$, $S \leftarrow S \cup e$.

3. Return T.

Theorem 16.13. search-MST *returns the MST, and can be implemented to run in $O(m + n \log n)$ time.*

Proof. We maintain a Fibonacci heap [FT87] over $V \setminus S$. For each vertex $v \in V \setminus S$, we use the minimum weight of any edge from v to S as the priority for v. This may be $+\infty$ if (early on) there is no such edge, and can be revised as more vertices are added to S. With this setup, remove-min returns the next vertex to add to S.

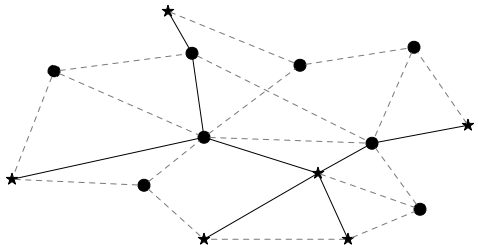
We call decrease-key once for every edge, and remove-min once for every vertex. This gives an overall running time of $O(m + n \log n)$. □

16.2.4 When weights are not distinct

Above we assumed the edge weights are distinct which simplified the analysis substantially. In particular, when edge weights are distinct, the minimum spanning tree is unique.

We can drop this assumption without redoing the analysis, by the following thought experiment. Consider any of the concrete algorithms above. The algorithm – taking minimum weight edges in different ways – still mostly makes sense, although the choice of edge may not be unique. The one caveat is Borůvka's algorithm, but as long as we break ties consistently, it is sensible. That said, consider any fixed run of one of these algorithms, which produces a tree T. As a thought experiment, we can perturb the edge weights so that (a) the edge weights are distinct, and (b) the algorithm would still return the same set of edges. (E.g., enumerate the edges, and increase the ith edge by $i\epsilon$, for ϵ sufficiently small.) Our analysis above applies directly to this thought experiment, which in turn justifies the actual algorithm run on non-distinct weights.

16.3 Steiner trees



We have seen that spanning trees — connecting *all* vertices — can be computed efficiently. What happens if we only need to connect a *subset* of vertices? Let $G = (V, E)$ be an undirected graph. Let $U \subset V$ be a subset of vertices called *terminals*. A *Steiner tree* over U is a tree $F \subseteq E$ in which U is connected. For example, a spanning tree is a Steiner tree for $U = V$. Above is a picture of a Steiner tree where terminals are indicated by $\star$'s, and excluded edges are drawn dashed.

Given positive edge weights $w : E \to \mathbb{R}$, the *minimum weight Steiner tree* problem is to find the Steiner tree whose edges have minimum total weights. Minimum weight Steiner tree is a very important problem in network design – interpreting the edge weights as costs, the minimum weight Steiner tree is the minimum cost network that connects a fixed set of points. Similarly it is very useful in automated VLSI design.

Theorem 16.14. *The Steiner tree problem is NP-Hard, even in unweighted graphs.*

Proof. We describe a reduction from 3-SAT. Let $f(x_1, \ldots, x_n)$ be a 3CNF with m clauses $C_1, \ldots, C_m$. We construct a three-layer graph. The top layer consists of the *variables*, the bottom layer consists of the *clauses*, and the middle layer consists of the *assignments*.

1. For each variable x_j, a terminal vertex (top layer).
2. For each clause C_i, a terminal vertex (bottom layer).
3. For each single-variable assignment such as $x_j =$ `true` or $x_j =$ `false`, a non-terminal vertex (middle layer).

This creates a total of $3n + m$ vertices. Next we create the edges.

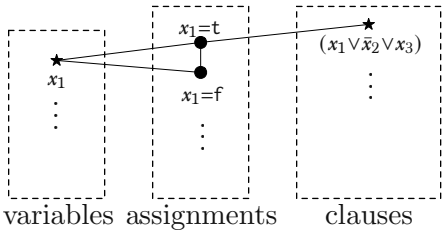
4. An edge between every variable and its two assignments.
5. An edge between every clause and every satisfying assignment.

6. An edge between every pair of assignments.

This creates

$$2n + 3m + \binom{2n}{2} = O\big(m + n^2\big)$$

edges. A high-level diagram of the construction is given below.

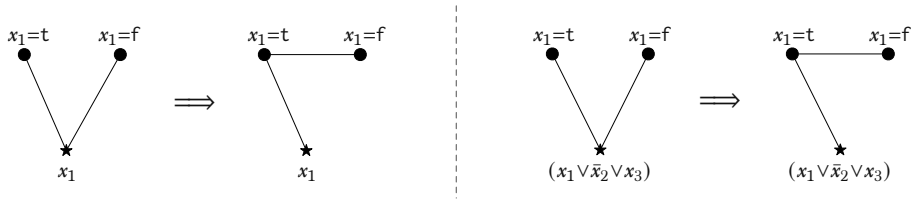


The idea behind the proof is as follows. The assignment vertices act as a switchboard: every connection between terminals must route through them. Each variable must connect to at least one of its two assignments, and each clause must connect to at least one satisfying assignment. The active assignments must themselves be interconnected, at a cost of at least $k - 1$ edges for k assignments. This gives a lower bound of $m + n + (k - 1)$ edges, where $k \geq n$ since each variable requires its own assignment. The minimum of $m + 2n - 1$ is achieved exactly when one assignment is active per variable and every clause reaches one of them — which is precisely a satisfying truth assignment.

We now proceed through the proof more carefully. We claim that f is satisfiable iff there is a Steiner tree of size $m + 2n - 1$.

Suppose we have a satisfying assignment for f. We take the corresponding n edges between variables and assignments. For each clause, we select an edge corresponding to a single-variable assignment that satisfies that clause. Then we add any spanning tree over the assignment vertices, which adds $n - 1$ edges. This gives a Steiner tree of size $m + 2n - 1$.

Now, consider any Steiner tree T in the graph. We claim that if $|T| \leq m + 2n - 1$, then f is satisfiable. We first claim that we can modify T so that each variable is incident to exactly one edge, and each clause is incident to exactly one edge, in T. Indeed, suppose T has both edges incident to a variable x_i. We can delete one of those edges and replace it with the edge between the two assignments $x_i = \texttt{true}$ and $x_i = \texttt{false}$, and T remains a Steiner tree. Likewise, if a clause has two edges in T, we can delete one and replace it with an edge between the two corresponding assignments. See the diagrams below.



Thus, for any Steiner tree T, we may assume that each variable and each clause is incident to exactly one edge. Consider just these $m+n$ edges, and the assignments at the opposite endpoints. Suppose there are k distinct assignments. None of them are connected to each other by these $m+n$ edges, and there is at least one for each variable x_j. Thus we have $k \geq n$ connected components, and the rest of T must have at least $k-1 \geq n-1$ edges to connect them. That is, $|T| \geq m+n+k-1 \geq m+2n-1$. We have $|T| = m+2n-1$ iff $k = n$, iff we are using one assignment for every variable. These assignments give a satisfying assignment for the SAT formula. □

16.4 Additional notes and materials

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

16.5 Exercises

Please see [Eri19, Chapter 7] for further exercises.

Exercise 16.1. Let $G = (V, E)$ be a connected graph, and $F, S \subseteq E$. Decide if the following statements are true or false.

1. F is spanning iff for every cycle C in G with k edges, F contains at least $k - 1$ edges. [2]
2. S is spanning iff S contains a spanning forest.
3. S is spanning iff $|S| \geq n - 1$.
4. If G has distinct edge weights then G has a unique MST.
5. If $F \subseteq \text{span}(S)$, then $|F| \leq |S|$.
6. If F is a forest, and $F \subseteq \text{span}(S)$, then $|F| \leq |S|$.

Exercise 16.2. Consider the following algorithm that claims to compute the MST (in a connected graph).

`decremental-MST`

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in decreasing order of weight.
2. For $i = 1, 2, \ldots, n$:
   A. Unless $E - e_i$ is disconnected:
      1. $E \leftarrow E - e_i$.
3. Return E.

Either

1. Describe an input graph where `decremental-MST` fails to find the MST, or
2. Prove that `decremental-MST` always returns the minimum weight spanning tree.

For simplicity you may focus on the setting where the edge weights are distinct.

[2] To be clear, a cycle specifically means a circuit with no repeating vertices (except at the common vertex where the walk starts and ends).

Exercise 16.3. Let $G = (V, E)$ be an undirected graph with real-valued edge weights. Recall that the MST minimizes the sum of edge weights over all spanning edge sets of G. Consider the problem of computing a spanning edge set $S \subseteq E$ minimizing the maximum edge weight, $\max_{e \in S} w(e)$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 16.4. Let $G = (V, E)$ be a undirected graph with distinct edge weights and let T be the MST. Suppose the weight of a single edge $e \in E$ is altered. The edge e may or may not be in T, and the weight may have been increased or decreased. Describe and analyze an algorithm that fixes the MST.[3]

Exercise 16.5. Consider the problem of computing the second smallest weight spanning tree of a weighted graph G. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 16.6. Let $G = (V, E)$ be an undirected graph with distinct nonnegative edge weights $w : E \to \mathbb{R}$. For a spanning tree T, we say that the *bottleneck weight* of T is the maximum weight edge in T, $\max_{e \in T} w(e)$.

1. Prove that the MST is also a minimum bottleneck weight spanning tree of G.
2. Design and analyze a $O(m + n)$-time algorithm for computing a minimum bottleneck weight spanning tree of G. (This is faster than any of our algorithms for MST.)[4]

Exercise 16.7. Of the multiple algorithms we have seen for MST, recall the parallel approach (Borůvka's algorithm) and the search approach (Prim's algorithm) with Fibonacci heaps. There is also a hybrid approach running in $O((m + n) \log \log n)$ time where one runs the parallel algorithm for some number of iterations (say, k iterations for a parameter k TBD) and contracts the selected edges, and then runs the search algorithm thereafter. (Such an algorithm is correct because ultimately it only takes good edges.) Describe and analyze this algorithm.[5]

[3] Of course one could build an entirely new MST from scratch. Can one do better?
[4] Here's step 1: compute the median edge weight in $O(m)$ time.
[5] It may be helpful to obtain a running time generically as a function of k, and then optimize k.

Exercise 16.8. Let $G = (V, E)$ be a connected graph with distinct edge weights and $k < n$. We define the k-MWF as the minimum weight forest among the forests with exactly k edges. Consider the problem of computing the k-MWF.

1. Call an edge e *k-great* if for every set of edges $F \subseteq E - e$ that (a) spans e or (b) contains a forest of at least k edges,

$$w(e) < \max_{f \in F} w(f).$$

   Prove that e is k-great iff every k-MWF contains e.

2. Design and analyze an algorithm for computing the k-MWF.

Exercise 16.9. For $p > 0$, the L_p-norm of a vector $x \in \mathbb{R}^k$ is defined by

$$\|x\|_p \stackrel{\text{def}}{=} \left(\sum_{i=1}^{k} |x_i|^p \right)^{1/p}.$$

Two edge cases are $p = 0$ and $p = \infty$: here we define $\|x\|_0$ as the number of indices i such that $x_i \neq 0$, and

$$\|x\|_\infty \stackrel{\text{def}}{=} \max_i |x_i|.$$

Let $G = (V, E)$ be an undirected graph with real-valued edge lengths $\ell : E \to \mathbb{R}$. We can take inspiration from L_p-norms to define the L_p-distance in graphs. For a walk w with edges $e_1, \ldots, e_k$, and $p > 0$, we define the L_p-norm of w as the L_p-norm of the vector of edge weights in w,

$$\|w\|_p \stackrel{\text{def}}{=} \left(\sum_{i=1}^{k} |\ell(e_i)|^p \right)^{1/p}.$$

Analogously we define the L_0 and L_∞ norms of w based on the definition for vectors. We define the L_p-distance (for $p > 0$, $p = 0$, and $p = \infty$) between two vertices s and t is the infimum of $\|w\|_p$ over all walks from s to t.

Let $s, t \in V$ be fixed. Below, for different values of p, we consider the problem of computing the L_p-distance from s to t in G. For each of these problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. (2 pts.) The L_0-distance from s to t.

2. (2 pts.) The $L_{1/2}$-distance from s to t.
3. (2 pts.) The L_1-distance from s to t.
4. (2 pts.) The L_2-distance from s to t.
5. (2 pts.) The L_∞-distance from s to t.[6]

Exercise 16.10. Let $G = (V, E)$ be an undirected graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. The *traveling salesman problem* (TSP) asks for the minimum-length tour in G. (A tour starts and ends at the same vertex, and visits each vertex at least once.)

The traveling salesman problem is extremely useful as a prototypical planning and routing problem (broadly construed). It has inspired a number of techniques both in theory and in practice, and also captured the popular imagination. See https://www.math.uwaterloo.ca/tsp/history/index.html for more on the history of this problem.

1. Prove that TSP is NP-Hard (in undirected graphs).[7]
2. Let $\text{OPT}_{\mathsf{TSP}}$ be the minimum TSP value and let $\text{OPT}_{\mathsf{MST}}$ be the weight of the MST with respect to ℓ. Prove that $\text{OPT}_{\mathsf{MST}} \leq (1 - 1/n)\,\text{OPT}_{\mathsf{TSP}}$.[8]
3. Given TSP is NP-Hard, we might try to get a polynomial time *approximation algorithm* instead. For $\alpha > 1$, an *α-approximation* for TSP is a tour of cost at most $\alpha\,\text{OPT}_{\mathsf{TSP}}$. Unlike exact algorithms, a polynomial time α-approximation algorithm does not immediately imply an exact polynomial time algorithm for SAT.

   Design and analyze a polynomial time 2-approximation algorithm for TSP.[9]

Exercise 16.11. Let $G = (V, E)$ be a connected undirected graph with positive edge weights $w : E \to \mathbb{R}_{>0}$, and let $U \subseteq V$ be a set of terminals. Recall that the minimum weight Steiner tree problem asks for a minimum weight tree in G connecting all terminals in U.

[6] For simplicity you may assume the edge weights are distinct.

[7] You can assume that Hamiltonian path in undirected graphs is NP-Hard. (Part 1 of exercise 12.4; it was briefy covered at the end of Kent's "Longest Path" lecture..)

[8] *Hint:* You may first want to show that $\text{OPT}_{\mathsf{MST}} \leq \text{OPT}_{\mathsf{TSP}}$.

[9] *Hint:* By part 2, the MST has weight less than $\text{OPT}_{\mathsf{TSP}}$. So two copies of the MST have weight...

Recall that the (shortest path) distances between vertices define a metric over V. The *metric closure* of G over U is the complete graph on U with edge weights given by the distances in G.

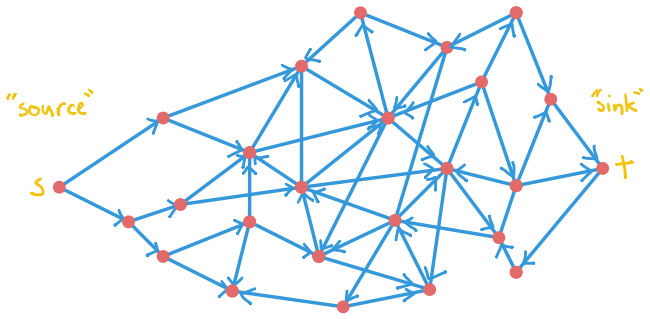
1. Design and analyze an algorithm to compute the metric closure of G over U.
2. Let $\mathrm{OPT}_{\mathsf{ST}}$ denote the minimum weight of a Steiner tree connecting U in G, and let $\mathrm{OPT}_{\mathsf{MST}}$ denote the weight of the MST of the metric closure of G over U. Prove that $\mathrm{OPT}_{\mathsf{MST}} \leq 2\,\mathrm{OPT}_{\mathsf{ST}}$.
3. Building on parts 1 and 2, design and analyze a polynomial time 2-approximation algorithm for the minimum weight Steiner tree problem in undirected graphs.

Exercise 16.12. Let $G = (V, E)$ be an undirected graph with edge weights $w : E \to \mathbb{R}$. Recall that the minimum weight spanning tree can be computed in polynomial time by a number of different algorithms. Suppose we imposed an additional constraint that the spanning tree has maximum (unweighted) degree at most Δ for some given integer $\Delta \in \mathbb{N}$.

1. (5 points) Consider the problem of computing the minimum weight spanning tree with maximum degree at most 2. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.
2. (5 points) Consider the problem of computing the minimum weight spanning tree with maximum degree at most Δ, for $\Delta \geq 3$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Chapter 17

Packing and covering paths



Let $G = (V, E)$ be a directed graph[1], and let $s, t \in V$ be two distinct vertices. We call s the *source* and t the *sink*. A *path packing* is a collection of edge disjoint paths. An (s, t)-path packing is a path packing of (s, t)-paths. The *maximum* (s, t)*-path packing* problem[2] is to

find a maximum cardinality packing of (s, t)*-paths.*

An (s, t)*-cut* is a set of edges whose removal leaves no paths from s to t. The *minimum* (s, t)*-cut problem* is to

find the minimum cardinality (s, t)*-cut.*

Both problems generalize reachability. Reachability asks whether there is a connection from s to t. Path packing and cuts measure the *strength* of the connection.

[1] G is allowed to be a *multi-graph*, with multiple copies of the same edge.

[2] This problem is also called the *uncapacitated maximum* (s, t)*-flow problem* for reasons we discuss later.

17.1 Duality

The (s,t)-path packing and (s,t)-cut problem are *dual* packing and covering problems, in the following sense. We are selecting paths that "pack in" to the edges of the graph – each path uses up all of its edges. Conversely, an (s,t)-cut must contain at least one edge from every (s,t)-path. That is, we are trying to "cover" the paths with edges, where we interpret each edge as a set that covers all the (s,t)-paths that contain that edge. In short:

paths pack into edges, and edges cover paths.

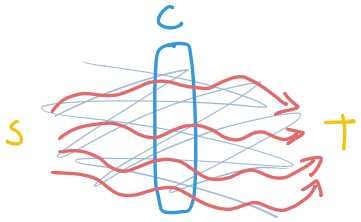
As with any packing and covering problem, we have the following inequality.

$$(\max\ (s,t)\text{-path packing}) \leq (\min\ (s,t)\text{-cut}). \tag{17.1}$$

Indeed, let P denote a path packing, and let $C \subseteq E$ be an (s,t)-cut. We have

$$|P| = \sum_{p \in P} 1 \overset{\text{(a)}}{\leq} \sum_{p \in P} |p \cap C| \overset{\text{(b)}}{\leq} |C|.$$

Above, we treat each path $p \in P$ as a subset of edges. (a) is because, as an (s,t)-cut, C contains at least one edge from every (s,t)-path. (b) is because the paths $p \in P$ are edge disjoint, so the sets $p \cap C$ over $p \in P$ are also disjoint. The following is a conceptual sketch of our argument.



The inequality (17.1) inspires some basic questions. Are the two sides ever equal? Is the inequality ever strict?

17.1.1 Greedy heuristics

Consider now the computational problem of finding the maximum (s,t)-path packing, or finding the minimum (s,t)-cut. For packing, consider the obvious approach of repeatedly taking (s,t)-paths, and adding them to a growing collection.

`greedy-path-packing`$(G = (V, E), s, t)$

1. $P \leftarrow \emptyset$.
2. While there is a path p in G from s to t: *// BFS or DFS*
   A. Add p to P.
   B. Remove all the edges in p from the graph.
3. Return P.

An equally simple and natural approach to the (s, t)-cut problem is as follows. Start with $C = E$; obviously C is an (s, t)-cut. Then repeatedly try to remove edges from C as long as C remains an (s, t)-cut.

`greedy-cut`$(G = (V, E), s, t)$

1. $C \leftarrow E$.
2. While there is an edge $e \in C$ such that $C - e$ is an (s, t)-cut
   A. Remove e from C.
3. return C.

Ideally, we would like to show that these algorithms return the optimum solutions of their respective problems. For starters, it is clear from their terminating conditions that they produce *locally optimal* solutions. `greedy-path-packing` terminates when no more paths can be added to the current solution. `greedy-cut` terminates when no more edges can be removed from the current solution. That is:

Lemma 17.1. `greedy-path-packing` *returns a* maximal (s, t)*-path packing* P*, and* `greedy-cut` *returns a* minimal (s, t)*-cut.*

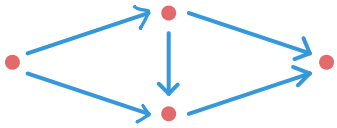
Tautologically, we have

$$(\textit{maximal} \text{ path packing}) \leq (\textit{maximum} \text{ path packing})$$

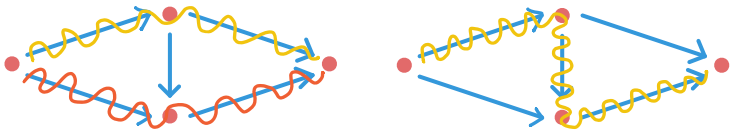
and

$$(\textit{minimal } (s, t)\text{-cut}) \geq (\textit{minimum } (s, t)\text{-cut}).$$

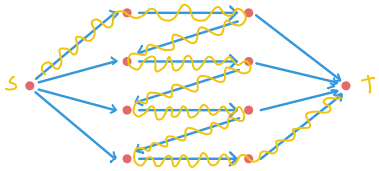
But these relations are only useful if we can strengthen the inequalities to equalities. Unfortunately, simple counterexamples show that they are not equal. Consider the following graph.



Let s be the leftmost vertex and t the rightmost vertex. Clearly the maximum packs two paths from left to right, by going along the top and along the bottom. But a single path using the middle edge also gives a maximal path packing.



Thus a maximal path packing is not a maximum path packing. It is not hard to find simple counterexamples that exaggerate the difference in size between a maximal and maximum path packing.



It is also easy to come up with counterexamples that show that a minimal (s,t)-cut is not a minimum (s,t)-cut.

The point is that

$$(\textit{maximal} \text{ path packing}) \neq (\textit{maximum} \text{ path packing})$$

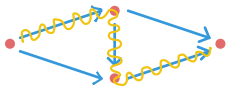
and

$$(\textit{minimal } (s,t)\text{-cut}) \neq (\textit{minimum } (s,t)\text{-cut}).$$

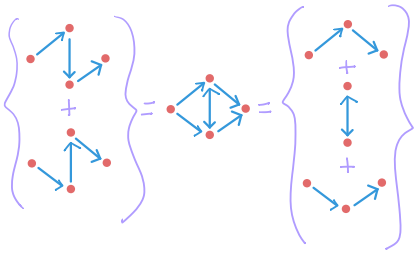
This is different from our previous discussion on spanning trees (Chapter 16). There, one can show that any maximal forest is a maximum forest, and that any minimal spanning set is a minimum spanning set. Moreover, the optimal solutions to both problems have the same value, meeting at a spanning forest. This makes the spanning tree problems conducive to greedy algorithms where we simply keep adding edges to a forest, or keep removing edges from a spanning set, until we can no longer do so while maintaining feasibility. But such simple greedy algorithms apparently do not work for (s,t)-flow and (s,t)-cut.

17.2 Augmenting paths

Recall our very simple counter example for the greedy path packing approach.



If we remove the selected path (in yellow), the two unused edges are separated and so there is no path remaining. But consider the following "path arithmetic":

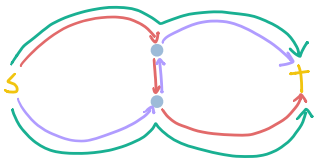


Call two directed edges *opposite edges* if they are the reverse of one another. The two paths on the LHS have a pair of opposite edges. We can "uncross" the two paths at the opposite edges, combining the initial part of one path with the remaining part of the other. The result is another path packing except now there are no opposite pairs of edges. This intuition leads to the following observation.

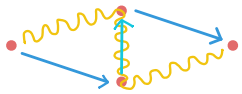
Lemma 17.2. *Let p, q be two edge-disjoint (s,t)-paths. Then there are two edge-disjoint (s,t)-paths p', q' with the following properties:*

1. *p' and q' use a subset of the edges in $p \cup q$.*
2. *p' and q' have no opposite pairs of edges.*

The proof is by induction on the number of opposite pairs of edges, and left to the reader.



Knowing now that we can uncross opposite pairs of edges and maintain a path packing, consider again our simple counter example. The problem before is that we used the middle edge going down, leaving a disconnect in the middle of the graph. But suppose we inserted a new edge in the opposite direction, going from bottom to top.



The new (s,t)-path in this augmented, *residual graph* can be uncrossed with the previously selected path to obtain two paths in the original graph. This clever idea leads to the following algorithm, due to Ford and Fulkerson [FF56]. On first pass, the reader should ignore `(3)` and focus on the path packing portion of the algorithm. Below, $\delta^+(S) \stackrel{\text{def}}{=} \{((u,v) \in E : u \in S,\, v \notin S)\}$ denotes the "out-cut" of edges leaving S.

`augmenting-paths(`$G=(V,E)$`,`s`,`t`)`

`/*` *Ford and Fulkerson [FF56]* `*/`

`1.` $P \leftarrow \emptyset$

`2.` While there is an (s,t)-path $p : s \rightsquigarrow t$:

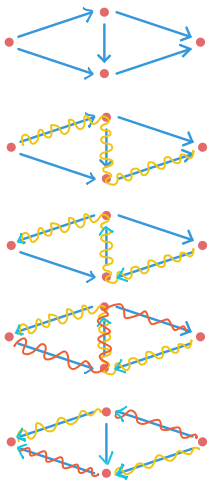
  `A.` Remove all of p's edges from the graph.

  `B.` Add the reverse of all of p's edges to the graph.

  `C.` Uncross p with previously selected paths, and add p to P.[3]

`3.` $S \leftarrow$ {vertices still reachable from s}.

`4.` Return the (s,t)-path packing P and the (s,t)-cut $\delta^+(S)$.



A simulation of `augmenting-paths` on our simple counterexample is given on the right. Here we start with a simple graph, and in the first iteration take the "bad path" in yellow. We then add the reverse edges, indicated by the new arrows in teal. With these new edges we now have another path, using the middle edge in reverse, from left to right. We update the residual graph as reflected by the updated teal arrows. In the last residual graph, there is no (s,t)-path remaining.

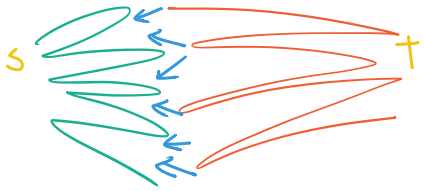
We remark that one can skip steps `(1)`, `(2.C)`, and `(3)` (and instead count the number of iterations of `(2)`) if the goal is only to compute the size of the maximum path packing.

We now prove optimality. Define the *residual graph* of a path packing as the graph obtained by removing the paths and adding their reverses. The algorithm maintains this residual graph throughout. The key observation is the following.

[3] Strictly speaking, after uncrossing, we may end up with a collection of walks rather than paths. But any feasible "walk-packing" can be converted into a path-packing by shortcutting any cycles that may arise.

Observation 17.3. *Let $S \subset V$ be any set of vertices with $s \in S$ and $t \notin S$. Let $p : s \rightsquigarrow t$ be an (s,t)-path. Then p has exactly one more edge from $\delta^+(S)$ than from $\delta^-(S)$. In particular, the residual graph w/r/t p decreases $|\delta^+(S)|$ by 1 and increases $|\delta^-(S)|$ by 1.*

The `augmenting-paths` algorithm terminates only when the residual graph, here denoted G', has no (s,t)-path. Let S denote the set of vertices reachable from s in the final residual graph, and let $T = V \setminus S$ denote the remaining vertices. Note that $s \in S$ and $t \in T$. Below, S is in green and T is in red.



Let $k = |\delta^+(S)|$ denote the number of edges in the out-cut of S in the original graph. Let us observe the changes to $\delta^+(S \,|\, G)$ over the course of the `augmenting-paths` algorithm. Each iteration, we increase our path packing by one, and the size of $\delta^+(S)$ decreases by one in the residual graph. Thus the algorithm terminates in exactly k iterations, with $|P| = k$. But we already know, from duality, that any path packing has at most k paths. Thus the path packing P produced by the `augmenting-path` algorithm is certifiably optimal. Moreover, since the size of any (s,t)-path packing is a lower bound for the size of any (s,t)-cut, we simultaneously obtain the fact that $\delta^+(S)$ is a minimum (s,t)-cut.

The algorithm produces a path packing and a cut of the same size and each certifies the other. It reveals graph structure.

Theorem 17.4 (Menger's theorem)**.** *In any directed graph, for any two vertices s, t,*

$$(\textit{maximum } (s,t)\textit{-path packing}) = (\textit{minimum } (s,t)\textit{-cut}).$$

Algorithmically:

Theorem 17.5 ([FF56])**.** *Let $G = (V, E)$ be a directed graph with m edges, and let $s, t \in V$ be distinct vertices. Let λ denote the size of the maximum (s,t)-path packing. Then* `augmenting-paths(`G`,`s`,`t`)` *produces both a maximum path packing and a minimum cut in $O(m\lambda)$ time.*

Proof. We have already shown the correctness of the algorithm. For the running time, each successive path can be obtained by BFS or DFS. By maintaining the paths as doubly linked lists, it is easy to "uncross" the selected paths in $O(n)$ time per path. □

For uncapacitated graphs, λ is at most m, so `augmenting-paths` is polynomial time. However, the next chapter will consider the generalization where edges have *capacities* $c : E \to \mathbb{R}_{>0}$, that allow us to reuse an edge e up to $c(e)$ times. Here λ may be exponentially large, and this running time is no longer satisfactory.

17.3 Multicommodity flow

We now consider a generalization to path packing with multiple source sink pairs. This is sometimes called *integral multicommodity flow*; here we will call it multicommodity path packing to emphasize the combinatorial nature and distinguish from the continuous version of the problem that we have not discussed.

Let $(s_1, t_1), \ldots, (s_k, t_k)$ be k source-sink pairs. For each i, let $d_i \in \mathbb{N}$ be a given *demand*. The goal is to find a path packing that, for each i, contains d_i (s_i, t_i)-paths. The (s, t)-path packing discussed earlier is a special case with $k = 1$.

Theorem 17.6. *The integral multicommodity flow problem is NP-Hard, even in uncapacitated graphs.*

Proof. We present a reduction from 3-SAT by Even, Itai, and Shamir [EIS76].

1. For every variable x_j, we make a gadget that looks like two parallel paths with the same starting point and two end point. One of these paths will encode the assignment $x_j =$ `true` and the other will encode the assignment $x_j =$ `false`.
   (a) We introduce two vertices A_j^- and A_j^+.
   (b) Let $i_1, \ldots, i_k$ be the indices of the clauses C_i satisfied by $x_j =$ `true`. For each index i_ℓ, we create two vertices $B_{i_\ell,j}^-$ and $B_{i_\ell,j}^+$. We add edges to form a path from A_j^- to A_j^+ through the pairs. More explicitly, we add edges from A_j^- to $B_{i_1}^-$, from $B_{i_\ell,j}^-$ to $B_{i_\ell,j}^+$ for each i_ℓ, and from $B_{i_\ell,j}^+$ to $B_{i_{\ell+1},j}^-$ for each $\ell < k$.
   (c) We similarly construct an (A_j^-, A_j^+)-path for the clauses satisfied by $x_j =$ `false`.
2. We connect the variable-gadgets by adding edges from A_j^+ to A_{j+1}^- for $j = 1, \ldots, n-1$.
3. We create a new source s_1 and a new sink t_1. We then add edges from s_1 to A_1^- and from A_n^+ to t_1. Observe that any (s_1, t_1) path has to choose, for each variable x_j, one of the two paths in the gadget for x_j.

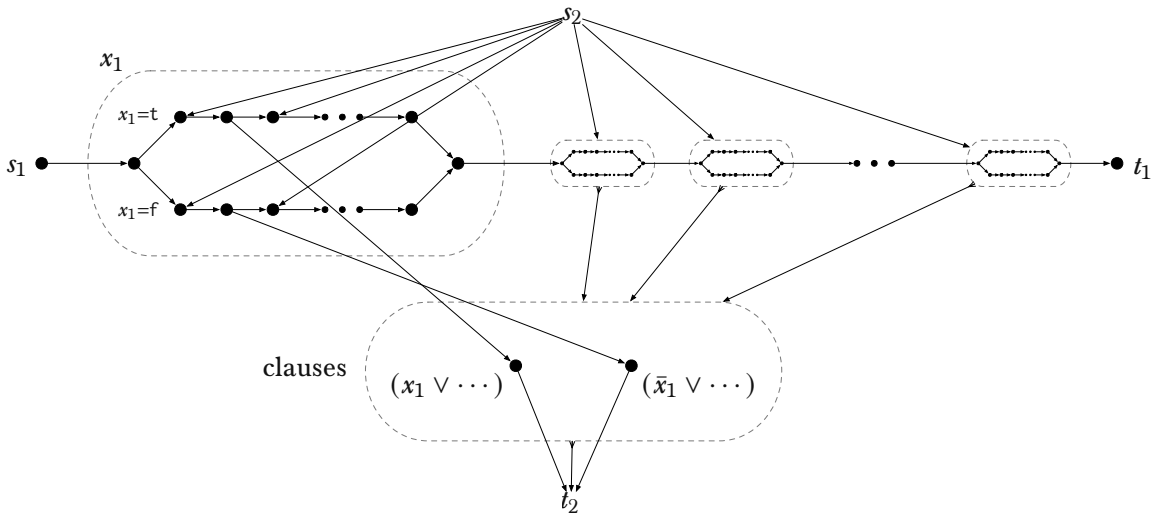


Figure 17.1: Reduction from SAT to multicommodity path packing, with two commodities and a simple directed graph.

4. For each clause C_i, we make variable that we also call C_i. For every variable x_j appearing in C_i, we add an edge from $B^+_{i,j}$ to (the vertex) C_i.

5. We create a new source s_2 and a new sink t_2. We add edges from s_2 to $B^-_{i,j}$ for all i, j and from C_i to t_2 for all i.

6. We make (s_1, t_1) one commodity with requirement one, and (s_2, t_2) another commodity with requirement m.

While the construction is lengthy to write out in full, with some help from Fig. 17.1, it is easy to see what is going on. To pack m (s_2, t_2)-paths, the solution has to pack one path per clause. Each of these paths goes through the "middle edge" of an assignment that satisfies it. Meanwhile, any (s_1, t_1)-path has to traverse, for each variable x_j, either the "$x_j =$ `true`" path or the "$x_j =$ `false`" path. Any feasible solution, based on which paths the (s_1, t_1)-path takes, gives a feasible solution for the SAT formula. Meanwhile a feasible solution maps to an (s_1, t_1)-path, after which it is either pack m paths (s_2, t_2) through all the vertices. We leave the verification of details to the reader. □

17.4 Additional notes and materials

Spring 2026 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.

- Handwritten notes annotated during the presentation.
- For some mysterious reason the iPad recording was lost (again!); please see the 2024 lecture instead.

Spring 2025 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Unfortunately we lost the recording when the iPad ran out of batteries; please see the 2024 lecture instead.

Spring 2024 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

17.5 Exercises

Exercise 17.1. Let $G = (V, E)$ be a directed graph and $s, t \in V$, where we assume there is no edge from s to t. A collection of (s, t)-paths $p_1, \ldots, p_k$ are said to be *vertex-disjoint* if the interior vertices (excluding s and t) of the paths are disjoint. A set of vertices $C \subseteq V \setminus \{s, t\}$ is a *vertex* (s, t)*-cut* if removing C (and all incident edges) from G disconnects s from t. Consider the problems of (a) computing the maximum cardinality collection of vertex disjoint (s, t)-paths, and (b) the minimum vertex (s, t)-cut. For both of these problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 17.2. Let $G = (V, E)$ be a directed graph. For $s, t \in V$, let $\lambda(s, t)$ denote the size of the minimum (s, t)-cut. Prove the following "triangle inequality" for cuts:

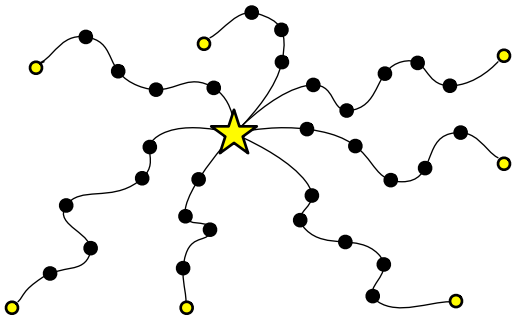
$$\lambda(a, b) \geq \min\{\lambda(a, c), \lambda(c, b)\}$$

for any three vertices $a, b, c \in V$.

Exercise 17.3. Let $G = (V, E)$ be an undirected graph, and let $a, b, c \in V$ be three distinct vertices. We define an (a, b, c)*-path* as a path from a to c that goes through b.

Consider the problem of deciding if there exists an (a, b, c)-path. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 17.4. A *star* is defined as a tree where at most one vertex has degree > 2. This special vertex is called the *center* of the star. (A star can have no particular center when it is just a path.)



Let $G = (V, E)$ be an undirected graph and $L \subseteq V$ a fixed set of vertices. Here we are interested in stars (as subgraphs of G, and centered anywhere) with as many leaves in L as possible. Consider the problem of computing the maximum number of leaves in L that can be obtained by any star in G. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 17.5. According to Wikipedia, a pinwheel is a simple child's toy made of a wheel of paper or plastic curls attached at its axle to a stick by a pin. It is designed to spin when blown upon by a person or by the wind.

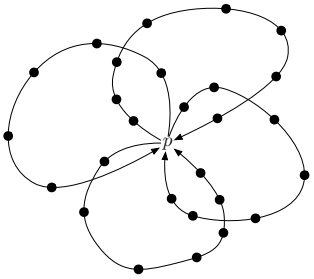
A similar toy had developed independently in Polynesia (known as pekapeka or peʻapeʻa) using either coconut palm leaflets or strips of pandanus leaves;[4][5] in colder climates like that of New Zealand (the toy also called pepepe in Māori), phormium leaves are used.[6]

[4]Koch, Gerd (1984). The Material Culture of Tuvalu. Suva: Institute of Pacific Studies, University of the South Pacific. p. 162. ISBN 9820202051.

[5]Te Rangi Hīroa (1930). Samoan Material Culture. Bernice P. Bishop Museum. p. 552.

[6]Beattie, Herries (1994). Traditional Lifeways of the Southern Maori: The Otago University Museum Ethnological Project, 1920. University of Otago Press. p. 68. ISBN 978-0-908569-79-3.

Let $G = (V, E)$ be a directed graph. A *pinwheel* (in G) is defined as a collection of directed cycles all containing a common vertex p and vertex disjoint otherwise. p is called the *pin* of the pinwheel, and each cycle is called a *curl*. Consider the problem of computing the maximum number of curls in any pinwheel. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

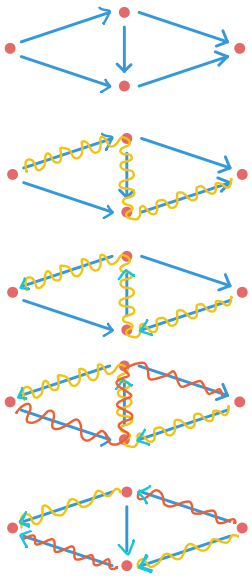


Exercise 17.6. As you know well, this class is a lot: lots of students, lots of homework, and a huge staff. To divvy up the grading load, we want to divide the staff into two teams, Team Divide and Team Conquer, which will alternate homework grading week by week. Some staff are already committed to a team. The rest are flexible. The staff also has many pairwise friends between them. We prefer to keep friends together; friendly grading teams make for more friendly grading. (The teams do not have to be the same size.)

Formally, let S denote the set of n staff members. We have as input a list of m pairwise friendships among the staff. We also have two disjoint subsets $D, C \subseteq S$ of staff members preassigned to Team Divide and Team Conquer, respectively. The goal is to assign the remaining staff members in $S \setminus (C \cup D)$ so as to minimize the number of friendships split across the teams. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Chapter 18

Network flow and linear programs



The previous chapter introduced two dual problems, packing edge-disjoint (s,t)-paths and finding a minimum (s,t)-cut. The maximum packing is at most the minimum cut, and this inequality holds for all packing/covering setups. The augmenting paths algorithm [FF56] computes both simultaneously. Each solution certifies the other's optimality, proving

$$\begin{pmatrix} \text{max \# edge-} \\ \text{disjoint } (s,t)\text{-paths} \end{pmatrix} = \begin{pmatrix} \text{min \# edges} \\ \text{in any } (s,t)\text{-cut} \end{pmatrix}.$$

Unlike spanning trees, greedily selecting (s,t)-paths does not work. The augmenting paths algorithm selects paths in a "residual graph" that lets the algorithm undo mistakes. The reverse edges in the residual graph encode the observation that two (s,t)-paths traversing the same edge in opposite directions can always be "uncrossed." The algorithm runs in $O(m\lambda)$ time where λ is the optimum value.

18.1 Edge capacities, fractional path packings, and capacitated cuts

Suppose we extend the graph with positive *edge capacities* $c : E \to \mathbb{R}_{>0}$. Each edge e can now be used by up to $c(e)$ paths.

To model this mathematically, let $\mathcal{P}_{s,t}$ denote the family of all (s,t)-paths. Note that $\mathcal{P}_{s,t}$ may be exponentially large. A (fractional) path packing is defined as a nonnegative combination $x : \mathcal{P}_{s,t} \to \mathbb{R}_{\geq 0}$ such that for all edges e,

$$\sum_{p \in \mathcal{P}_{s,t} : e \in p} x_p \leq c(e). \tag{18.1}$$

A fractional path packing x is said to be *integral* if every weight x_p is an integer. The *maximum (capacitated) path packing* problem is to compute the path packing x (satisfying (18.1) for all e) of maximum total size:

$$\begin{aligned} \text{maximize} \quad & \sum_{p \in \mathcal{P}_{s,t}} x_p \text{ over } x : \mathcal{P}_{s,t} \to \mathbb{R}_{\geq 0} \\ \text{s.t.} \quad & \sum_{p \in \mathcal{P}_{s,t} : e \in p} x_p \leq c(e) \text{ for all } e. \end{aligned} \tag{Max-Paths}$$

Let (Max-Paths) denote the optimum value of the packing problem (Max-Paths).

On the dual side, edge capacities become costs. The *minimum (capacitated) (s,t)-cut* minimizes the total capacity of edges in the cut.

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in C} c(e) \text{ over all } C \subseteq E \\ \text{s.t.} \quad & |p \cap C| \geq 1 \text{ for all } p \in \mathcal{P}_{s,t}. \end{aligned} \tag{Min-Cut}$$

Even with capacities, the maximum packing is at most the minimum cut. (Why?) But are they still equal?

Integer capacities. Suppose each capacity is an integer. Replace each edge e with $c(e)$ parallel copies to obtain an uncapacitated problem. The augmenting paths algorithm on this multigraph gives (Max-Paths) = (Min-Cut), and the maximum path packing is integral. The only catch is that the running time $O(m\lambda)$ is not polynomial.

Rational capacities. Suppose each edge capacity is a rational number. Multiplying by a common denominator reduces to the integer case, giving (Max-Paths) = (Min-Cut) for rational capacities.

Real-valued capacities. For real-valued capacities, we cannot clear denominators. Instead, approximate c by rational capacities $\tilde{c}$ within $\epsilon > 0$. The rational result gives

$$|\text{(Max-Paths)} - \text{(Min-Cut)}| \leq O(\epsilon m)$$

for the original capacities c. Taking $\epsilon \to 0$ gives (Max-Paths) = (Min-Cut) for real-valued capacities.

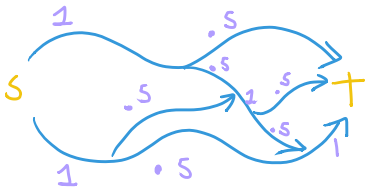
Theorem 18.1. *Let $G = (V, E)$ be a directed graph with positive capacities $c : E \to \mathbb{R}_{\geq 0}$. Then*

$$\text{(Max-Paths)} = \text{(Min-Cut)}.$$

Moreover, if the edge capacities are integral, then there is an integral maximum (s,t)-path packing.

This extends Menger's theorem to general capacities. We still do not have a strongly polynomial time algorithm.

18.2 Network flow



A path packing assigns a weight to each path. But the edge constraints depend only on the "amount" of paths on each edge. Consider the following "flow" model that forgets there were ever any paths to begin with. Imagine each edge as a pipe. A liquid "flow" enters the network at s and exits at t. The flow through any pipe is limited by its capacity. At every intermediate vertex, what flows in must flow out.

Mathematically we model "flow" as follows. An (s,t)-*flow* is a nonnegative vector, $f : E \to \mathbb{R}_{\geq 0}$, satisfying the following (linear) constraints.

1. *Edge capacity constraints.* For each edge e, $f_e \leq c(e)$.
2. *Conservation of flow.* For every non-terminal vertex $v \in V \setminus \{s,t\}$, the total amount of flow going into v equals the total amount of flow leaving v:

$$\sum_{e \in \delta^-(v)} f_e = \sum_{e \in \delta^+(v)} f_e.$$

Canceling opposite flow. Suppose an opposite pair of edges, $e_1 = (u,v)$ and $e_2 = (v,u)$, both have positive flow: $f(e_1) > 0$ and $f(e_2) > 0$. In this case we can subtract equal amounts from both edges without violating any constraints, insofar as both values remain nonnegative. More precisely, if we let $\delta = \min\{f(e_1), f(e_2)\}$, then subtracting δ from both $f(e_1)$ and $f(e_2)$ ensures that at most one of these directions carries nonzero flow. In this sense we can always "cancel out" opposite flow, just like "uncrossing paths" in the previous chapter. [1]

Net flow at the terminals. The *net flow into* a vertex v is defined as

$$\sum_{e \in \delta^-(v)} f_e - \sum_{e \in \delta^+(v)} f_e.$$

The *net flow out* of v is the negation of the net flow into v.

[1] Some of the literature defines flows so that at most one of $f(e_1)$ or $f(e_2)$ is positive at any point in time. We do not emphasize this point here, knowing that opposite flow can easily be canceled out.

Conservation of flow states that every non-terminal vertex has net flow 0. "What goes in must come out." It implies that the net flow out of the source s equals the net flow into the sink t. To verify this mathematically, we have

$$\begin{aligned} 0 &= \sum_{e \in E} f_e - f_e \overset{\text{(a)}}{=} \sum_{v \in V} \sum_{e \in \delta^-(v)} f_e - \sum_{e \in \delta^+(v)} f_e \\ &\overset{\text{(b)}}{=} \sum_{e \in \delta^-(s)} f_e - \sum_{e \in \delta^+(s)} f_e + \sum_{e \in \delta^-(t)} f_e - \sum_{e \in \delta^+(t)} f_e \\ &= (\text{net flow into } s) + (\text{net flow into } t) \end{aligned}$$

Here (a) distributes each term $f_e - f_e$ to the initial and end points of the edge e, giving a sum of net flows for every vertex. (b) applies conservation of flow to the non-terminal net-flows. Rearranging gives

$$(\text{net flow out of } s) = -(\text{net flow into } s) = (\text{net flow into } t),$$

as desired.

The net flow out of s (equivalently, the net flow into t) is called the *size* of the flow. We let $|f|$ denote the size of a flow f.

Max flow. The *maximum (s,t)-flow problem* is to compute a maximum size (s,t)-flow.

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e) \text{ over } f : E \to \mathbb{R}_{\geq 0} \\ \text{s.t.} \quad & \sum_{e \in \delta^+(v)} f(e) = \sum_{e \in \delta^-(v)} f(e) \text{ for all } v \in V \setminus \{s,t\}, \\ & f(e) \leq c(e) \text{ for all } e \in E. \end{aligned} \tag{MaxFlow}$$

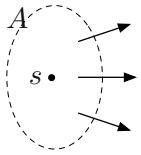
The objective is the size of f, expressed as the net flow out of s. The equality constraints enforce conservation of flow at each non-terminal vertex. The inequality constraints enforce edge capacities. Note that the objective and constraints are all linear, making this an example of a *linear program*. ((Max-Paths) is also a linear program, but with exponentially many variables.)

Max flow vs max path packing. Given an (s,t)-path packing $x : \mathcal{P}_{s,t} \to \mathbb{R}_{\geq 0}$, consider the (s,t)-flow f where the flow on an edge equals the total quantity of paths using that edge:

$$f_e = \sum_{p \in \mathcal{P}_{s,t} : e \in p} x_p.$$

Observe that f is a feasible (s,t)-flow, and that the size of f equals the total quantity of paths in x. Thus we always have (Max-Paths) $\leq$ (MaxFlow).

Max flow vs Min cut. Now consider any (s,t)-cut of the form $C = \delta^+(A)$, where $A \subseteq V$ with $s \in A$ and $t \notin A$. We have



$$\begin{aligned} |f| &= (\text{net flow out of } s) \overset{\text{(c)}}{=} \sum_{v \in A} (\text{net flow out of } v) \\ &= \sum_{v \in A} \sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e) \\ &\overset{\text{(d)}}{=} \sum_{e \in \delta^+(A)} f(e) - \sum_{e \in \delta^-(A)} f(e). \end{aligned}$$

Here (c) observes that all $v \in A - s$ have net flow 0. (d) observes that every edge with both endpoints in A appears as both $f(e)$ and $-f(e)$, canceling each other out. This leaves a positive sum of flows along edges leaving A, and a negated sum of flows along edges entering A.

The RHS above describes the net flow out of A. The net flow out of A is always bounded above by the capacity of $\delta^+(A)$, as

$$\sum_{e \in \delta^+(A)} f(e) - \sum_{e \in \delta^-(A)} f(e) \overset{\text{(e)}}{\leq} \sum_{e \in \delta^+(A)} f(e) \overset{\text{(f)}}{\leq} \sum_{e \in \delta^+(A)} c(e).$$

Here (e) is by nonnegativity of flow, and (f) is because the flow obeys edge capacities. Altogether, we have

$$|f| \leq \sum_{e \in \delta^+(A)} c(e)$$

for every set of vertices A that includes s and omits t. Taking the maximum over all such f, and the minimum over all such A gives

$$(\text{MaxFlow}) \leq (\text{Min-Cut}).$$

We have now established

$$(\text{Max-Paths}) \leq (\text{MaxFlow}) \leq (\text{Min-Cut}).$$

But we already know that (Max-Paths) = (Min-Cut), compressing the inequality into equality.

Theorem 18.2. *Let $G = (V, E)$ be a directed graph with positive edge capacities and $s, t \in V$. Let* (Max-Paths)*,* (Min-Cut)*, and* (MaxFlow) *denote the optimum values of the maximum (s,t)-path packing, the minimum (s,t)-cut, and the maximum (s,t)-flow, respectively. Then*

$$(\text{Max-Paths}) = (\text{MaxFlow}) = (\text{Min-Cut}).$$

Moreover, if the capacities are integer, then there is an integral maximum (s,t)-path packing, and an integral maximum (s,t)-flow.

18.3 Algorithms for maximum flow

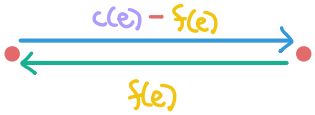
We now turn to computing the maximum flow in a capacitated graph. Previously we showed how to pack paths in an unweighted graph, by indiscriminately selecting paths in the residual graph. This gives us a polynomial time algorithm for maximum flow with unit capacities. However it does not scale with integral capacities as the number of iterations is the size of the flow.

18.3.1 Residual graphs

We first generalize residual graphs from path packings to flows.

Let $e \in E$ be a fixed edge, and suppose we have a flow f that sends some flow along e. Note that $0 \leq f(e) \leq c(e)$ by definition. We can still route $c(e) - f(e)$ more units of flow in the same direction while still satisfying the capacity constraints. We can also effectively route $f(e)$ *units of flow in the opposite direction*. Sending one unit of flow in the opposite direction is equivalent to simply decreasing $f(e)$ by one. In general, canceling flow in opposite directions is neutral w/r/t flow conservation, and can only help with the capacity constraints. Thus in the residual graph we introduce an edge with capacity $f(e)$ in the opposite direction.





The augmenting paths algorithm extends naturally. Whenever we find an augmenting path, we route as much flow as possible.

augmenting-paths($G = (V, E)$, $c : E \to \mathbb{R}_{>0}$, s, t)

1. $f \leftarrow \mathbb{0}$.
2. While there is an (s, t)-path in the residual graph:
   A. Let p be any (s, t)-path.
   // Route as much flow along p as possible
   B. Let γ be the minimum (residual) capacity of any edge in p
   C. Add γ units of flow along p to f and update the residual graph.
3. $S \leftarrow$ the set of vertices reachable from s in the residual graph.
4. Return the (s, t)-flow f and the directed (s, t)-cut induced by S.

This algorithm will terminate, but it can take a very long time. We challenge the reader to think of a simple example where this algorithm would be inefficient (exercise 18.1).

18.3.2 Shortest augmenting path

We make the following adjustment due to Edmonds and Karp [EK72]. Rather than selecting any augmenting path, we repeatedly select the shortest one.

`shortest-augmenting-paths`$(G = (V, E),\ c : E \to \mathbb{R}_{>0},\ s,\ t)$

// Edmonds and Karp [EK72]

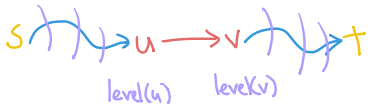
1. $f \leftarrow \mathbb{0}$
2. while there is an (s, t)-path in the residual graph
   A. let p be the *shortest* (s, t)-path
   // Route as much flow along p as possible
   B. let γ be the minimum (residual) capacity of any edge in p
   C. add γ units of flow along p to f and update the residual graph.
3. $S \leftarrow$ the set of vertices still reachable from s in the residual graph.
4. return the (s, t)-flow f and the directed (s, t)-cut induced by S

We want to argue that the `shortest-augmenting-paths` algorithm terminates in strongly polynomial time. The proof is perhaps surprising, as it is based on the *distance* from s to each vertex v in the residual graph. For a vertex v, let $\ell(v)$ denote the unweighted distance from s to v in the residual graph. Note that $\ell(v)$ is either $+\infty$ or at most $n - 1$ for all vertices v. The key lemma is as follows.

Lemma 18.3. *For all $v \in V$, the (unweighted) distance $\ell(v)$ from s to v in the residual graph never decreases.*

Let us first assume Lemma 18.3 and see why it useful in deriving a polynomial time algorithm. Consider a directed edge $e = (u, v) \in E$.

If e is removed from the residual graph, then e was in the shortest (s, t)-path in that iteration. In particular, we must have $\ell(v) = \ell(u) + 1$ at the beginning of the iteration.





Whenever $e = (u, v)$ is added back into the residual graph, the opposite edge (v, u) was in that iteration's shortest (s, t)-path. Then we must have $\ell(u) = \ell(v) + 1$ at the beginning of that iteration.

Now, if $\ell(v) = \ell(u) + 1$ at one point in time, and then later $\ell(u) = \ell(v) + 1$, then $\ell(u)$ and $\ell(v)$ must have switched order. Meanwhile Lemma 18.3 asserts that the distance levels never decrease. This implies that $\ell(u)$ must have increased by two. Since the maximum finite distance is $n - 1$, we see that e can get deleted or added to the residual graph at most n times.

The `shortest-augmenting-paths` algorithm routes the maximum amount of flow along each successive shortest path. For each path, the maximum amount equals the minimum (residual) capacity of any edge along the path. This is sometimes called the "bottleneck capacity" and the corresponding edge is sometimes called the "bottleneck edge". Routing the bottleneck capacity ensures that the bottleneck edge is removed from the residual graph. We charge that iteration to the bottleneck edge. Since an edge can be deleted at most $O(n)$ times from the residual graph, `shortest-augmenting-paths` has at most $O(mn)$ iterations. Each iteration takes $O(m + n)$ time to find the shortest augmenting path and update the residual graph.

In conclusion, if we assume Lemma 18.3 is true, then we have the following.

Theorem 18.4 ([EK72])**.** *The* `shortest-augmenting-paths` *algorithm computes the maximum (s, t)-flow and minimum (s, t)-cut in $O(m^2 n)$ time.*

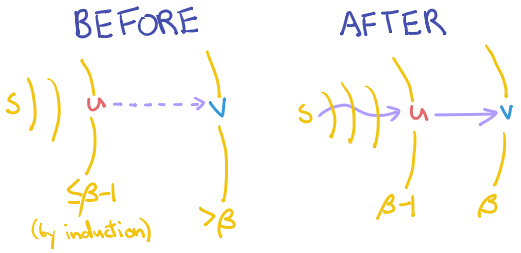
It remains to prove Lemma 18.3. We restate the claim for the reader's convenience.

Lemma 18.3. *For all $v \in V$, the (unweighted) distance $\ell(v)$ from s to v in the residual graph never decreases.*

Proof. Let us refer to the distance from s to a vertex v as the "level" of v, denoted $\ell(v)$ (in the spirit of BFS). Suppose we augment along the shortest (s, t)-path p. We need to show the level of every vertex does not decrease. We prove this by induction on the level of the vertex *after* updating the residual graph. In the base case, the only vertex that can have level 0 is s, and the level of s never changes.

In the general case, let v be a vertex with level > 0 after the augmentation. Let α be the level of v before augmentation, and let β be the level of v after augmentation. We want to prove that $\alpha \leq \beta$. Suppose by contradiction that $\beta < \alpha$.

Let u be the penultimate vertex in the shortest (s, v) path *after* the augmentation.



Then u is at level $\beta - 1$ after the augmentation. Note that $\beta - 1 < \alpha$ by assumption. Thus, by induction, the level of u was at most $\beta - 1$ before the augmentation as well. In particular, before augmentation, the level of u was (at least) two less than the level of v. This implies (u, v) was not in the residual graph before the augmentation. Since (u, v) is in the residual graph after augmenting and not before, (v, u) was in the augmenting path. But then, before augmenting, the level of v was one less than the level of u – a contradiction. □

The `shortest-augmenting-paths` algorithm computes the maximum flow f in $O(m^2 n)$ time. Perhaps one also desires a corresponding path packing. In this case, given the maximum flow f, one can decompose f into a path packing of at most m paths in $O(m^2)$ time. This task is left to the reader in exercise 18.3.

Note that the shortest augmenting path algorithm takes $O(m^2 n)$ time, no matter the magnitude of the capacities, assuming that basic arithmetic can be done in $O(1)$ time. Such an algorithm is said to run in *strongly polynomial time*. By contrast, *weakly polynomial time* algorithms run in time polynomial in the bit-complexity of the numerical input. Exercise 18.4 leads the reader through a weakly polynomial $O(m^2 + mn \log(n) \log(\lambda))$-time algorithm also from [EK72].

There have been many improvements in maximum flow and it remains an active area of research. For simplicity we restrict our attention to classical results.

18.4 Minimum cost metrics, linear programming and LP duality

18.4.1 Minimum fractional cuts

We can apply the same fractional perspective to the minimum cost (s, t)-cut problem. Recall that an (s, t)-cut contains at least one edge from every (s, t)-path. A *fractional (s, t)-cut* is a fractional combination of edges $y : E \to \mathbb{R}_{\geq 0}$ that contains (in sum) one unit of edges from every (s, t)-path. Edge costs $c(e)$ are now interpreted as the cost of one unit $y(e)$. The fractional relaxation of minimum cost (s, t)-cut is

$$\text{minimize } \sum_{e \in E} c(e) y_e \text{ over } y : E \to \mathbb{R}_{\geq 0} \text{ s.t. } \sum_{e \in p} y_e \geq 1 \text{ for all } p \in \mathcal{P}_{s,t}. \tag{18.2}$$

Note that any (s,t)-cut C corresponds to a $\{0,1\}^E$-indicator vector that is feasible to (18.2).

(18.2) has m variables but exponentially many constraints. This setup leads to the following situation. We encourage the reader to pause and consider the following question herself before reading on.

> *Suppose you were given a vector $y \in \mathbb{R}^E_{\geq 0}$. How would you verify, in polynomial time, that y is a feasible solution? In particular, how does one verify that for every (s,t)-path p, the sum of y_e's over $e \in p$ is at least 1? (Is it even possible?)*

The question is nontrivial because there is not enough time to enumerate every (s,t)-path. But let us reformulate the question slightly: verifying every path p has $\sum_{e\in p} y_e \geq 1$ is the same as verifying that the *minimum* $\sum_{e\in p} y_e$, over all $p \in \mathcal{P}_{s,t}$, is at least 1. Let us reinterpret the values $y : E \to \mathbb{R}_{\geq 0}$ as *edge lengths*. Then the covering constraint is really saying that the *length of the shortest (s,t)-path w/r/t edge lengths y_e, is ≥ 1*; we can verify this constraint by computing the shortest (s,t)-path w/r/t y. For this reason, (18.2) is called a *separable LP*, where the shortest path computation provides a *separation oracle*.

Equivalently, (18.2) can be stated as follows.

> *Find the minimum cost set of edge lengths $y : E \to \mathbb{R}_{\geq 0}$ subject to s and t having distance at least 1.*

This is a natural problem in its own right, and we call it the *(s,t)-minimum cost metric problem*. We write (Min-Metric) to denote both the problem and its optimum value.

18.4.2 Linear programs

The path packing and fractional cut problems above are both examples of *linear programs*. A linear program (LP) is an optimization problem of the form

$$\min/\max \langle b, x\rangle = \sum_{j=1}^{n} b_j x_j \text{ over } x \in \mathbb{R}^n$$
$$\text{s.t. } A_1 x \leq c_1,\ A_2 x = c_2, \text{ and } A_3 x \geq c_3.$$

where A_1, A_2, A_3 are matrices and b, c_1, c_2, c_3 are vectors. All LP's can be solved in polynomial time[2]. The algorithms are beyond our scope, but LP's are easy to

[2]More precisely, they are *weakly polynomial time solvable*, meaning the running times are polynomial in the bit complexity of the input.

understand and use as a black box. Modern solvers such as Gurobi are reliable in practice.

Packing LPs. A *packing LP* is a linear program of the form

$$\max \langle b, x \rangle \text{ over } x \in \mathbb{R}^n_{\geq 0} \text{ s.t. } Ax \leq c. \tag{P}$$

where $A \in \mathbb{R}^{m \times n}_{\geq 0}$, $b \in \mathbb{R}^n_{>0}$, and $c \in \mathbb{R}^m_{>0}$ all have nonnegative coefficients. We let $\textsc{Opt}(\mathsf{P})$ denote the optimum value of the LP (P).

The fractional path packing problem is our first example of a packing LP. For path packing, we have one variable for every path. Identifying edges and paths as coordinates, then, we have:

1. $b = \mathbb{1}^{\mathcal{P}_{s,t}}$, the all-ones vector in $\mathbb{R}^{\mathcal{P}_{s,t}}$
2. $c \in \mathbb{R}^E_{\geq 0}$ is the edge capacities.
3. $A \in \{0,1\}^{E \times \mathcal{P}_{s,t}}$ is the incidence matrix defined by

$$A_{e,p} = \begin{cases} 1 & \text{if } e \in p \\ 0 & \text{if } e \notin p, \end{cases}$$

for each edge $e \in E$ and path $p \in \mathcal{P}_{s,t}$.

Note that there are exponentially many variables in this LP so we could not even write it down in full in polynomial time, let alone apply a black box LP solver. Fortunately there are other ways to solve the LP (e.g., the *ellipsoid method* and *multiplicative weights update method*) where the LP does not have to be written out explicitly.

Covering LPs. A *covering LP* is a linear program of the form

$$\min \langle c, y \rangle \text{ over } y \in \mathbb{R}^m_{\geq 0} \text{ s.t. } A^T y \geq b, \tag{C}$$

where[3] $A \in \mathbb{R}^{m \times n}_{\geq 0}$, $b \in \mathbb{R}^n_{>0}$, and $c \in \mathbb{R}^m_{>0}$. We let $\textsc{Opt}(\mathsf{C})$ denote the optimum value of the LP (C).

The minimum cost metric problem above is our first example of a covering LP. For minimum cost metric, we have one variable/column for each edge, and one row/constraint for each (s,t)-path.

[3]Of course, in (C), we could have written A instead of its transpose A^T, and swapped b and c, which would more closely resemble (P). It is convenient for the subsequent discussion on LP duality for A, b and c to have the same dimensions in (P) and (C).

1. $c \in \mathbb{R}^E_{\geq 0}$ is the edge costs.
2. $b = \mathbb{1}^{\mathcal{P}_{s,t}}$ is the all-ones vector in $\mathbb{R}^{\mathcal{P}_{s,t}}$.
3. $A^T \in \{0,1\}^{\mathcal{P}_{s,t} \times E}$ is the $\{0,1\}$-incidence matrix defined by

$$A^T_{p.e} = A_{e,p} = \begin{cases} 1 & \text{if } e \in p \\ 0 & \text{if } e \notin p, \end{cases}$$

for each edge $e \in E$ and path $p \in \mathcal{P}_{s,t}$.

Note that A, b, c are the same between our two examples.

18.4.3 LP duality.

LP duality is about the relationship between the linear programs (P) and (C), in particular when the matrices and vectors A, b, c are the same for both problems. In this case (P) and (C) are said to be *dual* to one another.

Suppose we have a dual pair of (P) and (C); i.e., A, b, c refer to the same objects in either problem. Let $x \in \mathbb{R}^n_{\geq 0}$ be any feasible solution to (P) and let $y \in \mathbb{R}^n_{\geq 0}$ be any feasible solution to (C). We have

$$\langle b, x \rangle \overset{\text{(a)}}{\leq} \left\langle A^T y, x \right\rangle \overset{\text{(b)}}{=} \langle y, Ax \rangle \overset{\text{(c)}}{\leq} \langle y, c \rangle.$$

Here (a) is because $x \geq \mathbb{0}$ and $A^T y \geq b$. (b) is by definition of the transpose. (c) is because $y \geq \mathbb{0}$ and $Ax \leq c$. Thus, for a packing problem (P) and a covering problem (C) linked by duality, we have

$$\textsc{Opt}(\text{P}) \leq \textsc{Opt}(\text{C}).$$

If this argument seems familiar, it is because we saw it for packing and covering paths in Section 17.1.

We ask the same question for packing and covering LP's as we did for packing and covering paths. When, if ever, is $\textsc{Opt}(\text{P}) = \textsc{Opt}(\text{C})$? The all-important *LP duality theorem* (here restricted to packing and covering problems) states that in fact they are always equal.

Theorem 18.5 (LP Duality for packing and covering)**.** $\textsc{Opt}(\text{P}) = \textsc{Opt}(\text{C})$.

We note that Theorem 34.3 holds even if A, b, and c have negative coefficients.

(MaxFlow) is a packing LP and (Min-Metric) is a covering LP, and they are dual to one another. The LP duality theorem gives

$$\text{(MaxFlow)} = \text{(Min-Metric)}.$$

Now, as (Min-Metric) is a linear relaxation of (Min-Cut), we automatically have (Min-Metric) $\leq$ (Min-Cut). But we also know that both values are equal to (MaxFlow), giving (Min-Metric) = (Min-Cut). That is, allowing for "fractional" cuts does not improve the optimum solution.

Corollary 18.6. (MaxFlow) = (Min-Metric) = (Min-Cut); *moreover, there is always an integral optimum solution to* (Min-Metric)*.*

Max-flow min-cut is LP duality in its most canonical and intuitive form. The reader who understands why max-flow equals min-cut already understands the essence of LP duality.

18.5 Additional notes and references

This chapter overlaps with [KT06, Chapter 7] and [Eri19, Chapter 10]. The algorithm derived in exercise 18.4 is also from [EK72].

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

18.6 Exercises

Exercise 18.1. Recall the `augmenting-paths` algorithm for flow which augments along arbitrary augmenting paths. Design an (s,t)-flow problem with integer capacities where this algorithm may take exponentially many iterations to terminate.

Exercise 18.2. Consider an instance $(G = (V, E), s, t, c)$ of (s,t)-maximum flow with integral capacities, and suppose you have already computed a maximum integral flow f.

1. Suppose we increase the capacity of an edge by 1. Design and analyze an algorithm, as fast as possible, to compute the maximum flow in the updated graph.

2. Suppose we decrease the capacity of an edge by 1. Design and analyze an algorithm, as fast as possible, to compute the maximum flow in the updated graph.

Exercise 18.3. Suppose you had an (s,t)-flow f. We know that there exists an (s,t)-path packing of the same size as f; here we are interested in algorithms that take f and compute such a path packing. Such a path packing is called a *flow decomposition* of f.

Design and analyze an algorithm that, in $O(m^2)$ time, computes a maximum path packing x of the same size as f, such that:

1. There are at most m distinct paths (with nonzero value) in x.

2. If f is integral, then x is also integral.

Exercise 18.4. This exercise develops a $O((m^2 + mn\log(n))\log(\lambda))$-time algorithm for maximum (s,t)-flow and builds on ideas from exercise 18.3.

1. Prove the following: Given any (s,t)-flow problem with max flow value $\lambda > 0$, there exists an (s,t)-path where the minimum capacity edge is at least λ/m.

2. Describe a $O(m + n\log(n))$-time algorithm to find the path described above.[4]

3. Based on the two parts above, design and analyze an (s,t)-max flow algorithm that runs in $O(m(m + n\log(n))\log(\lambda))$ time for integer capacities, where λ denotes the value of the maximum flow.[5][6] (The algorithm does not know the true value of λ *a priori*.)

Exercise 18.5. Let $G = (V, E)$ be a directed graph with integer edge capacities $c : E \to \mathbb{N}$. Let each edge be assigned a color `red`, `white`, or `blue`. Recall that an *American walk* is a walk where the edges alternate `red`, `white`, `blue`, `red`, `white`, `blue`, ... Here the first color may be any color; all that matters is that we keep cycling through the colors patriotically. Given $s, t \in V$, consider the problem of computing the size of a maximum packing of American (s,t)-walks.

(Here the definitions are just like for (s,t)-paths. A "packing of walks" is a collection of walks that, for each edge e, uses the edge e at most as many times as its capacity. The "size of a packing" refers to the number of walks.)

[4] $O(m \log n)$ time is a little easier and this running time would still get partial credit. Even if the $O(m + n\log(n))$-running time eludes you, you can assume it as a black box for the next part.

[5] This is polynomial with respect to the bit complexity of the input.

[6] A possibly helpful bit of math: for (small) $\epsilon > 0$, $\log_{1+\epsilon}(x) \leq O(\log(x)/\epsilon)$ is a good approximation (which you may want to verify for yourself).

Chapter 19

Applications of flows and cuts



There is a rich variety of problems that can be solved by maximum flow, edge-disjoint paths, and minimum cut. We survey a few of these problems and their reductions here. To encourage the reader to attempt to find the reductions themselves, this chapter is organized as follows. We first introduce each problem. Then we revisit each of these problems and show how solve them by reductions to flows and cuts.

19.1 The problems

19.1.1 Vertex disjoint paths, vertex capacitated flow, and vertex cuts

Thus far we have discussed edge-disjoint paths and edge-capacitated flows. However one can also model flow problems where there are constraints on the vertices. For example, it is natural to ask for the maximum number of *vertex-disjoint* (s,t)-paths. In the dual, a *vertex* (s,t)*-cut* refers to a set of vertices whose removal disconnects s from t. Here we assume that s and t are not connected by an edge.

For example, in the following graph, there are two edge-disjoint paths from s to

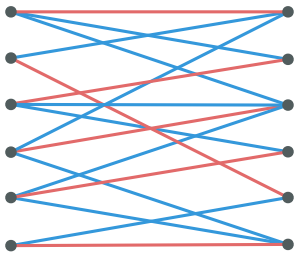
t, and the minimum edge-cut is 2. But there is only one vertex disjoint path, and a vertex cut of size one (circled in the middle).



One can also consider combinations of vertex and edge capacities. This is a reasonable constraint in communication networks, where each node represents a switch or router. The edge capacities reflect how much data can be carried through each cable. The vertex capacities model how much data can be passed through each router.

19.1.2 Bipartite matching

Let $G = (V, E)$ be a bipartite graph. That is, suppose there is a partition of V into two sets L, R (mnemonics for "left" and "right"), such that every edge $e \in E$ has exactly one endpoint in L and one endpoint in R.



A *matching* is a set of edges $M \subseteq E$ where no two edges share an endpoint. The *bipartite matching* problem is to compute the maximum cardinality matching.

19.1.3 Project selection

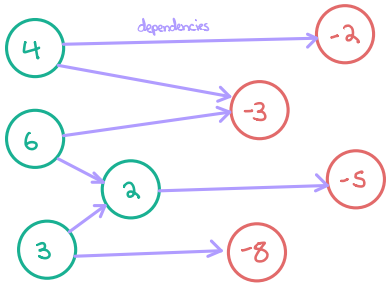
Here we consider a generic planning problem called *project selection*. The high level idea is that there are many possible projects that we can undertake. Some bring profit and others are net loss. What makes this problem interesting is that they depend on each other in complicated ways. Perhaps project A has to be completed before project B, and projects A and B both have to completed before project C. The goal is to choose a maximum profit subset of projects, where for each selected project, we have to also include the other projects that it depends on. Formally the input is as follows.

1. A set of projects, P.

2. For each project $p \in P$, a value $\mu(p) \in \mathbb{R}$ that measures the net profit of the project.

3. For each project $p \in P$, a set of *precedence constraints* $R(p) \subset P$, which reflects the set of projects that also have to be undertaken before p.

The following is a visual example where each vertex corresponds to a project. Each edge reflects a dependency – an edge (x, y) indicates that project x depends on project y.



We assume that the precedence constraints $R(p)$ induce a directed graph that is acyclic. (A directed cycle would give a list of projects $x_1, \ldots, x_k$ where each x_i has to be completed before x_{i+1} for each i, but also x_k has to be completed before x_1 – which describes a deadlock from a project planning point of view.) A variant of the problem that allows for cycles, so to speak, is given in exercise 19.4.)

19.1.4 Playoff Elimination

Imagine we are nearing the end of a baseball season, and several teams are competing for one spot in the playoffs. We want to know if our favorite team has a chance of making the playoffs. Suppose we want to know if the Boston Red Sox have been eliminated from the playoffs, and that standings are as follows.

Team	Wins	Games left	
New York	92	2	
Baltimore	91	3	
Toronto	91	3	
Boston	90	+ 2	92

Since Boston has 2 games, it is possible for them to win both and catch up to the Yankees. But we still have to ensure that the Yankees don't win, and that neither Baltimore nor Toronto win more than 1 game each. That can get tricky if these teams are playing with each other. For example, suppose the remaining schedule specifies the following games between the following four teams:

(wins)	(92) NewYork	(91) Baltimore	(91) Toronto	(90) Boston
NewYork	×	1	1	0
Baltimore	1	×	1	1
Toronto	1	1	×	1
Boston	0	1	1	×

A close examination reveals it is impossible for Boston to catch up to first place: assuming Boston wins all their games, and the Yankees lose all their games, (which are both obviously necessary), the winner of the game between Baltimore and Toronto will end up with 93 wins.

Can we automate such an analysis? In general, the input is as follows.

1. A set of teams T.
2. For every team $t \in T$, the number of wins $w(t)$ that they already have.
3. For every pair of teams $s, t \in T$, the number of games $g(s,t)$ between them.
4. The home team $h \in T$.

The goal is to figure out if there's any set of outcomes to the remaining games so that the home team h is at least tied for the most wins.

19.1.5 Assignment

Bipartite matching and generalizations of bipartite matching help model *assignment problems*. Consider for example the following example from Kuhn [Kuh55]. We have m individuals and n jobs; the goal is to assign individuals to all the jobs. However not all individuals are qualified for all jobs. What we use to encode these qualifications is via an $\{0,1\}^{m \times n}$ *qualification matrix*. Below we reproduce a 4×4 example from

[Kuh55].

$$Q = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The qualification matrix states that individual 1 is qualified for jobs $1, 2, 3$; individual 2 is qualified for job 3, and individuals 3 and 4 are both qualified for jobs 4. The assignment problem asks, to quote [Kuh55]:

> *What is the largest number of jobs that can be assigned to qualified individuals (with not more than one job assigned to each individual)?*

More general assignment problems. A more general version of the assignment job might, for example, state that certain jobs require a certain number of slots to be filled. Suppose that in addition to the qualification matrix above, we also have two sets of cardinalities:

1. $a_i \in \mathbb{N}$ for each individual $i \in [m]$, which says that individual i can take on at most a_i jobs.
2. $b_j \in \mathbb{N}$ for each job $j \in [n]$, which says that we have b_j slots of job j to try to fill.

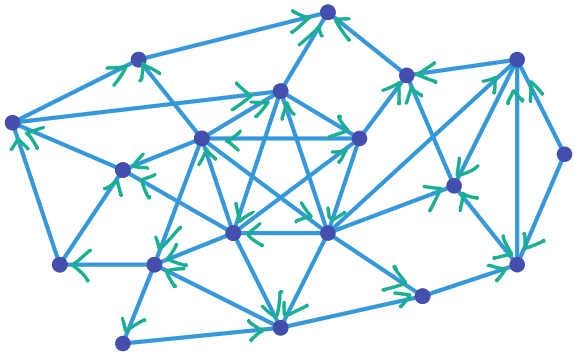
A particular individual can only be assigned to a particular job once.

The "simple" assignment version above was the case where $a_i = b_j = 1$ for all i and j.

This more general version of the assignment problem asks:

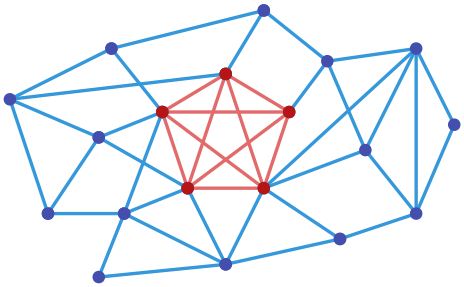
> *What is the largest number of jobs (counted with multiplicity) that can be assigned to qualified individuals?*

19.1.6 Edge orientation



Let $G = (V, E)$ be an undirected graph. The *edge orientation* problem is to assign directions to every edge $e \in E$ while minimizing the maximum in-degree in the resulting directed graph. There is also a parametrized version of the problem where one is given a parameter $h \in \mathbb{N}$, and the goal is to compute an orientation where the maximum degree is at most h (or declare that this is impossible).

19.1.7 Densest subgraph



Let $G = (V, E)$ be an undirected graph. The *density* of a graph G is defined as the ratio

$$(\text{density of } G) = \frac{|E|}{|V|}.$$

The *densest subgraph problem* is to compute the subgraph of G with greatest density. Equivalently, we want to select a set of vertices S maximizing the ratio $|E_S|/|S|$, where we let $E_S = \{e \in E : e \subseteq S\}$ denotes the set of edges with both endpoints in S.

Among other applications, densest subgraphs arise in the analysis of social networks, where a very dense subgraph can be interpreted as a tight-knit community.

19.2 Reductions to flows and cuts

We now explain how to solve each of the problems introduced above.

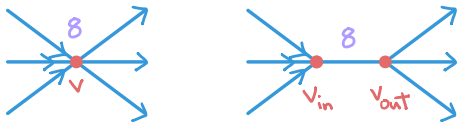
19.2.1 Vertex disjoint paths and vertex capacities

Recall the problem of vertex-capacitated flow from Section 19.1.1.

Theorem 19.1. *Let $G = (V, E)$ be a directed graph with positive vertex capacities $c : V \to \mathbb{R}_{>0}$. Let $s, t \in V$ be a fixed source and sink, respectively, and suppose there is no edge from s to t.*

1. *The maximum vertex capacitated (s,t)-flow equals the minimum capacity (s,t)-vertex-cut.*
2. *If the capacities are integral, then the maximum vertex capacitated flow (s,t)-flow is attained by an integral flow.*
3. *The maximum vertex capacitated (s,t)-flow and the minimum capacity (s,t)-vertex-cut can be reduce to edge-capacitated flows and cuts in a graph with $O(m+n)$ edges and $O(n)$ vertices.*

Proof sketch. We reduce from vertex capacities to edge capacities with the following simple trick. For each vertex v with capacity $c(v)$, we split v into two vertices, "v-in" v^- and "v-out" v^+. All edges directed into v are instead directed into v^-, and all edges directed out of v are instead directed out of v^+. We add an edge from v^- to v^+ with capacity $c(v)$.



Meanwhile, for the "normal" edges, we assign each of them capacity $+\infty$, since there is no limit to the amount of flow on that edge.[1] We leave it to the reader to apply the ideas from edge capacitated flow to this auxiliary graph to give the claims. □

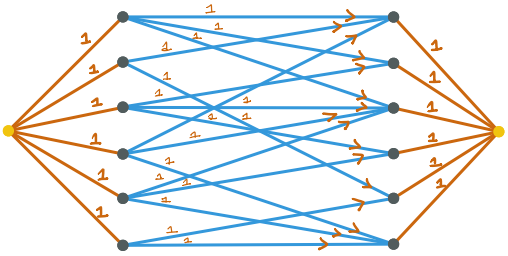
[1] This is particularly convenient for computing the minimum vertex (s,t)-cut, since the $+\infty$-capacity edges are always omitted.

19.2.2 Bipartite matching

Recall the bipartite matching problem from Section 19.1.2. We are given an undirected bipartite graph G and the goal is to compute the maximum cardinality matching.

Theorem 19.2. *The maximum cardinality matching reduces to an edge-disjoint paths problem with $O(m+n)$ edges and $O(n)$ vertices.*

Proof. The high-level ideas are sketched below.



Formally, suppose we are given a bipartite graph $G = (V, E)$, letting $L, R \subseteq V$ denote the two parts. We introduce a super-source vertex $s^\star$ on the left and a super-sink $t^\star$ on the right. We add an edge with capacity 1 from $s^\star$ to every $\ell \in L$ and an edge with capacity 1 from every $r \in R$ to $t^\star$. We also direct all the edges in E from L to R, and give each edge capacity 1.

Every path from $s^\star$ to $t^\star$ is of the form $(s^\star, \ell, r, t^\star)$, where $\ell \in L$, $r \in R$, and $\{\ell, r\} \in E$. For each edge $e = \{\ell, r\}$, let

$$p_e \stackrel{\text{def}}{=} (s^\star, \ell, r, t^\star)$$

denote that $(s^\star, t^\star)$-path via e. The mapping from $e \in E$ to p_e defines a bijection between E and $(s^\star, t^\star)$.

We claim that a set of edges $M \subseteq E$ is a matching iff the corresponding paths $\{p_e : e \in M\}$ is edge-disjoint. If so, then a maximum packing of edge-disjoint paths gives a maximum matching.

Let M be a matching. Since the endpoints of M are distinct, each vertex $\ell \in L$ appears in at most edge in M. Consequently the edge $(s^\star, \ell)$ appears in at most one path. Likewise for each $r \in R$, the edge $(r, t^\star)$ appears in at most one path. Lastly, we observe that each of the arcs (ℓ, r) appears in at most one path (namely, $p_{\{\ell,r\}}$). We conclude that the paths are edge-disjoint.

Conversely, suppose $\{p_e : e \in M\}$ is edge-disjoint. For each vertex $\ell \in L$, since $(s^\star, \ell)$ appears in at most one path p_e, ℓ is an endpoint to at most one edge in M. Similarly, for each vertex $r \in R$, since $(r, t^\star)$ appears in at most one path p_e, r is an endpoint to at most one edge in M. Thus M has distinct endpoints, and is a matching. □

19.2.3 Assignment

Recall the (simple) assignment problem from Section 19.1.5. We reduce the problem to edge-disjoint paths and the construction is essentially the same as bipartite matching.

1. We create a vertex ℓ_i for each individual $i \in [m]$, and a vertex r_j for each job $j \in [n]$.
2. We add a directed edge (ℓ_i, r_j) whenever individual i is qualified for job j.
3. We add a super-source vertex s, and a directed edge from s to ℓ_i for each $i \in [m]$.
4. We add a super-sink vertex t, and a directed edge from r_j to t for each job $j \in [n]$.

By construction, all (s,t)-paths are of the form (s, ℓ_i, r_j, t) for $i \in [m]$ and $j \in [n]$ such that individual i is qualified for job j. That is, we have a bijection between (s,t)-paths and pairs (i,j) where i is qualified for job j.

Now consider a family of edge-disjoint (s,t)-paths. For each $i \in [m]$, there is at most one path containing ℓ_i because ℓ_i has only one incoming edge. Similarly, for each $j \in [n]$, there is at most one path containing r_j. So the individual-to-job assignments corresponding to a family of edge-disjoint paths will feature each individual and each job at most once.

Conversely suppose we have an assignment where each individual and each job is assigned at most once. We claim the corresponding family of (s,t)-paths will be edge-disjoint. Indeed:

1. Each edge of the form (s, ℓ_i) is used at most once, because individual i is assigned at most once.
2. Each edge of the form (r_j, t) is used at most once, because each job j is assigned at most once.
3. Each edge of the form (ℓ_i, r_j) is used at most once because individual i is assigned at most once (or also because job j is assigned at most once).

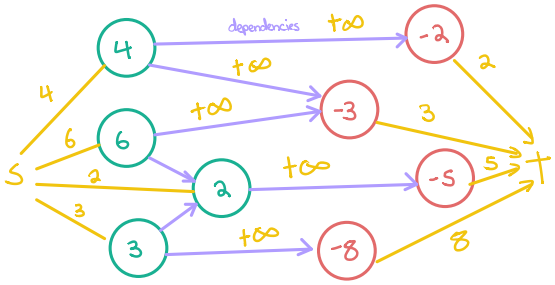
Thus there is a feasible assignment of size k iff there are k edge-disjoint (s,t)-paths, and we can map one family to the other as described above. We run the augmenting paths algorithm to get a maximum family of (s,t)-edge disjoint paths, and extract the assignment as described above. This takes $O(mn \min\{m,n\})$-time because there are at most $O(mn)$ edges, and the maximum size of the flow (so to speak) is $\min\{m,n\}$.

19.2.4 Project selection

Recall the project selection problem from Section 19.1.3.

Theorem 19.3. *The project selection problem with n projects and m dependencies reduces to computing the (s,t)-min cut in a flow network with $O(m+n)$ edges and $O(n)$ vertices.*

Proof sketch. We reduce to (s,t)-min-cut in the following flow network. We introduce a source s and add a directed edge from s to every profitable project. The edge is given capacity equal to the profit. We introduce a sink t, and a directed edge from every negative profit project to t. Each such edge is given capacity equal to the (positive) cost of that project. We add a directed edge for every dependency with capacity $+\infty$.



Consider any set $S \subset V$ that contains s and excludes t, and the induced (s,t)-cut $\delta^+(S)$. If the capacity of the cut is finite, then for every project p in S, every dependency of p must also be in S. That is, S is a feasible project selection. Note that the minimum (s,t)-cut has finite capacity, since (for instance) $\delta^+(\{s\})$ has finite capacity.

Now suppose the capacity of the cut $\delta^+(S)$ is finite. All of the edges in the cut $\delta^+(S)$ are one of two types.

1. The edge from a source to a profitable project excluded from S.
2. The edge from a negative profit project in S to t.

Thus the capacity of the cut is exactly equal to

$$\left(\begin{array}{c}\text{sum of profits of positive}\\ \text{projects excluded from } S\end{array}\right) - \left(\begin{array}{c}\text{sum of costs of}\\ \text{negative projects in } S\end{array}\right).$$

We can rewrite this as

$$\begin{pmatrix}\text{total sum of profits}\\ \text{of all positive projects}\end{pmatrix} - \begin{pmatrix}\text{sum of profits of all}\\ \text{positive projects in } S\end{pmatrix} - \begin{pmatrix}\text{sum of costs of}\\ \text{negative projects in } S\end{pmatrix}$$
$$= \begin{pmatrix}\text{total sum of profits of}\\ \text{all positive projects}\end{pmatrix} - (\text{net profit of } S).$$

Note that the total sum of profits of all positive projects is fixed independently of S. Thus the minimum (s,t)-cut will maximize the net profit of the source side of the cut. □

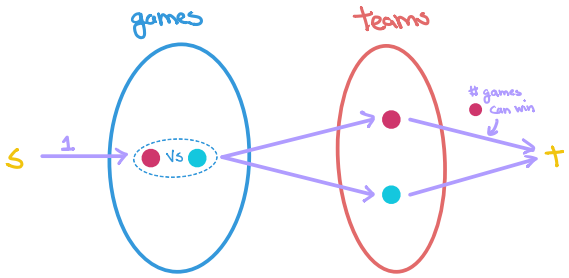
19.2.5 Playoff Elimination

Recall the playoff elimination problem from Section 19.1.4.

Theorem 19.4. *An instance of deciding playoff elimination with n teams, and m (unordered) pairs of teams with at least 1 game remaining, reduces to computing the maximum flow in a graph with $O(m+n)$ vertices and edges.*

Proof. We interpret playoff elimination as an allocation problem. We want to allocate one win from every game, while limiting the number of wins of all the non-home teams to keep the home team from being eliminated.

Let M be the maximum number of games the home team can win, if they win the rest of their games. We create a directed acyclic graph with 4 layers, as follows.



In the first layer we have a source s.

In the second layer we have a vertex $u_{\{a,b\}}$ for every pair of teams (a,b) with at least one game left to play. For each such pair (a,b), we add an edge from s to $u_{(a,b)}$ with capacity equal to the number of games left between teams a and b.

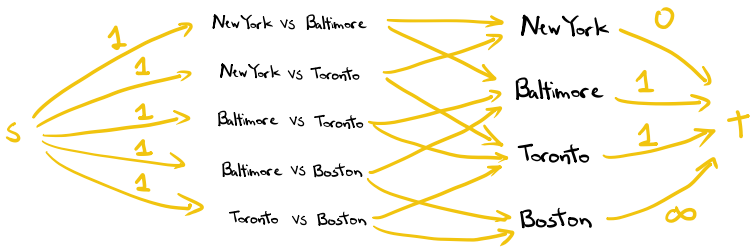
In the third layer we have a vertex v_a for every team a. For every team a, and every team b that a has left to play, we add an edge from the vertex $u_{\{a,b\}}$ to v_a with infinite capacity.

In the final layer we have a sink vertex t. For each team a except for the home team, we add an edge from v_a to t with capacity $M - w(a)$, where $w(a)$ is the number of wins team a already has. For the home team h, we add an edge from h to t with capacity $+\infty$.

Now, let G be the total number of games left to play. We compute the maximum flow in the graph described above. If the maximum flow has size G, then we declare

that the home team has not yet been eliminated. Otherwise we declare that the home team has been eliminated.

Below, we draw the flow network described above for the example described in Section 19.1.4.



We now prove that our reduction is sound. We first point out that the home team has not yet been eliminated iff there is a set of outcomes to all the games such that no other team has more than M wins. In one direction, suppose we have a set of outcomes where the home team makes the playoffs. We first reassign any game the home team loses so that the home team wins. The home team still makes the playoffs. Now the home team has M wins, and has at least as many wins as any other team, so this gives an outcome where no other team has more than M wins.

Conversely, suppose there is a set of outcomes where no other team has more than M wins. We can again reassign any game the home team loses so that the home team wins. Now the home team has M wins, while every other team still has at most M wins, giving a set of outcomes where the home team makes the playoffs.

Now we will show that there is a max flow of size G iff there is a set of outcomes where no other team wins more than M games.

Suppose first that there is an outcome of the games where every other team finishes with at most M wins. Changing the outcome of the home team's games if necessary, we may assume that the home team finishes with M wins. From these outcomes we construct a flow of size G that saturates all the edges from s, and where each matchup-vertex $u_{(a,b)}$ sends one unit of flow to team-vertex v_a for every game between a and b won by a, and one unit of flow to team-vertex v_b for every game won by b. Each team-vertex v_a gets an amount of flow equal to the number of games won by a, which is at most $M - w(a)$, and we send to t from the (v_a, t)-edge. This gives a feasible flow of size G in our auxiliary network.

Conversely suppose there is an (s,t)-flow of size G. We will construct an outcome of games where every team has at most M wins. We may assume the flow is integral because the capacities are integral, and decompose the flow as an integral path-packing of G paths. Each path is of the form $(s, u_{a,b}, v_a, t)$ for a pair of teams a and b. We also observe that the total capacity of the edges leaving s is G, which forces each

$(s, u_{a,b})$-edge to be saturated, hence the number of paths involving $v_{a,b}$ equals the number of games left between a and b.

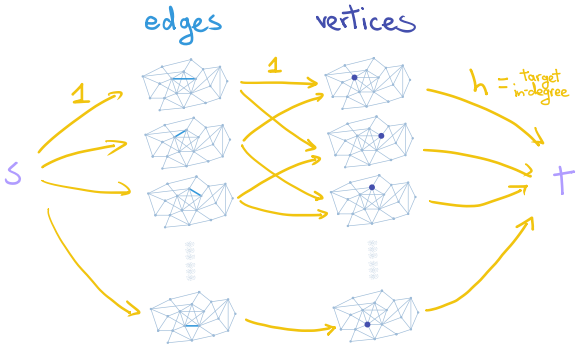
We create a set of outcomes where for each path $(s, u_{a,b}, v_a, t)$, we give one win from a matchup between a and b to a. This assigns an outcome to all G remaining games. Moreover, each team-vertex v_a can be in at most $M - w(a)$ paths since this is the capacity of the edge (v_a, t). Thus each team a ends up with at most M wins in this allocation. □

19.2.6 Edge orientation

Recall the edge orientation problem from Section 19.1.6.

Theorem 19.5. *Let $G = (V, E)$ be an undirected graph with m edges and n vertices, and $h \in \mathbb{N}$, and consider the problem of deciding whether there is an orientation with maximum in-degree of h. This problem reduces to an edge-disjoint paths problem with $O(m)$ edges and $O(n)$ vertices.*

Proof. We can re-interpret edge orientation as an assignment problem with capacities. "Orienting an edge" is like assigning an edge to one of its endpoints. We want to see edges to endpoints so that each vertex is assigned at most $O(h)$ edges. We create a flow network similar to assignment.



We initially have a bipartite graph where one side we have an auxiliary vertex a_e corresponding to each edge $e \in E$, and on the other side we have an auxiliary vertex b_v corresponding to each vertex $v \in V$. We have an edge from a_e to b_v whenever v is an endpoint of e. We introduce an auxiliary source s and an auxiliary sink t. We add an edge with capacity 1 from s to a_e for every e. We add an edge with capacity h from b_v to t for every vertex v.

After constructing this auxiliary graph, we compute a maximum path packing from s to t, and return `true` iff we obtain a path packing of size m.

To prove the reduction is sound, we show that there is a 1-1 correspondence between orientations with maximum in-degree h, and path-packings of size m. Given an orientation with maximum in-degree h, consider the path packing where for every edge e oriented toward an endpoint v, we take the path (s, a_e, b_v, t). This gives m paths that fits in the capacities; in particular, each edge of the form (b_v, t) is used at most h times because v has in-degree at most h in the orientation.

The inverse mapping takes path packings of size m and converts them into orientations with maximum in-degree h. We first observe that every path in a path packing is of the form (s, a_e, b_v, t), where v is an endpoint of e. Since there are m edges from s to distinct a_e's, in a path packing of m edges, there is exactly one path containing each a_e. For each e, we orient e towards the endpoint v where b_v is the next vertex in the corresponding path. This gives an orientation of every edge. Additionally, the maximum in-degree is at most h because that is the capacity of the edge (b_v, t).

This correspondence implies that there is an orientation with in-degree h iff there is a path packing of size m in the auxiliary graph, which completes the proof of correctness. □

Corollary 19.6. *Consider the problem of computing the minimum in-degree orientation in a graph G with m edges and n vertices. Then this problem can be solved via $O(\log n)$ instances of edge-disjoint paths with $O(m)$ edges and $O(n)$ vertices.*

Proof. In Theorem 19.5, we showed that the decision problem (given a target in-degree h) reduces to a single instance of edge-disjoint paths. We reduce the minimization problem (of finding the minimum feasible h) to $O(\log n)$ instances of the decision problem by binary search. Here observe that the optimum value of h is an integer between 1 and m. To "probe" a value of h, we apply Theorem 19.5 to see if there exists an orientation with maximum in-degree h. If so, then the optimum values is $\leq h$. If not, then the optimum value is greater. We guide our binary search accordingly. □

19.3 Additional notes and references

We recommend [KT06] and [Eri19] for further discussions on applications of flow (including several of the applications discussed here).

Spring 2026 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

19.4 Exercises

Exercise 19.1. Given integer values $b \in \mathbb{N}^V$, a *b-matching* is a set of edges $M \subseteq E$ where every vertex v is incident to at most $b(v)$ edges in M. Consider the problem of computing the largest b-matching in a bipartite graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.2. Every semester we have to schedule PSO's for all the students, which is complicated by (a) room availability mixed with (b) time availability both with respect to the students and the rooms. Suppose there are m total students, n available time slots, and p different rooms. Suppose we had the following information.

1. For each student (indexed by) $i \in [m]$, there is a subset of times $T_i \subseteq [n]$ of times when they can attend PSO.

2. For each time slot $j \in [n]$, there is a subset of rooms $R_j \subseteq [p]$ available at that time.

3. For each room $k \in [p]$ there is a maximum capacity C_k of the number of students that can fit in the room at any given time.

From this information we have the following scheduling problems.

1. Consider the problem of deciding if there is a way to schedule all the students to times and rooms while respecting all the constraints listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. In addition to making sure each room can fit all its students, we also want to minimize the maximum number of students in any room at any time, to decrease the maximum load on the TA's. Consider the problem of creating a schedule that minimizes the maximum number of students in any room, while also respecting the constraints listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.3. Suppose we are given a two-dimensional array $A[1..m][1..n]$ of non-negative real values such that each row and column sum is an integer. We want to round A to an integer matrix, replacing each entry $x \in A$ with either $\lceil x \rceil$ or $\lfloor x \rfloor$, while maintaining the sum in any row or column of A. For example,

$$\begin{pmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{pmatrix} \text{ rounds to } \begin{pmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{pmatrix}.$$

Consider the problem of computing such a rounding, or declaring that none exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.4. Recall the project selection problem in Section 19.1.3. Suppose we relax the problem so that the "precedence requirements" are no longer required to be acyclic. We instead interpret each set $R(p)$ as "co-requisites": if we take project p, then we also have to take on project q for every $q \in R(p)$. (That is, we remove the emphasis that every $q \in R(p)$ should be executed "before" p.) For this generalization of project selection, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.5. Let $G = (V, E)$ be an undirected graph with positive edge weights $w : E \to \mathbb{R}_{>0}$. We define the *weighted density* of a graph as the total edge weight divided by the number of vertices. Consider the problem of computing the weighted

densest subgraph of G, assuming the edge weights are all integral and between 1 and poly(n). For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.6. In the playoff elimination problem (Section 19.1.4), we asked whether a particular team was mathematically eliminated from making the playoffs. This allows the home team to win all of its remaining games, and other teams to lose all of their games, which may be unrealistic when there are many games left. Suppose we assumed that:

1. No team will win more than 90% of its remaining games, rounded up. (I.e., if a team has k games left, it can win at most $\lceil .9k \rceil$ of them.)
2. Every team will win at least 10% of its remaining games, rounded down. (I.e., if a team has k games left, it must win at least $\lfloor .1k \rfloor$ of them.)

Consider the problem of deciding whether the home team can make the playoffs under the additional restrictions listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.7. Evacuate the building! Before Purdue can begin construction on its next glorious computer science building, at least two times the size of Lawson and with at most half as many restrooms, the architectural plan has to pass a series of safety regulations. One of the most important safety requirements is having an *evacuation plan* where in the event of an emergency, one can empty all the rooms to safe locations without overcrowding any of the hallways and other pathways, and risking a stampede. Here we model the problem of computing a safe evacuation plan (or declaring that none exists) as a graph problem, as follows.

The input consists of a directed graph $G = (V, E)$, two disjoint subsets of vertices $X, Y \subseteq V$, and integer edge labels $z : E \to \mathbb{N}$. X represents the rooms that would need to be evacuated, and Y represents the safe locations that the rooms in X must be evacuated to. (One safe location can be the destination of any number of evacuation routes.) The edges represent hallways/pathways through the building, and each edge e is annotated with an integer label $z(e)$ that signifies the number of evacuation paths it can accommodate. We define an *evacuation plan* as a set of paths from rooms in X to safety locations in Y, with one path starting from each room in X. We say that an evacuation plan is *safe* if no hallway/pathway is used more than its declared limit $z(e)$.

The problem is to either compute a safe evacuation plan or declare that no safe evacuation plan exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.8. Recall that in chess, a *rook* (♖) is a piece that can move horizontally or vertically as far as it wants. (It cannot move diagonally.) We say that a rook on a particular square is *attacking* another chess piece at another square if the rook can be moved horizontally or vertically from its square to the square of that chess piece.

Consider the following problem which we call the "n-rooks problem". At a high level, there is an $n \times n$ board where some of the squares have been marked off as "unavailable". The remaining squares are "available". The goal is to place n rooks in the available squares so that none of the rooks are attacking each other. We call this a "non-attacking placement" of the n rooks. (While rooks cannot be placed on unavailable squares, they can still move over unavailable squares to attack other pieces on the other side of unavailable squares.)

Below we draw a non-attacking placement of 4 rooks on a 4×4 chess board. Unavailable squares are marked by `X`. Rooks are marked by ♖.

X	♖		X
♖	X	X	X
	X	♖	
X		X	♖

Formally, the input consists of an $n \times n$ bit-array $A[1..n, 1..n] \in \{\texttt{true}, \texttt{false}\}^{n \times n}$, where $A[i, j] = \texttt{true}$ indicates that square (i, j) is available, and $A[i, j] = \texttt{false}$ indicates that square (i, j) is unavailable. The goal is to decide if there is a non-attacking placement of n rooks in the available squares. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.9. This problem models the following scenario. To prepare for the final, you and your classmate have decided to try to solve all n of the exercises in the textbook. Due to time constraints, you've decided to split up the problems. To try to make the total effort even, each of you went through and skimmed all the problems, and for each problem, estimated the number of minutes it would take to solve the problem. (Thus producing two lists of estimates, one for each person.) The goal is to assign each exercise to you or your classmate so as to minimize the maximum time that either of you would spend solving their assigned problems. We call this problem the "optimum study allocation problem".

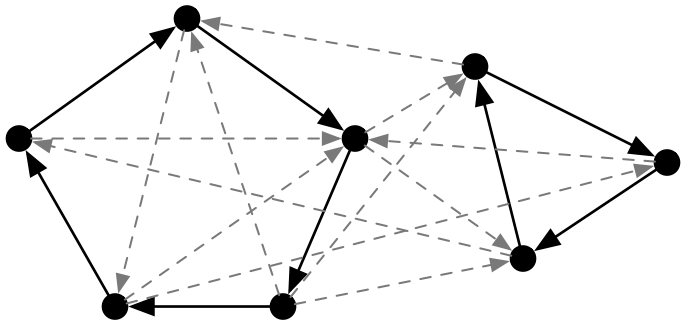
Formally, we let $\{1, \ldots, n\}$ represent the n exercises. The input consists of two lists of natural numbers $A[1..n]$ and $B[1..n]$. For $i = 1, \ldots, n$, $A[i]$ represents your estimated time for problem i and $B[i]$ represents your classmate's estimated time for problem i. The goal is to divide $\{1, \ldots, n\}$ into two sets, $S \subseteq \{1, \ldots, n\}$ and $T = \{1, \ldots, n\} \setminus S$, to minimize the maximum of

$$\sum_{i \in S} A[i] \text{ and } \sum_{j \in T} B[j].$$

Here $\sum_{i \in S} A[i]$ represents the total time you estimate to solve all the problems in S and $\sum_{j \in T} B[j]$ represents the total estimated time for your partner to solve all the problems in T.

For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.10. A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover every vertex in G. You can also think of a cycle cover as a subset of edges $F \subseteq E$ such that each vertex v has exactly one edge in F entering v, and exactly one edge in F leaving v. This implies that F has exactly n edges. Below is a directed graph where the solid arcs form a cycle cover, and the dashed arcs are the remaining edges in G.



Consider the problem of deciding if a directed graph G contains a cycle cover. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.11. You work at Build-A-Graph, which is sort of like Build-A-Bear, but for graphs. Your customers come in with graphs and you find them subgraphs based on certain specifications about the vertex degrees.

Formally, the input consists of a directed graph $G = (V, E)$, and two lists $d_{\text{IN}}, d_{\text{OUT}} : V \to \mathbb{Z}_{\geq 0}$. The goal is to compute a subgraph $H = (V, F)$ where each vertex $v \in V$ has in-degree $d_{\text{IN}}(v)$ and out-degree $d_{\text{OUT}}(v)$, or declare that no such subgraph exists. (Here "subgraph" means that $F \subseteq E$. You can assume that $\sum_{v \in V} d_{\text{IN}}(v) = \sum_{v \in V} d_{\text{OUT}}(v)$.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Chapter 20

Trees, flows, and cuts

We consider flow and cut problems for the particular setting of undirected graphs. It turns out that undirected graphs have substantially more symmetry than directed graphs, allowing for faster algorithms and very elegant ideas.

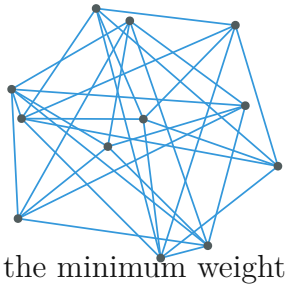
Two clean algorithmic bounds that are obtained in this chapter are as follows. To preface the first result, we mention that the global minimum cut is the minimum (s,t)-cut over all pairs $s,t \in V$.

1. In undirected graphs (with capacities), the global minimum cut can be computed in $O(mn + n^2 \log(n))$ time.

2. In undirected and uncapacited graphs, the maximum (s,t)-flow and the minimum (s,t)-cut can be computed in $O(m + n\lambda^2)$ time, where λ is the size of the maximum (s,t)-flow.

The first result is much faster than can be obtained by computing the minimum (s,t)-cut over all pairs $s,t \in V$. In fact it is even faster than the fastest algorithm we have analyzed for a single (s,t)-min cut, which is the $O(m^2 n)$ `shortest-augmenting-paths` algorithm from Chapter 18. The second result improves on the augmenting paths algorithm from Chapter 17. The improvement comes from connections to spanning trees (Chapter 16) and this connection is interesting in its own right.

20.1 Global minimum cut



In this section, let $G = (V, E)$ be an undirected graph, possibly with positive edge weights. We have previously discussed (s,t)-cuts, where $s,t \in V$, which are sets of edges whose removal disconnects s from t. We are now interested in *global cuts*, or simply *cuts*, which are set of edges whose removal leaves the graph disconnected. The *(edge) connectivity* of G is defined as the minimum weight

of any global cut. Consider the problem of computing the minimum global cut in G; for example, in the graph drawn above.

Of course we can reduce global cut to polynomially many (s,t)-cuts, by simply enumerating all pairs $(s,t) \in V$. If we let $\mathrm{MF}(m,n)$ denote the running time to compute max-flow in a graph with m edges and n vertices, then this approach would take

$$\binom{n}{2} \mathrm{MF}(m,n)$$

time to compute the global minimum cut. One can do better by observing that it suffices to fix s, and loop over all choices of t. Then we can compute the global minimum cut in $n\,\mathrm{MF}(m,n)$. Note that the best running time we have for $\mathrm{MF}(m,n)$ (independent of the size of the flow) is $O(m^2 n)$ via shortest augmenting paths (Section 18.3.2), which gives a $O(m^2 n^2)$-time algorithm for minimum cut.

In this section we analyze an algorithm that computes the global minimum cut in $O(mn + n^2 \log(n))$ time. The algorithm is given in Fig. 20.1 on Page 349. It is a deceptively simple, recursive algorithm that is also a bit mysterious. We first give a high-level description. In the base case there are only two vertices, and only one possible cut, which is returned. So let us assume that there are more than two vertices. The min-cut algorithm orders the vertices $v_1, \ldots, v_n$ by a certain greedy ordering. (More on this later.) It looks at the graph G' obtained by contracting v_{n-1} and v_n to a single vertex. It recurses on G' to produce a cut $\delta(S)$ that (when lifted to G) does not separate v_{n-1} and v_n. It compares this cut to the singleton cut $\delta(v_n)$, and returns the better of the two cuts. Figure 20.2 gives an example of the algorithm on an unweighted graph with 8 vertices and a minimum cut of size 2.

The $O(mn + n^2 \log n)$ running time is fairly transparent: The ordering takes $O(m + n \log n)$, and each recursive call decreases the number of vertices by 1. Overall the bottleneck is computing n orderings which takes $O(m + n \log n)$ time. On the other hand, it is not all clear why the algorithm should produce the minimum cut.

We first introduce some convenient notation. We let $\lambda(G)$ denote the edge connectivity of a graph G. For two vertices $u, v \in V$, the *(edge) connectivity of* $\{u,v\}$ is the weight of the minimum (u,v)-cut. We let $\lambda(u,v)$ be the connectivity between u and v.

The proof of correctness rests on the following two claims.

Claim 1. $\lambda(G) = \min\{\lambda(v_{n-1}, v_n), \lambda(G')\}$.

Claim 2. $\lambda(v_{n-1}, v_n) = \deg(v_n)$.

`mysterious-min-cut`$(G = (V, E, c))$

/ A mysterious algorithm that computes the global minimum cut. */*

1. If $|V| = 2$, then return the unique cut separating the two vertices.
   // Base case, $O(1)$.
2. Let $v_1 \in V$ be any vertex. For $i = 2, \ldots, n$:
   / Entire loop takes $O(m + n \log n)$ time w/ Fibonacci heaps. */*
   A. Let $v_i \in V$ maximize, over all $v \in V \setminus \{v_1, \ldots, v_{i-1}\}$, the quantity $\sum_{e=\{v,v_j\} \in E : j < i} c(e)$.
3. Let $G' = (V', E')$ be the graph obtained by contracting $\{v_{n-1}, v_n\}$ to a single vertex.
4. Recursively call `mysterious-min-cut`(G') to obtain a cut of the form $\delta(S)$, where $S \subseteq \{v_1, \ldots, v_{n-2}\}$.
5. Compare $\delta(S)$ to the singleton cut $\delta(v_n)$, and return the smaller of the two cuts.

Figure 20.1: An algorithm computing the global minimum cut in $O(m(m + n \log n))$ time in weighted, undirected graphs.

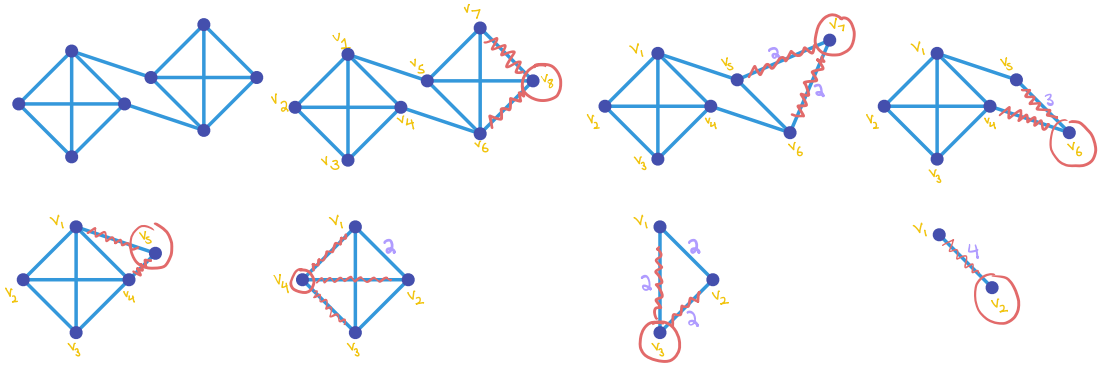


Figure 20.2: Tracing the execution of `mysterious-min-cut` on an initially unweighted graph (from left to right, top to bottom). The true minimum cut (of size 2) is identified in the fifth graph (first graph, second row).

The first claim can be proven right now: either the minimum cut separates v_{n-1} and v_n, or it doesn't, in which case the minimum cut in G' has the same value as the minimum cut in G. The second claim is more mysterious, and most of this section is in the service of proving it.

If the second claim does hold, then we can prove the algorithm correctly computes the minimum cut by induction on the number of vertices. In the base case, $n = 2$, and the algorithm returns the only cut. In the general case, $n > 2$, the minimum cut is either the minimum (v_{n-1}, v_n)-cut, or the minimum cut in G'. In the former scenario, the second claim above implies that it is the singleton cut $\delta(v_n)$. In the latter scenario, by induction on n, the recursive call computes the minimum cut in G'.

Assuming the second claim, we have shown the following theorem.

Theorem 20.1 ([NI92b])**.** *The global minimum cut in a weighted, undirected graph can be computed in $O(mn + n^2 \log n)$ time.*

20.1.1 Maximum adjacency (MA) orderings.

To analyze the min-cut algorithm, we draw attention to the ordering of vertices produced in `(2)`. Nagamochi and Ibaraki [NI92b; NI08] call this a *maximum adjacency ordering*, defined formally as follows.

Definition 20.2. *Let $v_1, v_2, \ldots, v_n$ be an ordering of V. We say that the ordering is a* maximum adjacency ordering *(abbrev.* MA ordering*) if for each $i > 1$, v_i maximizes the weight of edges from v_i to $\{v_1, \ldots, v_{i-1}\}$ over all vertices in $V \setminus \{v_1, \ldots, v_{i-1}\}$.*

We note that computing an MA ordering takes $O(m + n \log n)$ time, by using a Fibonacci heap (cf. fact 7.2) to track the remaining vertex with the maximum weight of edges to previously selected vertices. We also note that in unweighted graphs an MA ordering. We state the running times as a lemma below and defer the algorithmic details to Section 20.A. On the following page we rewrite "`mysterious-min-cut`" in terms of MA-orderings; the algorithm is also renamed as `MA-min-cut`.

Lemma 20.3. *In an unweighted graph, a maximum adjacency ordering starting with any initial vertex v_1 can be computed in $O(m + n)$ time. In a weighted graph, a maximum adjacency ordering starting with any initial vertex v_1 can be computed in $O(m + n \log n)$ time.*

Here, then, is the key "claim 2", restated in terms of MA-orderings, and its proof.

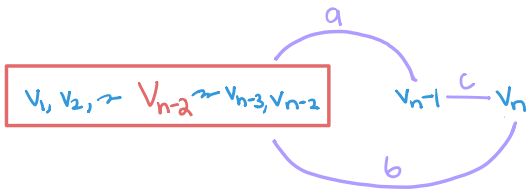
Lemma 20.4. *Let $v_1, \ldots, v_n$ be a maximum adjacency ordering. Then $\lambda(v_{n-1}, v_n) = \deg(v_n)$.*

`MA-min-cut`$(G = (V, E))$

1. If V has only two vertices then return the unique cut separating them. // $O(1)$.
2. Compute an MA-ordering $v_1, \ldots, v_n$. // $O(m + n \log n)$ *w/ Fibonacci heaps.*
3. Let $G' = (V', E')$ be the graph obtained by contracting $\{v_{n-1}, v_n\}$ to a single vertex.
4. Recursively call `MA-min-cut`(G') to obtain a cut of the form $\delta(S)$, where $S \subseteq \{v_1, \ldots, v_{n-2}\}$.
5. Compare $\delta(S)$ to the singleton cut $\delta(v_n)$ and return the smaller of the two cuts.

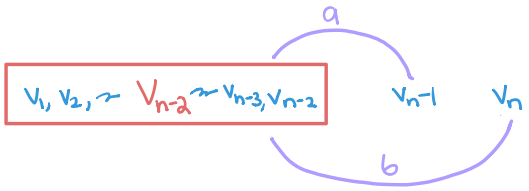
Figure 20.3: The min-cut algorithm from Fig. 20.1 rewritten in terms of MA orderings.

Proof. The proof is by induction on n, where the base case $n = 2$ is immediate. Suppose $n > 2$.



Let $V_{n-1} = \{v_1, v_2, \ldots, v_{n-2}\}$. Let a denote the total weight of edges from V_{n-2} to v_{n-1}, let b denote the total weight of edges from from V_{n-2} to v_n, and let c denote the weight of the edge $\{v_{n-1}, v_n\}$ (if it exists, or otherwise 0). We have two cases depending on whether $c = 0$ or $c > 0$.

Case 1: $c = 0$.



In this case, the claim is that $\lambda(v_n, v_{n-1}) = b$. Clearly it is $\leq b$, since this is the size of the singleton cut $\delta(v_n)$. We need to show that $\lambda(v_n, v_{n-1}) \geq b$.

1. Let $G_0 = G - v_n$ and let λ_0 denote the edge connectivity in G_0. Observe that $v_1, \ldots, v_{n-1}$ is a maximum adjacency ordering for G_0.
2. Let $G_1 = G - v_{n-1}$ and let λ_1 denote the edge connectivity in G_1. Observe that $v_1, \ldots, v_{n-2}, v_n$ is a MA ordering for G_1.

We have

$$\lambda(v_n, v_{n-1}) \geq \min\{\lambda(v_n, v_{n-2}), \lambda(v_{n-2}, v_{n-1})\}$$

by the triangle inequality for cuts (Cf. exercise 17.2). We claim that both $\lambda(v_n, v_{n-2})$ and $\lambda(v_{n-2}, v_{n-1})$ are $\geq b$. For the first, we have We have

$$\lambda(v_{n-2}, v_{n-1}) \overset{(a)}{\geq} \lambda_0(v_{n-2}, v_{n-1}) \overset{(b)}{=} a \overset{(c)}{\geq} b,$$

as desired. (a) is because deleting v_n and its incident edges cannot increase the connectivity. (b) is by induction on n. (c) is because of the maximal MA ordering.

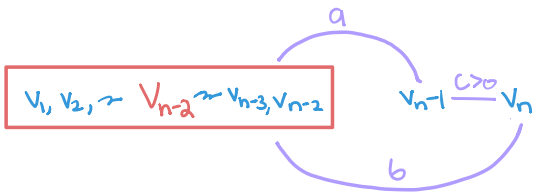
For $\lambda(v_{n-2}, v_n)$, we have

$$\lambda(v_{n-2}, v_n) \overset{(d)}{\geq} \lambda_1(v_{n-2}, v_n) \overset{(e)}{=} b,$$

as desired. Similar to before, (d) is because deleting v_{n-1} and its incident edges cannot increase the size of the max $(v_{n-2}v_n)$-flow. (e) is by induction on n.

This completes the proof for the case $c = 0$.

Case 2: $c > 0$.



Let G_0 be the graph obtained by deleting the (v_{n-1}, v_n) edge. Let λ_0 denote the edge connectivities in G_0. Observe that $v_1, \ldots, v_n$ is a MA ordering in G_0. We have

$$\lambda(v_n, v_{n-1}) \overset{(f)}{=} c + \lambda_0(v_n, v_{n-1}) \overset{(g)}{=} c + b.$$

(f) is because the edge $\{v_{n-1}, v_n\}$ is in every $\{v_{n-1}, v_n\}$-cut, and remove that edge leaves the graph G_0. (g) is by the previously analyzed case, $c = 0$. □

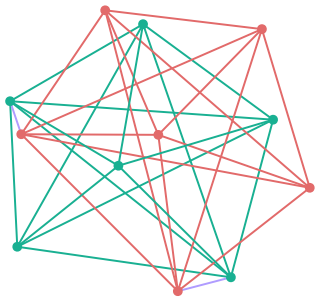


Figure 20.4: A cut of size 2 in the graph from Page 347.

20.2 Sparsification and faster augmenting paths

Sparsification refers to the general method of taking a large, dense graph and producing a smaller, sparse graph (over the same vertex set) that preserves some desired structure of the original graph.

We start with a very familiar example. Recall that two vertices s and t are *connected* in an undirected graph if there is a path from s to t. Given an undirected graph $G = (V, E)$, suppose we wanted a sparse subgraph $G' = (V, E')$ such that any two vertices $s, t \in V$ are connected in G iff they are connected in G'. Here there is a solution G' with (slightly less than) n edges. The reader probably knows the answer and should pause to find it.

20.2.1 Small subgraphs preserving small connectivities

Recall that $\lambda(s, t)$ denotes the edge connectivity between s and t; i.e., the size of the minimum (s, t)-cut. We write $\lambda_G(s, t)$ to specifically refer to the connectivity in graph G, when the choice of graph is ambiguous.

Above, we asked for sparsifiers that maintain all pairwise connectivities in an undirected graph up to connectivity 1. (The answer was a spanning forest.) Here we generalize to connectivities up to a fixed cardinality $k \in \mathbb{N}$. More precisely, given an undirected graph $G = (V, E)$ and a parameter k, we will compute a subgraph H with less than kn edges such that for all $s, t \in V$,

$$\lambda_H(s, t) \geq \min\{k, \lambda_G(s, t)\}. \tag{20.1}$$

To construct such a graph H, Nagamochi and Ibaraki [NI92a] proposed the following simple greedy algorithm. Consider the partition of E into forests $F_1, F_2, F_3, \ldots$ constructed as follows.

`greedy-forest-packing`$(G = (V, E))$

1. $G_0 \leftarrow G$
2. for $i = 1, 2, \ldots$ until G_{i-1} is empty
   A. $F_i \leftarrow$ spanning forest of G_{i-1}
   B. $G_i \leftarrow G_{i-1} - F_i$
3. Return the forests $F_1, F_2, \ldots$

Here "$G_{i-1} - F_i$" is a shorthand for removing the edges of F_i from the graph G_{i-1}.

This construction is sometimes called a *greedy forest packing*; we are iteratively packing forests in the graph, and each forest in sequence is maximal (and spanning) among the remaining edges. It is not hard to see that the follow process has the same effect .

`greedy-forest-packing`$(G = (V, E))$

1. Set $F_i = \emptyset$ for all $i \in \mathbb{N}$.
2. For each edge $e \in E$ (in any order):
   A. Let F_i be the first forest such that $e \notin \text{span}(F_i)$.
   B. $F_i \leftarrow F_i + e$.
3. Return the forests $F_1, F_2, \ldots$.

For each $k \in \mathbb{N}$, let $H_k = F_1 \cup \cdots \cup F_k$ denote the union of the first k forests, as a subgraph of G. Note that H_k has at most $k(n-1)$ edges. We claim that for each k, H_k preserves connectivities up to k in the sense of (20.1).

Theorem 20.5 ([NI92a])**.** *Let $s, t \in V$ have connectivity ℓ in G. Then for all $k \leq \ell$, s and t have connectivity $\geq k$ in H_k.*

The key observation to proving Theorem 20.5 is the following.

Lemma 20.6. *Let $C = \delta(S)$ be an induced cut with $|C| = \ell$ edges, and let $k \leq \ell$. Then $|C \cap H_k| \geq k$.*

Proof. Suppose not. Then there is is an edge $e \in C \setminus H_k$. For each $i \in [k]$, if $e \notin F_i$, then e is spanned by F_i, and there must be at least one edge in $C \cap F_i$. Therefore C has at least one edge in each F_i, hence at least k total in H_k - a contradiction. □

Now we prove Theorem 20.5. Let $s, t \in V$ have connectivity ℓ and let $k \leq \ell$. Every $\{s, t\}$-cut $C = \delta(S)$ has at least ℓ edges, and by Lemma 20.6, $|C \cap H_k| \geq k$, as desired.

Theorem 20.5 gives us our first nontrivial sparsifier. With $< kn$ edges, we can preserve all connectivities up to size k. Don't be fooled by its simplicity! This algorithm has inspired many more ideas.

20.2.2 Computing a forest packing

How long does it take to compute a greedy forest packing? Consider the second approach above, where each edge e is inserted into the first forest F_i that does not span e. Instead of trying $F_1, F_2, \dots$ in order, we can apply binary search for the first F_i. Each probe takes a forest F_i and an edge e and asks if F_i spans e. To serve this query, we can use the disjoint union data structure for each forest F_i. Applying the bounds given in Lemma 16.9, this gives a total running time of $O(m \log(n)\alpha(m,n))$, where $\alpha(m,n)$ is the inverse Ackerman function. However one can actually do better:

Theorem 20.7. *A maximal forest packing can be computed in $O(m)$ time.*

We will prove this at the end, in Section 20.3.

20.2.3 Speeding up Ford-Fulkerson in undirected graphs

Consider an instance of (s,t)-max flow in an unweighted, undirected graph. Suppose the maximum flow has size exactly λ; the `augmenting-paths` algorithm takes $O(m\lambda)$ time to compute the maximum flow. But we know that the first λ forests contain all the edges necessary to compute the maximum flow. If we knew the value of λ *a priori*, then we could obtain a running time of $O(m + n\lambda^2)$ by first computing the the greedy forest packing in $O(m)$ time, and then running Ford-Fulkerson in H_k.

Of course we do not know λ *a priori*. But we can feel it out incrementally, so to speak, by adding including one more forest from the greedy forest packing in each iteration. That is, we start with an empty graph $H = (V, \emptyset)$. In the first orientation, we add the first forest, F_1, to H, search for an augmenting path, and update the residual graph. Generally speaking, in the ith iteration, we add the ith forest F_i to (the running residual graph of) H before searching for the next path. If we do not find an augmenting path in the ith, then we know that our current flow f is the maximum flow of size $i-1$ in the union of the first i forests, H_i. If there was a bigger (s,t)-flow in input graph G, then there would have been a flow of size i in H_i. But there wasn't, so f is the maximum flow.

The advantage is that the ith iteration occurs within the first i forests, which has only $O(ni)$ edges. Consequently the ith iteration takes $O(ni)$ time. Summing over all i, we obtain the following.

Theorem 20.8. *In undirected and unweighted graphs, the maximum (s,t)-flow and minimum (s,t)-cut can be computed in $O(m + n\lambda^2)$ time, where λ denotes the value of the maximum flow / minimum cut.*

20.3 Linear time greedy forest packings

In this final section, we describe the linear time algorithm for computing the greedy forest packing alluded to in Section 20.2. Let us first present the algorithm up front, because it is breathtakingly simple. The algorithm takes as input an undirected graph and computes a maximum adjacency ordering. It then uses the ordering to assign edges to directly to different forests. The assignment seems almost arbitrary on first impression.

`MA-forests`$(G = (V, E))$

1. Let $v_1, \ldots, v_n$ be an MA-ordering of G. // $O(m)$
2. $F_1, F_2, \cdots \leftarrow \emptyset$
3. For $j = 1, \ldots, n$
   A. $k \leftarrow 1$
   B. For each edge $e = \{v_i, v_j\}$ w/ $i < j$, in increasing order of v_i:
      1. $F_k \leftarrow F_k + e$ and $k \leftarrow k + 1$
4. return $F_1, \ldots, F_m$

It is easy to see that the above algorithm runs in $O(m)$ time. It is not obvious that it correctly returns a greedy forest packing.

Having described the full code for `MA-forests`, let us set it aside and start designing an algorithm anew. We will first develop an algorithm that is not necessarily fast, but the correctness will be clear. We will then make observations that help us understand `MA-forests` as a fast implementation of the slower algorithm. We introduce the following helpful notation. For a set of vertices $S \subsetneq V$ and a vertex $v \in V \setminus S$, we denote the number of edges from v to S by

$$\deg(v \,|\, S) \stackrel{\text{def}}{=} (\#\text{ edges from } v \text{ to } S)$$

20.3.1 A single forest

Towards a recursive algorithm, let us first present a simple algorithm that generates a single spanning forest based on a given MA ordering. It is essentially the search

algorithm for MST, except without the weights, and processing the vertices in the given MA order.

`MA-forest(`$G = (V, E)$`, MA ordering` $v_1, \dots, v_n$ `of` V`)`

1. $F \leftarrow \emptyset$ and $S \leftarrow \{v_1\}$
2. For each v_i in increasing order of i:
   A. For each edge $e = \{v_i, v_j\}$ where $j > i$
      1. If v_j is not in S
         a. $F \leftarrow F + e$ and $S \leftarrow S + v_j$.
   B. Return F.

An equivalent algorithm would be as follows.

`MA-forest(`$G = (V, E)$`, MA ordering` $v_1, \dots, v_n$ `of` V`)`

1. $F \leftarrow \emptyset$.
2. For each v_j in increasing order of i:
   A. If there is an edge $e = \{v_i, v_j\}$ where $i < j$:
      1. Let $e = \{v_i, v_j\}$ minimize v_i.
      2. $F \leftarrow F + e$ and $S \leftarrow S + v_j$.
   B. Return F.

For the analysis, let $G = (V, E)$ be a fixed undirected graph. Let $v_1, \dots, v_n$ be a fixed MA-ordering of G. For each i, let $S_i = \{v_1, \dots, v_i\}$. Let $F =$ `MA-forest(`$G, v_1, \dots, v_n$`)` be the forest returned by `MA-forest`.

Lemma 20.9. *F is a spanning forest in G.*

Proof. To show that F is a forest, we observe that for every v_j, F contains at most one edge of the form $\{v_i, v_j\}$ where $i < j$. This implies there are no cycles. Indeed, if F had a cyclie, then the maximum index vertex v_j in the cycle would have at least two edges of the form $\{v_i, v_j\}$, where $i < j$.

Next we show that F is spanning. Restricting our attention to a single component of G, we may assume G is connected. We need to show that F is connected; i.e., F is a tree. Suppose by contradiction that F is not connected. Let v_i be the first vertex that is not connected to v_1 in F. Then $\deg(v_i \mid S_{i-1}) = 0$. As $v_1, \dots, v_n$ is a MA-ordering, this implies that $\deg(v_j \mid S_{i-1}) = 0$ for all $j \geq i$. That is, there are no edges between S_{i-1} and $V \setminus S_{i-1}$, a contradiction to G being connected. □

Lemma 20.10. *Let $G' = G \setminus F$ be the graph obtained by removing F. Then $v_1, \dots, v_n$ is a maximal adjacency ordering of G'.*

Proof. By analyzing each component separately, it suffices to assume that G is connected. In particular, we have $\deg(v_i \mid S_{i-1}) \geq 1$ for all i.

For any vertex v, let $\deg'(v \mid S_{i-1})$ denote the number of edges between v and S_{i-1} in G'. Observe that for $j > i$, if $\deg(v_j \mid S_i) > 0$, then we have $\deg'(v_j \mid S_i) = \deg(v_j \mid S_i) - 1$, since F takes exactly one edge between S_i and v_j (should one exist).

Now, let $i < j$. We need to show that

$$\deg'(v_i \mid S_{i-1}) \geq \deg'(v_j \mid S_{i-1}).$$

We may assume that $\deg'(v_j \mid S_{i-1}) \geq 1$ since the LHS is always nonnegative. In this case, we have

$$\deg'(v_i \mid S_{i-1}) = \deg(v_i \mid S_{i-1}) - 1 = \deg(v_j \mid S_{i-1}) - 1 = \deg'(v_j \mid S_{i-1}),$$

as desired. □

20.3.2 Recursively packing forests

Lemma 20.10 implies that we can reuse the initial MA order to greedily pack the second forest, the third forest, and so forth. The following algorithm reuses the same MA-ordering to compute a greedy forest packing.

`recursive-MA-forests(`$G = (V, E)$`, MA-ordering` $v_1, \dots, v_n$ `of` V`)`

1. if $E = \emptyset$ then return $\emptyset$
2. $F_1 \leftarrow$ `MA-forest(`$G, v_1, \dots, v_n$`)`
3. $G_1 \leftarrow G \setminus F_1$
4. $F_2, \dots, F_k \leftarrow$ `recursive-MA-forests(`$G, v_1, \dots, v_n$`)`
5. return $F_1, \dots, F_k$.

Lemma 20.11. *Let F_1 be a spanning forest of G, and let $F_2, \dots, F_k$ be a greedy forest packing of $G - F_1$. Then $F_1, \dots, F_k$ is a greedy forest packing of G.*

20.3.3 Shortcutting the recursion

Lemma 20.12. *Consider an edge $e = \{v_i, v_j\}$ where $i < j$. Then* `recursive-MA-forests` *places e in the kth tree for $k = \deg(v_j \mid S_{i-1})$.*

Proof. List the edges between S_i and v_j,= $e_1 = \{v_{i_1}, v_j\}, e_2 = \{v_{i_2}, v_j\}, \ldots, e_k = \{v_{i_k}, v_j\}$, in increasing order of v_{i_ℓ}. For $\ell = 1, \ldots, k$, e_ℓ is in F_ℓ. □

We see now that `MA-forests` is placing the edges in forests exactly as dictated by the above lemma. Thus it is really just an extremely efficient implementation of `recursive-MA-forests`, from which correctness follows.

20.4 Notes and references

The algorithms presented here are from Nagamochi and Ibaraki [NI92a; NI92b]. They are also covered in the book [NI08] by the same authors, which explores additional related topics. Our presentation here is arranged differently from these sources, but the ideas are essentially the same.

Lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

20.A Computing MA orderings in linear time in unweighted graphs

In unweighted graphs, an MA ordering can be computed in $O(m + n)$ time basically with appropriate data structures. The subroutine we present selects the vertices $v_1, v_2, \ldots$ one at a time, in MA-order. We maintain, for each vertex v not yet selected, the number of edges between v and the vertices among all the vertices not yet selected. This quantity is always an integer between 0 and m, and increasing.

We maintain two types of linked lists. For each $k \in \{0, \ldots, m\}$, we have a doubly linked list of all the remaining vertices with (exactly) k edges to the selected vertices. We also maintain a sorted, doubly linked list of all values of k where there is at least one remaining vertex with k edges to the selected vertices.

These doubly-linked lists allow us to retrieve the next vertex v_i in $O(1)$ time. Indeed, we can first get the maximum value of k for which there is at least one remaining vertex with k edges to the selected vertices. We can then look in the corresponding list to obtain one such vertex.

When we select v_i, we need to update the lists by (a) removing v_i, and (b) adding the edges from v_i to the remaining vertices $V - \{v_1, \ldots, v_i\}$ to their respective tallies.

Removing v_i means we remove v_i from its corresponding doubly linked lists. If v_i had k edges to previously selected vertices, and was the last vertex in its list, then we remove k from the list of values. Each of these operations take $O(1)$ time.

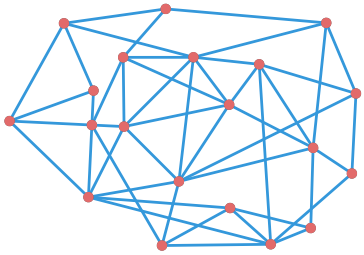
For each edge e from v_i to a remaining vertex u, we increase the counter representing the number of edges from u to selected vertices, say, from k to $k+1$. In doing so, we also remove u from the list of vertices with k edges to selected vertices and add u to the list of vertices with $k+1$ edges. If u was the last vertex was k edges to selected vertices, or becomes the only vertex with $k+1$ edges to selected vertices, then we need to update the list of values k accordingly. All of these operations take $O(1)$ time.

Putting it all together, identifying the next vertex v_i takes $O(1)$ time, and we spend $O(1)$ per edge between v_i and the remaining vertices to update our data structure. To order all the vertices, we end up processing each edge once. This gives a $O(m+n)$ running time overall.

Chapter 21

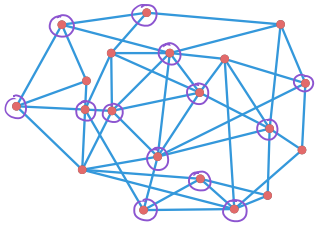
Matchings and vertex covers

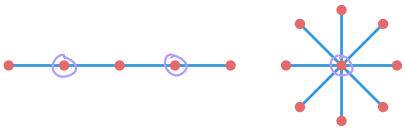
21.1 Packing and covering edges



Let $G = (V, E)$ be an undirected graph. A *vertex cover* is a set of vertices $U \subset V$ that contains at least one endpoint of every edge: $U \cap e \neq \emptyset$ for all $e \in E$.

The *minimum vertex cover* problem asks for a vertex cover of minimum cardinality[1]. On the right we circle a vertex cover for the graph drawn above. (Can you find a smaller one?) Below are two more examples where the minimum vertex covers are circled in purple.



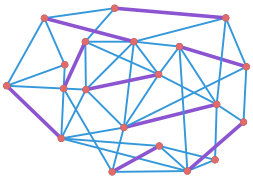


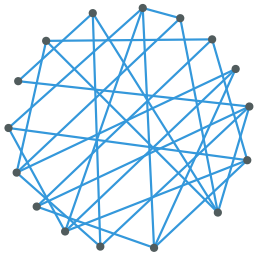
A *matching* is a set of edges, $M \subset E$, whose endpoints are distinct:

$$e \cap f = \emptyset \text{ for all distinct } e, f \in M.$$

[1]There are also weighted versions of maximum matching and minimum vertex cover, which we will not discuss in detail here.

One the right we highlight a matching in the graph drawn at the beginning of the section. The *maximum matching* problem asks for a matching of maximum cardinality. Consider the graph below: what is the maximum matching in this graph?





Duality. Matching and vertex cover are dual packing and covering problems. Matchings are packing edges into vertices. Vertex covers are covering edges with vertices. Similar to (s,t)-path packing before, one inequality between the sizes of a matching and a vertex cover is immediate.

Lemma 21.1. *Let M be a matching and let U be a vertex cover. Then $|M| \leq |U|$.*

Proof. Indeed,

$$|M| \overset{\text{(a)}}{\leq} \sum_{e \in M} |e \cap U| \overset{\text{(b)}}{=} \left| \bigcup_{e \in M} (e \cap U) \right| \leq |U|.$$

Here (a) is because every edge has at least one endpoint in U. (b) is because M is a matching, so the sets $\{e \cap U\}$ are disjoint across $e \in M$. □

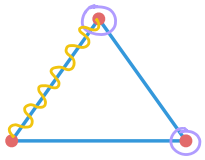
To emphasize the primal-dual setup:

edges pack into vertices, and vertices cover edges.

The big questions are:

1. *Is there a polynomial time algorithm for minimum vertex cover, or is it NP-hard?*
2. *Is there a polynomial time algorithm for maximum matching, or is it NP-hard?*
3. *Does the minimum vertex cover and the maximum matching have the same size?*

With regards to the third question, recall that in the edge disjoint paths, we found the maximum number of edge disjoint (s,t)-paths is exactly equal to the size of the minimum (s,t)-cut. One might hope for a similar equality between matchings and vertex covers. But consider a triangle graph (drawn below). A vertex cover must take 2 corners / vertices. A matching can only take one side / edge.



The remaining questions are algorithmic questions about matchings and vertex cover. Unlike disjoint paths and cuts, since maximum matching and minimum vertex cover can have different size means, it is possible for one to be polynomial time and the other to be NP-Hard.

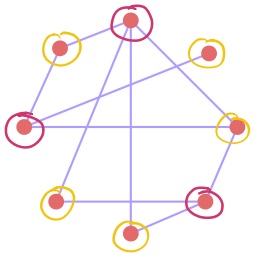
21.2 Vertex cover

We start with vertex cover.

Theorem 21.2. *Minimum vertex cover is NP-Hard.*

Proof. We show that minimum vertex cover solves maximum independent set, which we have already shown to be NP-Hard.

Consider an instance of the independent set problem defined by a graph $G = (V, E)$ with n vertices. For a set $S \subset V$, we denote its complement by $\bar{S} \stackrel{\text{def}}{=} V \setminus S$. We claim that S is an independent set iff $\bar{S}$ is a vertex cover. If so, then finding the minimum vertex cover is equivalent to finding the maximum independent set. In the example below, an independent set is circled in red, and a vertex cover is circled in yellow. Note that they are complementary vertex sets.



We prove the claim one direction at a time. First, let S be an independent set, and let e be any edge. Since no two vertices in S are connected by an edge, e can have at most one endpoint in S. That means e has at least one endpoint in $\bar{S}$. As this holds for all $e \in E$, $\bar{S}$ is a vertex cover.

Conversely, suppose $S \subset V$ is a vertex cover. Consider any two vertices $u, v \in \bar{S}$. If u and v were endpoints of an edge e, then S would not cover e. Thus $\bar{S}$ is an independent set. □

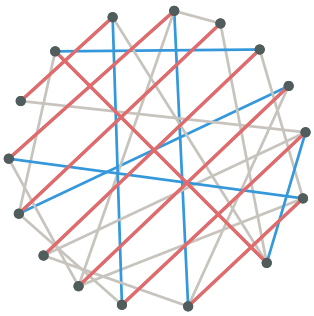
21.3 General matching

We now turn to the packing side of our primal-dual setup, matchings.

21.3.1 Alternating paths and cycles

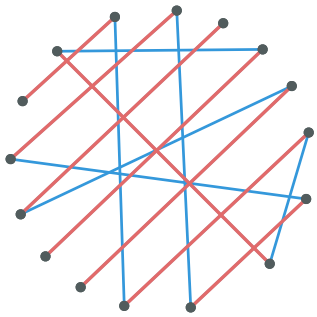
Let $G = (V, E)$ be an undirected graph. Let $B, A \subseteq E$ be two matchings where A is *maximum* and B is *maximal*. A simple example would be as follows. Here we have f5 edges. The blue (even) edges form a maximal matching B, and the red (odd) edges for a maximum matching A.

Below is a more complicated example. Again, the maximal matching B is in blue and the maximum matching A is in red.

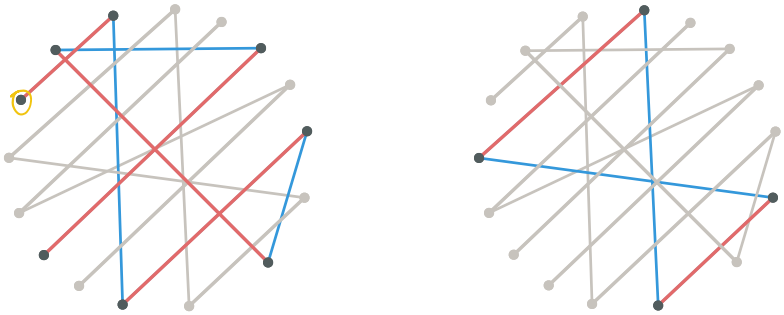


By definition of maximum, we have $|B| \leq |A|$. The first example shows that it is certainly possible for the inequality to be strict: $|B| < |A|$. (This is in contrast to forests.) Is there a converse inequality? For example, maybe there is a constant $c > 0$ such that for all maximal matchings B and all maximum matchings A, $|B| \geq c|A|$. Or perhaps the relation depends on the number of vertices, n. Is there a function $f(n)$ such that $|B| \geq f(n)|A|$? For example, can we show that $|B| \geq |A|/\log(n)$? Or $|B| \geq |A|/\sqrt{n}$?

To answer this question, let B be a maximal matching and let A be a maximum matching. We know that $|B| \leq |A|$ automatically, and we want to find a lower bound on the ratio $|B|/|A|$. We can assume B and A are disjoint. Consider the graph induced by just B and A (deleting all other edges from the graph).



This graph has maximum degree 2, since each vertex is incident to at most one edge from B and one edge from A. The connected components of a graph with maximum degree 2 is a vertex-disjoint collection of paths and cycles. Below on the left we highlight a path in $B \cup A$ and on the right we highlight a cycle.



Observe that the edges alternate between B and A; for this reason they are called *alternating paths* and *alternating cycles*. Clearly, in any cycle C, the number of edges from B equals the number of edges in A. Meanwhile, consider any component P of $B \cup A$ that is a path. Since P alternates between B and A, we have

$$|B \cap P| \geq |A \cap P| - 1.$$

If $|A \cap P| \geq 2$, then this implies that $|B \cap P| \geq |A \cap P|/2$. If $A \cap P$ is a single edge e, then by maximality of B, B must have at least one edge incident to e, size $|B \cap P| \geq 1$. Thus in either case, we have

$$|B \cap P| \geq |A \cap P|/2.$$

Let us review and conclude the argument. Given a maximal matching B and a maximum matching A, the connected components of $B \cup A$ are of three types.

1. Edges common to both B and A.

2. Cycles alternating between B and A.

3. Paths alternating between B and A.

In the first two cases, B has the same number of edges as A. In the third case, we have argued that B has at least half as many edges as A. Thus we have shown the following.

Theorem 21.3. *Let B be a maximal matching and let A be a maximum matching. Then $|B| \geq |A|/2$.*

21.3.2 Augmenting paths

To extend our line of reasoning further, we introduce the notion of free vertices and augmenting paths. Given a matching M, a free vertex is a vertex that is not incident to any edge in M. An augmenting path is an alternating path P from one free vertex to another. For any augmenting path P, the edge set $M \triangle P$ is a matching with one more edge than M.

Clearly, if B is a maximum matching, then there are no augmenting paths. Conversely, let B be a matching for which there are no augmenting paths, and let A be a maximum matching as before. Note that B is necessarily maximal, since an edge between two free vertices is a (very short) augmenting path. Consider any path P that forms a connected component of $B \cup A$ (that is not just an edge in $B \cap A$). If $|A \cap P| > |B \cap P|$, then since P alternates between B and A, we must have that P is an odd-length path starting and ending with A-edges. More to the point, P is an augmenting path for B.

This establishes the following optimality condition for matchings.

Lemma 21.4. *A matching M is a maximum matching iff there are no augmenting paths.*

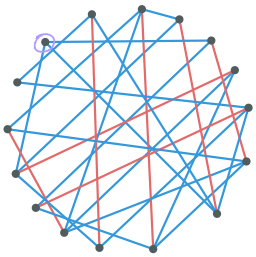
21.3.3 Augmenting path algorithms

We now turn to matchings in general graphs. Lemma 21.4 suggests the following augmenting path algorithm similar to previous algorithms.

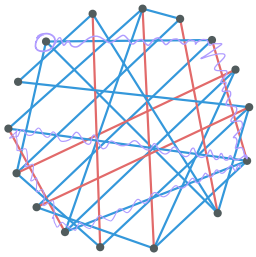
`generic-augmenting-paths`$(G = (V, E))$

1. $M \leftarrow \emptyset$
2. repeatedly (until M is maximum)
   A. Look for an augmenting path P with respect to M.
   B. Augment M along P.
   C. return M.

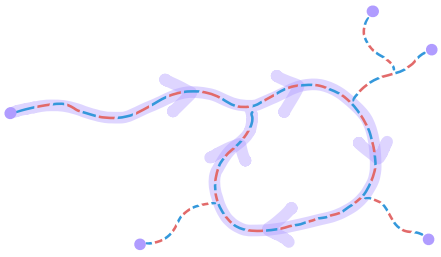
Certainly this algorithm would work – if it were implementable. Of the steps above, (2.A) is actually trickier then it looks.t Take the following graph, where a matching M is highlighted in red, and a free vertex is circled in purple.



If we follow alternating edges from the free vertex, then if we might trace an alternating path that runs into the following situation.



Above, the search has stumbled into a triangle of the form with one unmatched, one matched edge and one unmatched edge, in sequence. In general, the augmenting path approach can get stuck at odd alternating cycles.



This combinatorial obstacle is modeled by the following definitions. A *flower* is defined as an alternating walk where $(v_1, \ldots, v_k)$ where the first vertex is free and $v_1, \ldots, v_{k-1}$ forms a path, and, for some index $i < k$, $v_i, \ldots, v_k$ forms an odd cycle. The odd cycle $(v_i, v_{i+1}, \ldots, v_k)$ is called a *blossom*. The vertex $v_i = v_k$ is called the *base* of a blossom. The initial part of the flower $(v_1, \ldots, v_i)$ that forms a path to the base of the blossom is called the *stem*. Note that the stem may be an empty path when the base of the blossom is free.

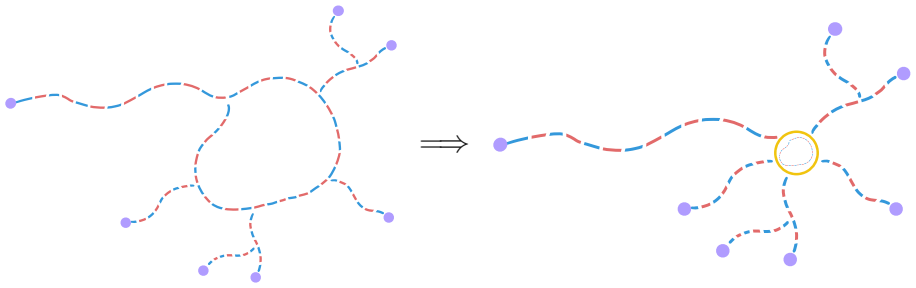
Lemma 21.5. *Let $(v_1, \ldots, v_k)$ be a flower with and let $v_k = v_i$ be the base of the blossom. Then the two edges in the blossom incident to v_i, (v_i, v_{i+1}) and (v_{k-1}, v_k), are unmatched edge.*

Proof. Because the cycle $(v_i, v_{i+1}, \ldots, v_k = v_i)$ has an odd number of edges, (v_i, v_{i+1}) and $(v_{k-1}, v_k = v_i)$ are either both matched or both unmatched edges. But v_i can only be incident to one matched edge, so they must both be unmatched. □

The presence of flowers and blossoms completely thwart the augmenting path approach. We cannot simply discard blossoms as they arise - parts of the cycle may be helpful in reaching other vertices. We also cannot continue on after finding a blossom, since it will add two matched edges incident to the base.

Many years ago, Jack Edmonds [Edm65] was stuck at this same point. And here he made the following elegant observation: you can *shrink* the blossom! This means we take the entire odd cycle B and contract it to a single vertex. (This can possibly create duplicate edges and self-loops which are ignored.)

In the picture below, we contract the odd cycle to a single vertex in yellow.



This creates a contracted graph in which the remaining edges of M still form a matching.

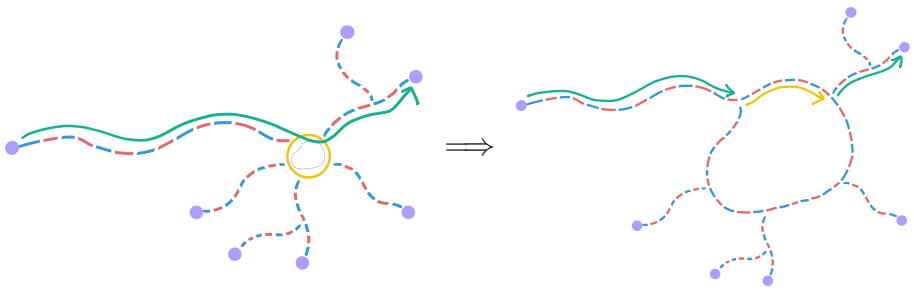
Let us now study the contraction step more formally. The first lemma shows that an augmenting path in the contracted graph can be expanded out to an augmenting path in the original graph. We point out that while the proof is somewhat careful for the sake of completeness, there are some pictures provided below the proof that make the claim very intuitive.

Lemma 21.6. *Let M be a matching, and let $B \subseteq E$ be a blossom w/r/t M. Let $G' = G/B$ be the graph obtained by contracting the blossom B, and let $M' = M \setminus B$ be the remaining matching in G'. Let P' be an augmenting path for M' in G. Then there is an augmenting path P in M such that $P' = P/M$.*

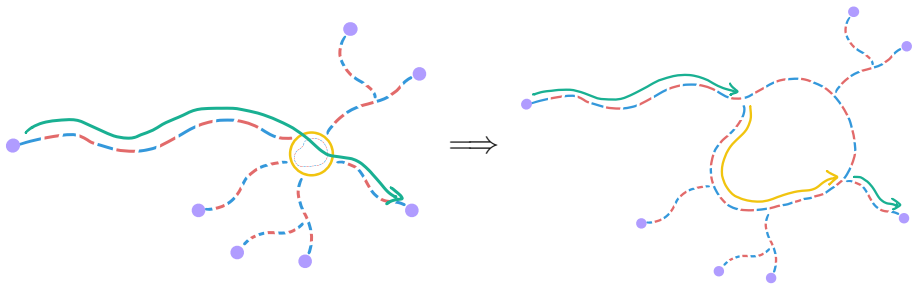
Proof. Let b' denote the vertex corresponding to the blossom B in the contracted graph. If P' does not include b', then it is already a path in G, and there is nothing to prove.

Otherwise, b' appears somewhere in the middle of the path P'. In the original graph G, this corresponds to two alternating paths. The part of P' before b' becomes an alternating path that ends at the stem s of the blossom B via a matched edge. The part of P' after b' corresponds to an alternating path that leaves at some other vertex t of B along an unmatched edge. Now, because B is an odd cycle, one of the two ways from the stem s to the departure point t around the cycle has an even number of edges. The even route is an alternating path starting at an unmatched edge and ending at a matched edge. We concatenate the first part of P' to s, the even length path in B to t, and the remaining part of P' from t to obtain an augmenting path in G. □

It is helpful to see some examples of how an augmenting path in the contracted graph is converted to an augmenting path in the original graph. In one case, suppose the augmenting path P' leaves the blossom on an unmatched edge. In our example, this implies we should traverse the blossom in the clockwise direction to match the parity.



In the second case, suppose the augmenting path leaves the blossom on an unmatched edge. To make the parities match, we traverse the blossom in the opposite direction.



Lemma 21.7. *Let M be a matching, and let $B \subseteq E$ be a blossom w/r/t M. Let $G' = G/B$ and $M' = M/B$ be the graph and matching obtained by contracting B, respectively. If M is not a maximum matching, M' is not a maximum matching in G'.*

Proof. If M is not maximum, then it has an augmenting path P.

1. *P does not touch B.*

Then it is already an augmenting path for M' in G', hence M' is not maximum.

2. *P touches B.*

Then P either enters B (for the first time) or leaves (for the last time) along an edge that is not matched, since there is at most one matched edge with one endpoint in B. Reversing P if necessary, we may assume that P enters B along an unmatched edge.

By definition, B is part of a flower, which also includes an alternating path from the base of B to a free vertex (which may be empty, if the base of the blossom is free). Consider the alternating path P' in G' where we follow P to the point where it enters B along an unmatched edge, and then follow the stem out to a free vertex. This gives an augmenting path for M', so M' is not maximum. □

We have now formally justified blossom contractions in an augmenting path framework. However it is not still not clear that we can easily find blossoms either! Fortunately the following lemma shows that running BFS along alternating paths will always find either an augmenting path or a blossom.

Lemma 21.8. *Let $W = (v_1, \dots, v_\ell)$ be a shortest alternating walk such that v_1 and v_ℓ are both free. Then either (a) W is an augmenting path or (b) $(v_1, \dots, v_k)$ is a flower for some $k < \ell$.*

Proof. Suppose W is not a path. Let $v_i = v_k$ for two indices $i < k$, with k as small as possible. In particular, $(v_1, \dots, v_{k-1})$ is a path, and $(v_i, \dots, v_k)$ is a cycle. If $(v_i, \dots, v_k)$ is an even length cycle, then we could remove it from W and obtain a shorter walk, a contradiction. So $(v_i, \dots, v_k)$ is an odd length cycle. □

We put everything together to obtain the following algorithm of Edmonds [Edm65].

`Edmonds-algorithm`$(G = (V, E))$

1. $M \leftarrow \emptyset$
2. Repeatedly search for an augmenting path in G w/r/t M
   A. If an augmenting path is found then uncontract all the blossoms and augment M along the augmenting path.
   B. Otherwise if a blossom B is found, then contract B.
   C. Otherwise uncontract all the blossoms and return M.

The overall running time is $O(mn^2)$, obtained as follows.

1. It takes $O(m)$ time to find either a blossom or an augmenting path.
2. We can find at most n blossoms before finding an augmenting path.
3. There are at most $n/2$ augmenting paths until we have a perfect matching.

In conclusion, we have established the following theorem.

Theorem 21.9 ([Edm65])**.** *The maximum cardinality matching can be computed in $O(mn^2)$ time.*

21.4 Additional notes and materials

We also recommend section 5.2 in [Sch], lecture notes by Michel Goemans[2], lecture notes by Chandra Chekuri[3], and slides by Bob Tarjan[4].

A 1991 interview of Jack Edmonds, in which he recounts his discovery of the blossom algorithm, can be found in [RSL+91] [5]. A 2015 issue of OPTIMA [6] celebrated the 50th anniversary of the blossom algorithm with recent related results and includes another retrospective interview with Edmonds.

[2] http://math.mit.edu/~goemans/18453S17/matching-nonbip-notes.pdf
[3] https://courses.engr.illinois.edu/cs598csc/sp2010/Lectures/Lecture7.pdf
[4] https://www.cs.princeton.edu/courses/archive/spr11/cos423/Lectures/NonbipartiteMatching.pdf
[5] https://www.ise.ncsu.edu/fuzzy-neural/wp-content/uploads/sites/9/2016/02/A-Glimpse-of-Heaven.pdf
[6] http://www.mathopt.org/Optima-Issues/optima97.pdf

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten .pdf prepared before the lecture (and .note file).
- Handwritten .pdf annotated during the presentation (and .note file).
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

21.5 Exercises

Exercise 21.1. Consider the following problem which is similar to vertex cover except the roles of vertices and edges are switched. Let $G = (V, E)$ be an undirected graph. The goal is to compute a minimum size set of edges $S \subseteq E$ such that every vertex is an endpoint of at least one edge in S. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 21.2. Let $G = (V, E)$ be an undirected graph with positive vertex weights $w : V \to \mathbb{R}_{>0}$. Recall that a *vertex cover* is a set $S \subseteq V$ such that every edge has at least one endpoint in S. Computing a minimum-weight vertex cover is NP-hard. Our goal is to produce a 2-approximation in polynomial time.

Consider the following *linear program* (LP):

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w(v)\, x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \text{ for each edge } \{u, v\} \in E, \\ & 0 \leq x_v \leq 1 \text{ for each vertex } v \in V. \end{aligned}$$

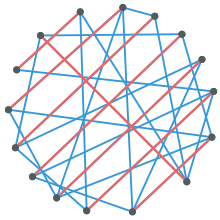


Figure 21.1: A maximum matching for the graph from Page 362.

This LP relaxes the integrality of vertex covers: any vertex cover $C \subseteq V$ corresponds to the LP-feasible point $x_v = 1$ for $v \in C$, and $x_v = 0$ otherwise. However, an optimal LP solution may take fractional values. You may use as a black box that an optimal LP solution can be computed in polynomial time.

Let x^* denote an optimal solution to the LP, and define

$$S = \{v \in V : x^*_v \geq 1/2\}.$$

1. (4 points) Prove that S is a vertex cover.
2. (6 points) Prove that the total weight of S is at most twice the weight of any vertex cover. (I.e., S is a 2-approximation to the minimum-weight vertex cover.)

Chapter 22

Lazy data structures

22.1 As easy as 1, 2, 3

We have all coded the line

```
i++;
```

meaning, take an integer i, and increase it by 1. It's the simplest operation in the world, and understood to be very fast. Now consider the underlying bits as we repeatedly increment i, starting from 1.

`1`, `10`, `11`, `100`, `101`...

No big deal. But subtly, in the short list above, one of the increments takes a little bit longer than the others. To go from `11` (3) to `100` (4), we actually flipped *two bits*, rather than one. Of course two bits is no big deal. If we take the number $2^{32} - 1$, and add one:

`11111111111111111111111111111111`, `10000000000000000000000000000000`, ...

we flip 32 bits![1] In general, a k bit number may flip k bits to increase by 1. How odd: the worst case performance of an increment is not uniform, and can be larger for large numbers. Yet we all have coded `i++` many times before and never really worried about this. Should we be worried?

When we count from 1 to n, how many total bits are flipped? Clearly the first (least significant) bit flips every time. The second bit flips every other time. The third bit flips every four times, and the fourth bit flips every 8 times. In general, the kth bit flips every 2^{k-1} times. So the number of bits flips is

$$\sum_{k=1}^{\infty} \left\lfloor \frac{2n}{2^k} \right\rfloor \leq n \sum_{k=1}^{\lceil \log n \rceil} \frac{1}{2^{k-1}} < 2n.$$

[1] Of course, for modern computers, 32 bits is no big deal either. But that's besides the point.

It is *as if* we flipped at most two bits with every increment. *In the aggregate*, an increment acts like a constant time operation. *On average*, each increment takes $O(1)$ time. Realistically, unless we are timing each increment individually (and usually we aren't), we don't notice the difference from a uniformly (worst-case) $O(1)$ running time and an on-average $O(1)$ running time. The technical term to express this $O(1)$ "average time" is to say that incrementing takes $O(1)$ *amortized time*.

22.2 Amortized analysis

In general, an operation takes "T amortized time" if any sequence of n invocations to the operation takes nT total time. To generalize this to multiple operations, suppose we have some data structure that has k different operations $\mathsf{op}_1, \ldots, \mathsf{op}_k$. Let $T_1, \ldots, T_k$ be k different values. We say that each operation op_i takes T_i amortized time if the total running time of any sequence of operations $\mathsf{op}_{i_1}, \mathsf{op}_{i_2}, \ldots, \mathsf{op}_{i_n}$ is

$$O(T_{i_1} + T_{i_2} + \cdots + T_{i_n}).$$

The example of a binary counter above has a single operation, increment, which was shown to take $O(1)$ amortized time.

The algorithm here is the most natural one. Just increment and flip bits as needed. Relaxing from worst-case to amortized bounds often permits such simplicity. The sophistication shifts from the algorithm to the analysis.

Amortized analysis is still worst-case analysis. The total running time bounds hold *in the worst case over all inputs*. The only relaxation is the change in measure from individual operations to aggregate performance.

A direct counting argument suffices to establish the amortized running time of a binary counter. There are also alternative bookkeeping frameworks for amortized analysis that can be more intuitive. We present two, called "credit schemes" (or the "banker's method") and "potential functions".

Credit schemes. The idea behind credit schemes is to have the cheap operations pay additional "credit" that can be exchanged for work later. Each unit of credit is worth $O(1)$ work. The rule is that we can only spend credit that was saved up in previous iterations.

Let us illustrate with the binary counter. Whenever we call `i++`, imagine placing one unit of credit on the first bit, half a unit of credit on the second bit, one fourth a unit of credit on the third bit, and so forth. This spreads 2 units of credit total.

In this credit scheme, the ith bit gets $1/2^{i-1}$ units of credit in each increment. Recall also that we only flip the ith bit every 2^{i-1} increments. Consequently every

time we have to flip the ith bit, it has already accumulated one unit of credit to "pay" for the flip. For each increment, then, we have the following:

$$(\text{time for increment}) + (\text{net increase in credit}) \leq O(1).$$

Over n increments, we have

$$\sum_{k=1}^{n} \left(\begin{pmatrix} \text{time for } k\text{th} \\ \text{increment} \end{pmatrix} + \begin{pmatrix} \text{net increase in credit} \\ \text{for } k\text{th increment} \end{pmatrix} \right) \leq O(n).$$

We also have

$$\sum_{k=1}^{n} \left(\begin{pmatrix} \text{time for } k\text{th} \\ \text{increment} \end{pmatrix} + \begin{pmatrix} \text{net increase in credit} \\ \text{for } k\text{th increment} \end{pmatrix} \right) = (\text{total time}) + \begin{pmatrix} \text{total credit} \\ \text{at the end} \end{pmatrix}.$$

Combining these two equations, rearranging, and observing that the amount of credit is always nonnegative, we have

$$(\text{total time}) \leq O(n) - \begin{pmatrix} \text{total credit} \\ \text{at the end} \end{pmatrix} \leq O(n).$$

So again we conclude that increment takes $O(1)$ amortized time.

The analysis above leveraged the following facts that are necessary for any legal "credit system".

1. Initially, there was 0 total credits.
2. The total amount of credits is always nonnegative.

Lemma 22.1. *Let operation $\mathsf{op}_1, \ldots, \mathsf{op}_k$ be k operations. Suppose we have a credit scheme (obeying the requirements above) such that for each operation op_i, we have that*

$$(\textit{time for } \mathsf{op}_i) + (\textit{change in credit}) \leq T_i$$

for values $T_1, \ldots, T_k \in \mathbb{N}$. Then each op_i takes T_i amortized time.

Proof. The general analysis here is very similar to the one specific to binary counters above. Consider a sequence of m operations $\mathsf{op}_{i_1}, \ldots, \mathsf{op}_{i_m}$. On one hand, we have

$$\sum_{j=1}^{m} \left(\text{time for } \mathsf{op}_{i_j} \right) + \begin{pmatrix} \text{change in credit} \\ \text{for } j\text{th operation} \end{pmatrix} \leq \sum_{j=1}^{m} T_{i_j}$$

by assumption. On the other hand we have

$$\sum_{j=1}^{m}\Big(\text{time for } \mathsf{op}_{i_j}\Big) + \begin{pmatrix}\text{change in credit}\\ \text{for } j\text{th operation}\end{pmatrix} = (\text{total time}) + \begin{pmatrix}\text{total credit}\\ \text{at the end}\end{pmatrix}.$$

Combining these two equations, rearranging, and observing that the amount of credit is always nonnegative, we have

$$\begin{aligned}(\text{total time}) &= \left(\sum_{j=1}^{m}\Big(\text{time for } \mathsf{op}_{i_j}\Big) + \begin{pmatrix}\text{change in credit}\\ \text{for } j\text{th operation}\end{pmatrix}\right) - \begin{pmatrix}\text{total credit}\\ \text{at the end}\end{pmatrix}\\ &\leq O\left(\sum_{j=1}^{m} T_{i_j}\right) - \begin{pmatrix}\text{total credit}\\ \text{at the end}\end{pmatrix} \leq O\left(\sum_{j=1}^{m} T_{i_j}\right),\end{aligned}$$

as desired. □

22.2.1 Potential function arguments.

A ball at rest at the top of a hill is given the slightest nudge. It rolls down the hill, picking up speed. Where did the energy come from? The ball at the top of the hill had *potential energy*, which was converted in equal parts into *kinetic energy* in motion.

A potential function tracks stored work in a data structure the same way. Let Φ be a nonnegative function of the state of the data structure, initially set to 0. For each operation, we bound

$$(\text{running time}) + \Delta\Phi,$$

where $\Delta\Phi$ denotes the change in Φ. Cheap operations increase Φ, storing potential for later. Expensive operations decrease Φ, spending that stored potential to pay for their cost. The chart below tracks a potential Φ going up (in red) and down (in green).



We first demonstrate the use of this method on binary counters. We define a potential function Φ by

$$\Phi = (\#\text{ bits set to 1}).$$

Note that Φ is 0 initially, and always nonnegative.

Consider Φ when we call `i++`. Recall that the net effect of `i++` is to flip h bits from 1 to 0 (for some $h \in \mathbb{Z}_{\geq 0}$), and at most one bit from 0 to 1. Thus the operation takes $O(h)$ time, and the potential Φ decreases by $h - 1$. That is, $\Delta\Phi \leq 1 - h$. The decrease in $\Delta\Phi$ by h offsets the time spent for flipping the h bits. Together we have that for any increment,

$$(\text{running time}) + \Delta\Phi = O(1) \tag{22.1}$$

Now consider a sequence of n increments. We have

$$\begin{aligned}\sum_{k=1}^{n}\left(\begin{array}{c}\text{time for } k\text{th}\\ \text{increment}\end{array}\right) &\overset{\text{(a)}}{=} \sum_{k=1}^{n}\left(\left(\begin{array}{c}\text{time for } k\text{th}\\ \text{increment}\end{array}\right)+\left(\begin{array}{c}\Delta\Phi \text{ for } k\text{th}\\ \text{increment}\end{array}\right)\right) - (\Phi \text{ at the end})\\ &\overset{\text{(b)}}{=} O(n) - (\Phi \text{ at the end}) \overset{\text{(c)}}{\leq} O(n).\end{aligned}$$

Here, in (a), we observe that the sum of changes in Φ from beginning to end is simply the value of Φ at the end. (Note that $\Phi = 0$ initially). In (b), we apply (22.1) to each increment. (c) observes that Φ is always nonnegative.

The potential-based argument above is generalizes beyond binary counters. The only point specific to binary counters was (22.1). The following lemma gives sufficient conditions so that for other applications, obtaining the analog of (22.1) implies the corresponding amortized bounds.

Lemma 22.2. *Let operation $\mathsf{op}_1, \ldots, \mathsf{op}_k$ be k operations, Φ a nonnegative potential function initially set to 0, and $T_1, \ldots, T_k$ k values, such that each time we call op_i, we have*

$$(\textit{time for } \mathsf{op}_i) + \Delta\Phi \leq T_i. \tag{22.2}$$

Then each op_i takes T_i amortized time.

Proof. The proof follows essentially the same format as the analysis above for binary counter; the only difference is that this proof is generic. Consider any sequence of operations $\mathsf{op}_{i_1}, \ldots, \mathsf{op}_{i_m}$. Let $\Phi_0 = 0$ denote the initial potential, and for each $k \in \mathbb{N}$, let Φ_k denote the potential after the kth operation op_{i_k}. Let $\Delta_k\Phi = \Phi_k - \Phi_{k-1}$ denote the change in potential from the kth operation. We have

$$\begin{aligned}&(\text{time for } \mathsf{op}_{i_1}) + \cdots + (\text{time for } \mathsf{op}_{i_m})\\ &\overset{\text{(a)}}{=} ((\text{time for } \mathsf{op}_{i_1}) + \Delta_1\Phi) + \cdots + ((\text{time for } \mathsf{op}_{i_m}) + \Delta_m\Phi) - \Phi_m\\ &\overset{\text{(b)}}{=} T_{i_1} + \cdots + T_{i_m} - \Phi_m \overset{\text{(c)}}{\leq} T_{i_1} + \cdots + T_{i_m},\end{aligned}$$

as desired. Here (a) observes that $\sum_{i=1}^{m} \Delta_i\Phi = \sum_{i=1}^{m} \Phi_i - \Phi_{i-1} = \Phi_m - \Phi_0 = \Phi_m$. (b) applies (22.2) to each operation. (c) is because the potential is always nonnegative. □

Remark 22.3. The charging and potential schemes are implicitly similar to each other: the potential acts as credit that pays for heavy operations later on; conversely, the total credit defines a potential function. Perhaps this is not surprising since they are both implicitly achieving the same goal of proving amortized running times. Let us not take the question of their formal distinction too seriously: ultimately these are metaphors meant only to aid our thinking.

22.3 Array Lists

We all know that an array is a sequence of entries $A[1..n]$ where each $A[i]$ can be accessed in constant time. The entries $A[1..n]$ are literally arranged in consecutive slots in memory. To retrieve $A[i]$, we really only need to know the location of $A[1]$, and then add $i-1$ memory units[2] to compute the location of $A[i]$. We can then jump straight to the location of $A[i]$ in memory.

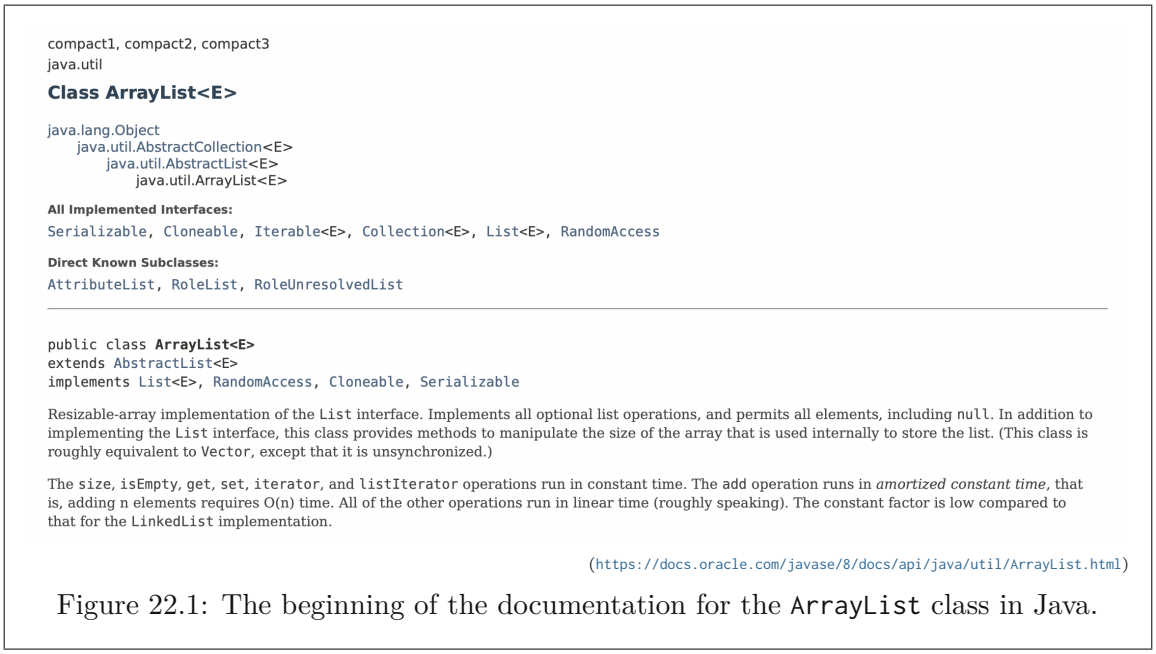


The downside to this convenience is that, after allocating $A[1..n]$, it is hard to adjust n. If we decide later to append an $(n+1)$th element, there is no clear place to put it. One can use pointers and start making a more involved data structure, but then we are giving up the direct memory access that makes arrays so appealing. Alternatively, we could make a new array $B[1..n+1]$ of size $n+1$, copy over the n elements from A, and store the $(n+1)$th item in the $(n+1)$th slot. But of course this takes $O(n)$ time to do the copying. And after that, what happens on the $(n+2)$th item? Do we copy everything over again?

A slight modification of this last strategy, sometimes called the "doubling trick", is assumed by the ubiquitous `ArrayList` data structure in `Java` (see Fig. 22.1) and by the built-in lists in `Python`. When we want to append elements beyond the capacity of $A[1..n]$, we allocate an array $B[1..2n]$ of *twice* the capacity, and copy the n elements of A over to the first n slots of B. See Fig. 22.2 for pseudocode.



[2]For lack of a better word.



compact1, compact2, compact3
java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:
`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess`

Direct Known Subclasses:
`AttributeList`, `RoleList`, `RoleUnresolvedList`

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

(https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)

Figure 22.1: The beginning of the documentation for the `ArrayList` class in Java.

Now, it seems like common sense to allocate generously so that we don't have to copy as often. But this design does not have $O(1)$ worst-case update time. To justify this approach, we turn to amortized analysis.

Theorem 22.4. *The array list supports constant time access (by index) to any element in the list, and can append an element to the list in $O(1)$ amortized time.*

Here are three different analyses.

Direct proof. Over k append operations, the total time spent by the data structure is equal to $O(k)$ plus the total time doubling and rebuilding the array. We rebuild every time the size of the list doubles, and rebuilding takes time proportional to the size of the list. This gives us a total running time of

$$\begin{aligned} &O(k) + \sum_{i=1}^{\lceil \log k \rceil} \Big(\text{time to double in size from } 2^i \text{ to } 2^{i+1}\Big) \\ &= O(k) + \sum_{i=1}^{\lceil \log k \rceil} O\Big(2^i\Big) = O(k). \end{aligned}$$

So each insertion takes $O(1)$ amortized time. □

append(x)

/* *We assume the ArrayList data structure is defined by a* ***capacity*** m*, a* ***size*** $n \leq m$*, and an array* $A[1..m]$ *of which the first* n *entries are occupied by the* n *elements of the list, in order.* */

1. If $m = n$:
   /* *Double the capacity.* */
   A. Allocate a new array A' of size $2m$.
   B. Cope the n elements of A into the first n entries of A'.
   C. Replace A with A' and n with $2n$.
2. Set $n = n + 1$ and $A[n] = 1$.

Figure 22.2: Appending an element to an array-backed list, doubling the capacity when needed.

Proof by charging. Whenever we append an element, we first pay an additional 1 unit of credit worth $O(1)$ additional work later.

Now, each append takes $O(1)$ time plus $O(n)$ time whenever we rebuild for a list of size n. Whenever we need to rebuild a list of size n, the list has doubled in size since its last rebuild. Consequently we have at least $n/2$ credits to pay for the $O(n)$ time required to rebuild the array. □

Proof by potential method. We define a potential

$$\Phi = \max\{n - \lfloor m/2 \rfloor, 0\},$$

where m is the capacity of the array, and n is the number of elements in the list. Note that Φ is always nonnegative, and equal to 0 when the array is empty.

Recall that the algorithm doubles the array if $m = n$, to ensure there is space, and then adds the $(n+1)$th element. Consider a rebuild. Since $m = n$, we have $\Phi \geq n/2$. After doubling the array and setting $m = 2n$, we have $\Phi = 0$. So $\Delta\Phi = -n/2$, and this decrease in potential offsets the $O(n)$ time needed to build the new array.

After that, adding the $(n+1)$th element to the array takes constant time and increases Φ by 1. All put together, whether or not we rebuild, we have

$$(\text{running time}) + O(1)\Delta\Phi = O(1).$$

We now invoke Lemma 22.2 and conclude that the array list supports $O(1)$ time per insertion. □

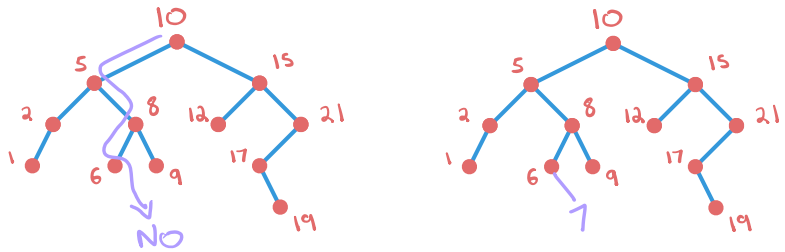


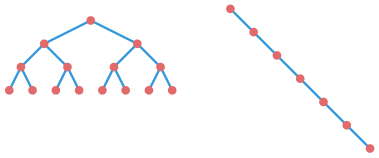
Figure 22.3: Searching for and inserting the key $k = 7$ in a binary search tree.

You might also want a `remove` operation that removes the last element from the list. (Together, `append` and `remove` define an array-based stack.) It's easy enough to clear the entry and decrease the size counter. But after many removals, the array's capacity may far exceed the number of stored elements. To keep the capacity close to the actual population, see exercise 22.1.

22.4 Lazy search trees

A binary search tree maintains a collection of ordered keys in a rooted tree. At each node, smaller keys go left, larger keys go right. Beyond search and insertion, BSTs support predecessor and successor queries, range aggregations, and ordered enumeration, all in time proportional to the height of the tree. See Fig. 22.3 for a sketch and Fig. 22.4 for pseudocode.

Tree operations are bounded by the height of the tree. Ideally, a tree would be *balanced*, where each subtree of a node containing roughly half the descendants. A balanced tree has height $O(\log n)$ for n keys, which is the minimum possible. An unbalanced tree can have ruinous height $\Omega(n)$.



itshape Balanced and unbalanced trees.

Building a balanced tree statically is easy. Sort the keys and keep dividing in half. Maintaining balance dynamically, as keys are inserted and deleted, is the challenge. Red-black trees and AVL trees carefully navigate rotations, marking, and case analysis to maintain balance at all times. Splay trees are another ingenious approach, analyzed in **??**.

/* *We assume each node x has pointers x.left and x.right to the roots of their left and right subtrees, respectively. Each node x represents a distinct key k stored at x.key, associated with a value stored at x.value. (The implementation here is* ***immutable****, making fresh copies of each node that is modified, and never writing over existing data.)* */

search(node x, key k)

/* *Return the value associated with key k in the binary search tree rooted by x. Return* null *if key k is not in the search tree.* */

1. If x is null then return null.
2. If $k = x$.key then return x.value.
3. If $k < x$.key then return search(x.left, k).
4. If $k > x$.key then return search(x.right, k).

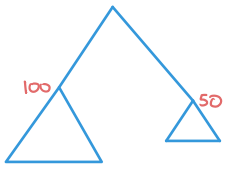
insert(node x, key k, value v)

/* *Insert value v at key k in the search tree rooted at x, return the root of the updated search tree. If key k is not present in the tree, then z has the same key as another node y in the search tree, then we replace y with x.* */

1. If x is null then return a new node y with y.left = null, y.right = null, y.key = k, and y.value = v.
2. If $k < x$.key then return a new node y copying x except with y.left = insert(x.left, k, v).
3. If $k > x$.key then return a new node y copying x except with y.right = insert(x.right, k, v).
4. If $k = x$.key then return a new node copying x except with x.value = v.

Figure 22.4: Implementation of a binary search tree mapping keys to values.

An embarrassingly lazy approach starts from a standard (unbalanced) binary search tree. For each node, we track the sizes of its two subtrees. Whenever one subtree has at least twice as many nodes as the other, we rebuild the entire subtree as a balanced tree. We call this strategy *lazy rebuilding*.



We extend the code in Fig. 22.4 to implement lazy rebuilding as follows. We augment each node with a counter that tracks the number of nodes in the subtree rooted at that node. We update the insert subroutine to update these counters as nodes are inserted. With these counters, each node can easily compare the difference in size of its two subtrees. Whenever we notice that, for some node x, one subtree of x has at least half as many nodes as the other subtree, we rebuild the entire subtree

rooted at x as a balanced search tree. Here we note that one can easily balance a tree with n elements in $O(n)$ time.[3]

In the following, we let n denote the number of elements in the binary search tree.

Theorem 22.5. *With lazy building, one can maintain a binary search tree of height $O(\log n)$ in $O(\log n)$ amortized time per insertion.*

Proof. Lazy rebuilding ensures that at any point in time, the number of nodes in one subtree is at most $2/3$ the number of nodes in the parent subtree. Therefore the tree always has height at most $\log_{3/2}(n) = O(\log n)$.

For each node v in the tree, let T_v denote the subtree rooted at v, and let $|T_v|$ denote the number of nodes in the tree.

We will define a potential Φ_v for each node v, and take the total potential as the sum of all these potentials. Let v be a node with children x and y. We define Φ_v as roughly absolute differences in size between the two subtrees T_x and T_y:

$$\Phi_v = \max\{0, |T_x| - |T_y| - 1, |T_y| - |T_x| - 1\}.$$

Above, the -1 term is for the case when $|T_x| + |T_y|$ is odd. If $||T_x| - |T_y|| = 1$, then this is as balanced as possible, and the potential is 0. Put another way, the -1 term ensures that $\Phi_v = 0$ iff v is a median over T_v.

We now define a global potential Φ as the sum of all the individual potentials,

$$\Phi = \sum_v \Phi_v.$$

Initially, when the tree is empty, $\Phi = 0$. More generally, we have $\Phi = 0$ iff the BST is perfectly balanced in the sense that each node represents the median of its subtree.

Claim 1. The part of insertion that comes before rebuilding $O(\log n)$ amortized time.

Because the tree has height $O(\log n)$, it takes $O(\log n)$ time to find the right leaf to place the key. The insertion increases the potential Φ_v by at most 1 for each of the $O(\log n)$ nodes v on the root-to-leaf path.

Claim 2. Rebuilding at an imbalanced node v takes $O(1)$ amortized time.

[3]For example, one can first traverse the subtree in order linear time, writing down the data in order in an array. With this array, one can split the array into subarrays by the median, and recursively continue to split the subarrays by their medians. Each median represents the root of a subtree; each subarray represents a subtree. Choosing the median ensures it is balanced.

Suppose v has k nodes in its subtree T_v. Then v's potential, Φ_v, must have been at least $k/3 - 1$. Optimally rebalancing the subtree T_v takes $O(k)$ time and decreases the potential Φ_x to 0 for every node in T_v, included v. The decrease in Φ_v alone pays for the time spent rebuilding.

Combining the two claims above gives $O(\log n)$ amortized time overall. □

22.4.1 Lazy deletions

What about deletions? Removing a node from a standard BST is messy because (unless it is a leaf) it can leave a "hole" in the tree. One could fill this hole with the greatest predecessor or least successor in the subtree, recursively filling the hole left behind. It can get messy.

Instead of restructuring the tree, suppose we just mark the node as deleted and adjust the search to skip marked nodes. The only catch is that after many deletions, most nodes may be marked for deletion, which is wasteful. Let m denote the total number of nodes, marked for deletion or not, in the tree, and let n denote the number of unmarked (undeleted) keys in the tree. Our tree will have height $O(\log m)$, rather than $O(\log n)$, and our space usage will be $O(m)$, rather than $O(n)$.

We want to keep the tree size m within a constant factor of the "true size" n in a lazy fashion. We maintain a counter for the number of nodes marked for deletion; whenever it exceeds $m/2$, we clean up everything by deleting all the marked nodes and rebuilding the entire tree in $O(m) = O(n)$ time. These operations maintain the following invariant.

Lemma 22.6. *At any point in time, less than half the nodes in the tree are marked for deletion.*

Consequently we maintain optimal height:

Lemma 22.7. *At any point in time, the tree has height $O(\log n)$, where n refers to the number of keys that have not been deleted.*

Proof. By the same argument as above, the tree has height logarithmic in the total number of nodes (marked or unmarked). But the total number of nodes is within a constant factor of the total number of unmarked vertices. □

Theorem 22.8. *Lazy rebalancing and lazy deletions maintains a binary search tree with height $O(\log n)$ in $O(\log n)$ amortized time per insertion or deletion, where n is the number of keys in the tree.*

Proof. Let m denote to the total number of nodes in the tree, marked or unmarked. Let n denote the number of unmarked nodes; i.e., the number of keys represented by the search tree.

The lazy deletion keeps $n \geq m/2$ at all times; consequently $O(\log n) = O(\log m)$ and we do not distinguish the two quantities below.

Compared to the insertion-only analysis above, we continue to have the same per-vertex potential functions Φ_v, but here we should clarify that $|T_v|$ denotes the number of nodes (marked or unmarked) in the subtree rooted at v. We again let $\Phi = \sum_v \Phi_v$.

We also introduce a new potential function Ψ that counts the total number of vertices marked for deletion. The total potential is now $\Phi + \Psi$.

Each insertion takes $O(\log n)$ amortized time by the same argument as before; to recap, the part where we add a leaf to the BST increases Φ by $O(\log n)$, while the time spent rebuilding a subtree is paid for by the accompanying decrease in Φ.

Each deletion first takes $O(\log n)$ time to find the key to mark for deletion (because the height is $O(\log n)$), and then increases Ψ by 1. When we rebuild on account of deletions, Ψ is at least $m/2$. The rebuild takes $O(m)$ time and decreases Ψ to 0. So the decrease in potential pays for the rebuild. We conclude that deletion takes $O(\log n)$ time as well. □

Remark 22.9. To support other miscellaneous tree operations, it is sometimes preferable to ensure that in every subtree, at most half the nodes are marked for deletion. To this end, we can keep a count of the number of nodes marked for deletion for every subtree, and rebuild the entire subtree whenever this count reaches half the total number of nodes in the subtree. This will still have $O(\log n)$ amortized time by a similar analysis as above.

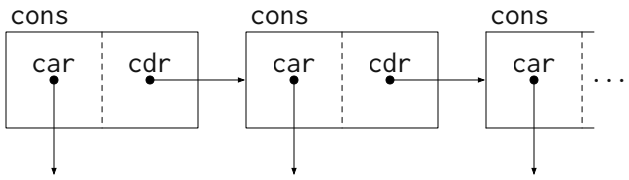
22.5 Immutable data structures

In purely functional programming languages such as Haskell[4], all data is required to be *immutable*. Immutability is a fancy name for one simple rule: once something is written down, it can never be changed.

This might seem restrictive. But now when you hand off data to a function, you are promised that the function won't mess with your data. This guarantee is life-saving in concurrent programs where many threads share data. Compilers can also optimize more aggressively when they know nothing mutates, often winning back the apparent inefficiency.

[4] `http://learnyouahaskell.com/chapters` - have fun!

A (singly) linked list is inherently immutable. A linked list consists of `nodes`, each of which contains two pointers. One pointer is to the next node in the list, or `null` if it is the end of a list. The other pointer is to some piece of data that we would want to associate with the node. In `lisp`, a node is called a `cons` cell, the pointer to the data is called the `car`, and the pointer to the next `cons` is called the `cdr`.



To insert (`push` / `cons`) an element x to the beginning of a linked list, we create a new node that points to x and to the first node in the list. Henceforth the new node is treated at the beginning of the list. Observe that no changes were made in the existing list. In particular, any other pointers to the top of the list will not notice any changes.

Likewise we can immutably remove (`pop`) elements from the beginning of a linked list. Given a pointer to the beginning of a list, we just follow the pointer to the second element in the list, and use that as the top of the pointer to the list. Note that we didn't make any change to the data structure from an external point of view. We simply updated our own reference to the data structure, and the data structure is changed only from our own perspective.

Suppose you want to update one field of an object. Without mutation, you have to make a fresh copy of the entire object. This sounds expensive. But an object is typically just a few fields and pointers. Copy the fields, change what you need, and the pointers still point to the same shared data. The cost is one node instead of the whole structure.

The lazy search trees from Section 22.4 are a good example. Inserting a key changes nodes along a single root-to-leaf path. The immutable version copies just those nodes, reusing pointers to the subtrees hanging off the path. The tree has height $O(\log n)$ so the cost is $O(\log n)$ per insertion.

Copy-on-write also gives you snapshots and an undo stack for free.

22.5.1 Stacks and Queues

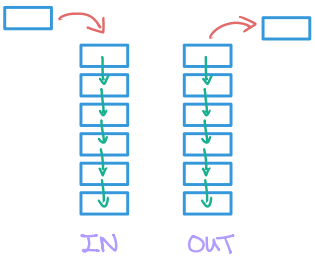


Recall that a *stack* is a data structure that allows us to insert and delete items strictly in *first-in-last-out* order. Namely, there are two operations, `push` and `pop`. `push` inserts an object. `pop` removes the most recently `push`'d

object that had not yet been popped. Imagine a stack of plates. The linked list discussed above gives an immutable implementation of a stack.

What about a *queue*? A *queue* inserts and removes objects in *first-in-first-out* order. A singly linked list does not suffice. We need to insert and delete from opposite ends. Can we achieve $O(1)$ amortized time per operation?

Two immutable stacks are enough.



1. Maintain two (immutable) stacks, called the input and output stacks.
2. Insert into the input stack.
3. Remove from the output stack.
4. If the output stack is empty, then replace the output stack with the input stack reversed, and set the input stack to be empty.

Theorem 22.10. *The two-stack queue implements insertion and deletion in $O(1)$ amortized time.*

Proof. We maintain a potential function Φ equal to the number of elements in the input stack. Each insertion takes $O(1)$ (real) time and then increases the potential by 1, so it takes $O(1)$ amortized time. Each deletion, excluding the part where we rebuild the output stack, takes $O(1)$ real time. Rebuilding the list takes $O(\Phi)$ time, where Φ is the number of elements in the input stack. Meanwhile the potential Φ decreases to 0. □

Amortized analysis makes many more data structures immutable for free. See [Oka99].

22.6 Additional notes and references

Amortized analysis has long been used implicitly to analyze data structures and algorithms, and the general concept was more formally introduced by Tarjan [Tar85]. See [Eri13a], [DDL15, Lecture 5][5], and [Blu11a, Chapter 7] for additional notes on amortized analysis. Slightly different lazy rebuilding strategies for BSTs are described in [Eri13b].

[5] https://www.youtube.com/watch?v=3MpzavN3Mco

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten .pdf prepared before the lecture (and .note file).
- Handwritten .pdf annotated during the presentation (and .note file).
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

22.7 Exercises

Additional exercises can be found in [Eri13a].

Exercise 22.1. Recall the array-backed lists from Section 22.3, where we showed how to support `append` in $O(1)$ amortized time via the "doubling trick". As discussed at the end of Section 22.3, one may also want to include a `remove` operation that removes the last item in the list. To implement this, one could just unallocate the corresponding spot in memory, and decrease the counter tracking the size of our list. However, after many such removals, we may end up with an array that is mostly empty and much larger then the number of items in the list. It would be desirable to maintain the array to be within a constant factor. Here we will develop an analog of the doubling trick to facilitate deletions.

1. A first approach to addressing deletions might be as follows.

   > *After removing the element from the array and decreasing the size counter, if the size of the list is now less than half of the capacity of the array, allocate a new array of half the capacity and copy the elements of the list over.*

Unfortunately, this approach will not run in $O(1)$ amortized time. To prove this, devise a sequence of n `append` / `remove` operations (for n arbitrarily larger) on which the data structure would take $\Omega(n^2)$ time.

2. While the approach above didn't quite work, a slight modification of it will. Implement `remove` so that `append` and `remove` both take $O(1)$ amortized time. (Analyze these operations as well.) Your implementation should still follow the general format of allocating a new array and copying all the elements at certain points in time.

Exercise 22.2. Recall that a linked-list based stack supports the `push(`x`)` and `pop()` operations in $O(1)$ worst-case time. Consider the following operation, called `multipop`, that takes as input $k \in \mathbb{N}$, and removes and returns the top k items on the stack.

`multipop(stack` S`,` k`)`

1. $L \leftarrow$ empty linked list
2. For $i = 1, \ldots, k$ as long as S is not empty:
   A. Let $x = S$`.pop()`, and attach x to the top of the list.
3. Reverse L and return it.

Prove that after incorporating `multipop`, all three operations `push`, `pop`, and `multipop` each take $O(1)$ amortized time.

Exercise 22.3. Section 22.5.1 showed how to use two (immutable) stacks to implement a queue with $O(1)$ amortized performance. Here we will develop a more sophisticated, *double-ended queue* using three immutable stacks. (It's mostly using two stacks, but occasionally needs a third stack as a buffer.)

A double ended queue is like a queue except we can insert and delete from both ends of the queue. Here we designate one end as the "front" and the other end as the "back", and define the following four operations.

1. `push-front(`x`)`: inserts element x at the front of the queue.
2. `push-back(`x`)`: inserts element x at the back of the queue.
3. `pop-front()`: removes the first element of the queue.
4. `pop-back()`: removes the last element of the queue.

(The data structure throws an error if the user calls either pop operation and there are no elements remaining.)

1. The natural extension to the two-stack approach is to flip all the elements over whenever you are popping from a side with an empty stack. Prove that this approach does *not* get $O(1)$ amortized time.

2. Design and analyze a different data structure, using at most 3 immutable stacks, that obtains $O(1)$ amortized time for each operation. (Our solution is somewhat similar to the two-stack approach, but the modification requires using a third stack on occasion as a buffer.)

Exercise 22.4. A *stacqueué* is a French list-like data structure supporting both the standard operations of a stack and a queue:

1. dequeue: removes an item from the beginning of the list.

2. push (a.k.a. enqueue): inserts an item at the end of the list.

3. pop: removes an item from the end of the list

Recall that a stack implements the push and pop operations in $O(1)$ worst case time. Design and analyze a stacqueué using only stacks that supports each operation in $O(1)$ amortized time.

Exercise 22.5. Implement `merge-sort` in $O(n \log n)$ time using only (immutable) stacks. Here the input consists of a stack of elements. The output is a stack of elements in sorted order (increasing from the top of the stack to the bottom.)

Chapter 23

Priority queues and disjoint union

23.1 Priority queues

The humble heap stores key-value pairs and supports two operations.

- `insert(`k`,`v`)`: insert a new key-value pair (k, v).
- `remove-min()`: remove and return the pair (k, v) with minimum value v.

The standard binary heap supports both operations in $O(\log n)$ time. Among other applications, heaps can sort n values in $O(n \log n)$ time.

We have seen two basic graph algorithms, for shortest paths and for MST, that use heaps. Both algorithms build out trees starting from a single vertex and use heaps to identify the next closest vertex to add. In addition to calling each of the heap operations above once for every vertex, they also call the following operation for every edge in the graph.

- `decrease(`k`,`v`)`: decrease the value of an existing key k to v.

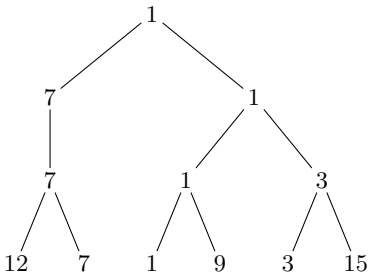
A *priority queue* is a heap that also supports `decrease`.

A binary heap supports `decrease` in $O(\log n)$ time. Surprisingly, Fredman and Tarjan [FT87] showed that `decrease` can be implemented in $O(1)$ amortized time, with a data structure called the *Fibonacci heap*.

Theorem 23.1 ([FT87])**.** *There exists a data structure supporting* `insert`*,* `decrease`*, and* `remove-min` *in* $O(1)$*,* $O(1)$*, and* $O(\log n)$ *amortized time, respectively.*

We follow a simpler construction from [Cha13].

Tournament trees. A tournament tree is a rooted tree where every node has one or two children, and all leaves are at the same depth. All elements are stored at the leaves. Each internal node stores a pointer to the minimum element in its subtree. The *top node* of an element is the highest node storing it.



Two tournament trees of the same height h can be combined to form a single tournament tree of height $h+1$, by making them children of a new root. This is called *linking*.

If x is a non-root top node, we can *cut* the subtree rooted at x to form its own tournament tree. The parent becomes unary, and the rest of the tree remains a tournament tree. (Why?)

Maintaining tournament trees. Our heap is a collection of tournament trees satisfying two invariants.

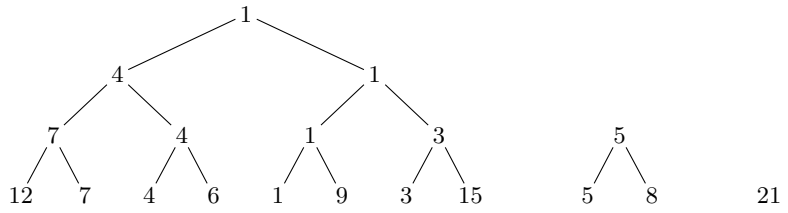
(a) All tree heights are distinct.

(b) Every tree has height $O(\log n)$.

To pay for this housekeeping, consider the potential

$$\Phi = C_t(\#\text{ trees}) + C_s(\#\text{ unary nodes}) + C_n(\#\text{ nodes}),$$

for suitable positive constants C_t, C_s, C_n.



If two trees have the same height h, we link them. This adds one new root and decreases the number of trees by 1. For C_t a constant factor larger than C_n, the drop in Φ pays for the linking.

Cuts create unary nodes. Too many of them near the top make the tree taller than it needs to be. Whenever at least half the nodes at some height h are unary, we delete every node at height h and above. This leaves a collection of trees of height

$h - 1$. The number of new trees is at most a constant factor larger than the number of deleted unary nodes at height h. For C_s a constant factor larger than C_t, the drop in unary nodes pays for the new trees. The drop in the number of nodes pays for the deletions.

The cleanup is free. Linking ensures that all tree heights are distinct. Rebuilding when there are too many unary nodes keeps the heights bounded by $O(\log n)$. There are only $O(\log n)$ trees.

Priority queue operations. `insert` creates a singleton tree. `decrease` cuts at the top node of the decreased element and updates its value. Both take constant time and increase the potential by at most a constant, so the amortized cost of each is $O(1)$.

`remove-min` scans the roots to find the minimum, say at key k, and deletes the root-to-leaf path of nodes containing k. This creates $O(\log n)$ new trees. The time and potential increase are both $O(\log n)$, so the amortized cost is $O(\log n)$.

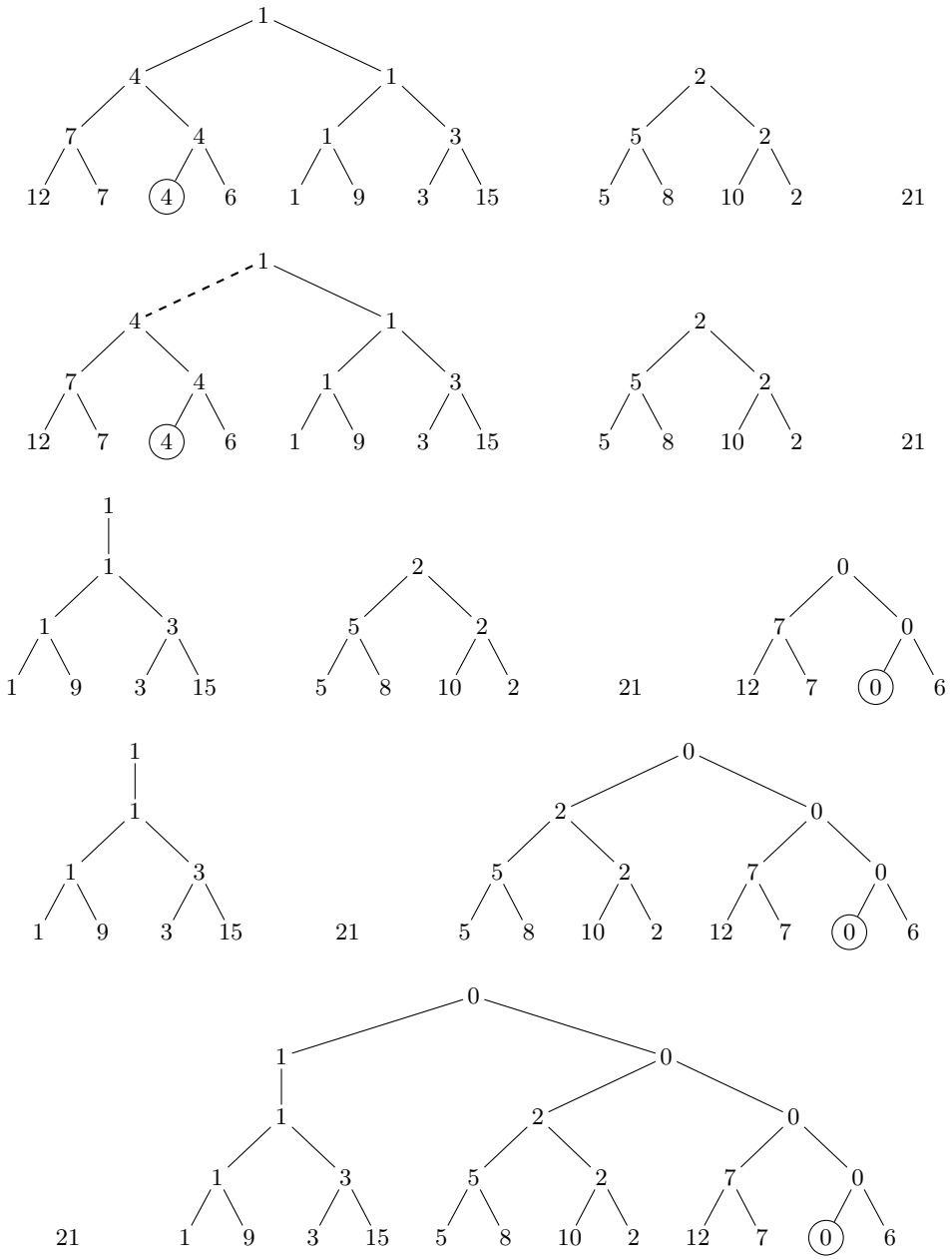


Figure 23.1: Decreasing element 4 to 0, and consolidating.

23.2 Disjoint union

The greedy algorithm processes the edges in increasing order of weight, taking any edge whose endpoints lie in distinct components. A *disjoint-union* data structure tracks these components as they merge.

A disjoint-union data structure over a set of n elements supports the following operations.

- `new-set(`x`)`: Create a new singleton set containing x.
- `union(`x`,`y`)`: Merge the set containing x with the set containing y.
- `same-set(`x`,`y`)`: Return `true` if x and y are in the same set; else return `false`.

Represent each set as a rooted tree. Each element stores a parent pointer. The root has no parent and serves as the *representative* of the set.

The basic operation is `find(`x`)`, which returns the representative of the set containing x.

`find(`x`)`

1. Follow parent pointers from x and return the root.

Then `same-set(`x`,`y`)` checks whether `find(`x`)` = `find(`y`)`.

To implement `union`, we need a rule for linking two roots. Each node stores a rank that upper-bounds the height of the subtree rooted there. Initially every node has rank 0. When linking two roots, the lower-rank root becomes a child of the higher-rank root. If the ranks are equal, either root can become the parent, and its rank increases by 1.

`union(`x`,`y`)`

1. Let $a \leftarrow$ `find(`x`)` and $b \leftarrow$ `find(`y`)`.
2. If a`.rank` $>$ b`.rank` then make b a child of a.
3. Else if b`.rank` $>$ a`.rank` then make a a child of b.
4. Otherwise make b a child of a and increment the rank of a.

The number of non-root nodes is bounded by the number of calls to `union`. (Why?) A node with rank k has at least 2^k nodes in its subtree. (Why?) So every rank is at most $\log n$. Since rank upper-bounds height, every tree has height $O(\log n)$. Accordingly, `find` takes $O(\log n)$ time. `new-set` takes constant time. `union` is just two calls to `find`.

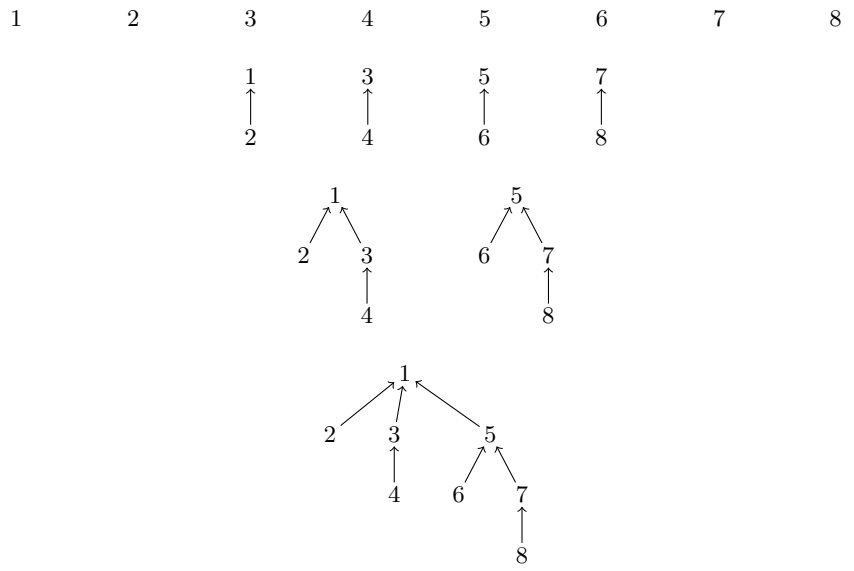
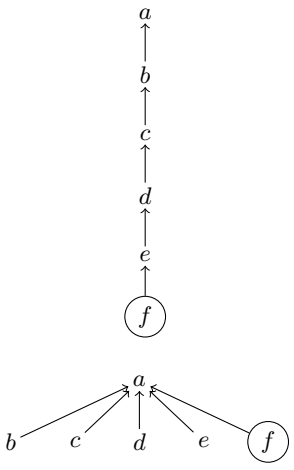


Figure 23.2: Building a tree by union by rank. Eight singletons are merged in three rounds.

Path compression. `find(`x`)` traverses the path from x to its root. Once the root is known, there is no reason to leave that path in place. Redirect x to the root, and while at it redirect the other nodes on the path to the root. Use the work of traversing the path to flatten it. This is called *path compression*. The next time we call `find` on any of these nodes, it returns in constant time.



`find(`x`)`

`/*` *Follow parent pointers from x to its root, compressing the x-to-root path along the way.* `*/`

1. If x`.parent` is `nil` then return x.
2. x`.parent` $\leftarrow$ `find(`x`.parent)`.
3. Return x`.parent`.

The cost of each `find` is proportional to the number of parent pointers followed. So the analysis is a count of pointer traversals.

We continue to use rank-based linking for `union`. Ranks never change, and they increase strictly from child to parent.

There are at most $n/2^i$ nodes of rank i. (Why?)

To $\log^* n$... All ranks lie in the interval $[0, \lfloor \log n \rfloor]$. Choose thresholds

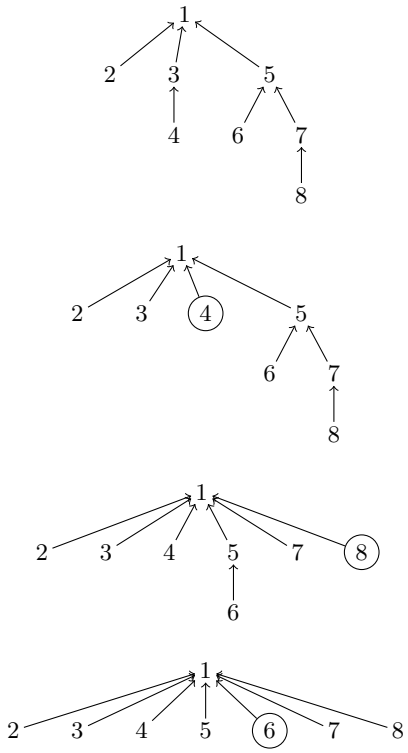
$$0 = t_0 < t_1 < \cdots < t_k = \lfloor \log n \rfloor.$$

These determine intervals

$$T_1 = [t_0, t_1),\ T_2 = [t_1, t_2),\ \ldots,\ T_{k-1} = [t_{k-2}, t_{k-1})$$
$$T_k = [t_{k-1}, t_k]$$

Each interval T_i has $\sum_{j \in T_i} n/2^j = O(n/2^{t_{i-1}})$ nodes.

Each node lies in a particular interval by its rank. Call a parent pointer "small" if it is contained in an interval, and "big" if it crosses intervals. There are at most k big pointers along any `find`, because each big pointer crosses to a higher interval.

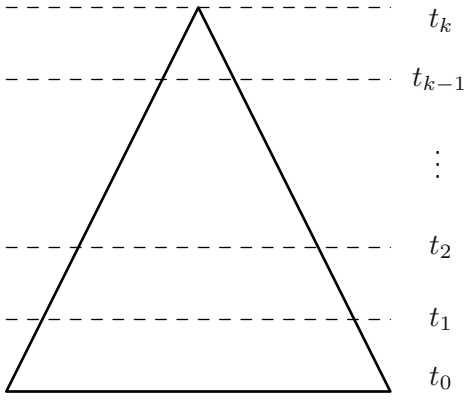


Flattening a set over three calls to `find`.

Path compression limits the total number of small pointers followed across all `find` operations. When `find` follows a small pointer from x to a non-root parent, it resets $\texttt{parent}(x)$ to the root, strictly increasing $\text{rank}(\texttt{parent}(x))$. Once $\text{rank}(\texttt{parent}(x)) \geq t_i$, the pointer from x becomes big. A node in T_i contributes at most t_i small non-root traversals in total. Since T_i has $O(n/2^{t_{i-1}})$ nodes, the total from T_i is $O(t_i n/2^{t_{i-1}})$. Each `find` also follows at most k big pointers and at most one pointer to the root. The total cost of all operations is

$$O\left(km + \sum_{i=1}^{k} \frac{t_i}{2^{t_{i-1}}} n\right).$$

Let $t_i = \min\{2^{t_{i-1}}, \lfloor \log n \rfloor\}$. Then $t_i/2^{t_{i-1}} \leq 1$ for each i, and the total is $O(k(m+n))$. k needs to be just large enough that $t_k = \lfloor \log n \rfloor$. Each threshold exponentiates the previous one. For $k = \log^* n$, $2^{t_{k-1}} \geq \log(n)$ and $t_k = \lfloor \log n \rfloor$. The total cost of all operations is $O((m+n)\log^* n)$.



And beyond! The $\log^* n$ proof used one fixed partition of the entire range of ranks. A sharper analysis gives each node its own thresholds, generated recursively from its rank.

Fix a non-root node x of rank $r \geq 2$. We measure $\text{rank}(\texttt{parent}(x))$ against a hierarchy of scales starting from r. The first few look like

$$\begin{array}{llll} r+1, & r+2, & r+3, & \dots, \quad 2r-1, \\ 2r, & 2^2 r, & 2^3 r, & \dots, \quad 2^{r-1} r, \\ 2^r r, & 2^{2^r r}(2^r r), & 2^{2^{2^r r}(2^r r)}(2^{2^r r}(2^r r)), & \dots, \quad 2^{2^{\cdot^{\cdot^{\cdot}}(\cdots)}}(\cdots), \end{array}$$

By the third scale the formulas are already unwieldy. The pattern is clearer recursively. Define functions $f_0, f_1, f_2, \dots$ of an input t by

$$f_0(t) = t + 1,$$
$$f_{k+1}(t) = f_k^{(t)}(t).$$

Scale i consists of the $r-1$ thresholds obtained by repeatedly applying f_i to r.

$$\begin{array}{lllll} \text{scale } 0: & f_0(r) = r+1, & f_0^{(2)}(r) = r+2, & \dots, & f_0^{(r-1)}(r) = 2r-1, \\ \text{scale } 1: & f_1(r) = f_0^{(r)}(r) = 2r, & f_1^{(2)}(r) = 2^2 r, & \dots, & f_1^{(r-1)}(r) = 2^{r-1} r, \\ \text{scale } 2: & f_2(r) = f_1^{(r)}(r) = 2^r r, & f_2^{(2)}(r), & \dots, & f_2^{(r-1)}(r), \\ & \vdots & \vdots & \ddots & \vdots \\ \text{scale } k: & f_k(r) = f_{k-1}^{(r)}(r), & f_k^{(2)}(r), & \dots, & f_k^{(r-1)}(r). \end{array}$$

Each function f_k is increasing.

These thresholds divide the possible values of $\text{rank}(\texttt{parent}(x))$ into intervals. For $\ell \geq 0$ and $1 \leq i \leq r-1$, let

$$I_{\ell,i} = [f_\ell^{(i)}(r), f_\ell^{(i+1)}(r)).$$

The intervals in each scale are increasing, and the last interval in one scale ends where the first interval in the next begins. So for a fixed node x, there are unique values $\ell(x) \geq 0$ and $1 \leq \iota(x) \leq r-1$ such that

$$\text{rank}(\texttt{parent}(x)) \in I_{\ell(x),\iota(x)}.$$

The *level* $\ell(x)$ identifies the scale, and the *index* $\iota(x)$ identifies the interval inside that scale.

Each time path compression gives x a new parent, the parent's rank increases. The level can only increase, and the index can only increase within a fixed level.

A potential function. Let $L \in \mathbb{N}$ be any upper bound on the level of every node of rank at least 2. For each non-root node x of rank at least 2, define

$$\phi(x) = (L - \ell(x)) \operatorname{rank}(x) - \iota(x).$$

Let Φ be the sum of $\phi(x)$ over all such nodes.

If x has rank r and level k, then

$$\operatorname{rank}(\texttt{parent}(x)) < f_{k+1}(r) = f_k^{(r)}(r),$$

so $\iota(x) \in \{1, 2, \ldots, r-1\}$. It follows that

$$0 \le \phi(x) < L \operatorname{rank}(x).$$

Whenever a union makes a root into a child, it creates at most $L \operatorname{rank}(x)$ units of new potential. Each node becomes a child at most once, so the total potential ever introduced is at most

$$L \sum_x \operatorname{rank}(x) \le L \sum_{i \ge 0} i \cdot \frac{n}{2^i} = O(nL).$$

`find`. Consider one call to `find`, tracing a path $x_0, x_1, \ldots, x_h$ to the root.

Any traversed node of rank 0 or 1 is charged directly to the operation. There can be at most one node of each rank on the path, so this costs only $O(1)$.

Now look at traversed nodes of rank at least 2. Call such a node x_i *big* if its level is larger than the level of every such node above it on the path. Call it *small* otherwise. Big nodes are charged directly to the operation. Their levels strictly decrease, so there are at most L of them.

Now let x_i be a small node. Then some node x_j lies above it on the path with $\ell(x_j) \ge \ell(x_i)$. We claim that path compression then decreases $\phi(x_i)$ by at least 1.

We have

$$\operatorname{rank}(x_h) \overset{\text{(a)}}{\ge} \operatorname{rank}(x_{j+1}) \overset{\text{(b)}}{\ge} f_{\ell(x_j)}(\operatorname{rank}(x_j)) \overset{\text{(c)}}{\ge} f_{\ell(x_i)}(\operatorname{rank}(x_j)).$$

Here (a) is by monotonicity of ranks, (b) is by definition of level, and (c) is because higher scales start later.

Also

$$\operatorname{rank}(x_j) \overset{\text{(d)}}{\ge} \operatorname{rank}(x_{i+1}) \overset{\text{(e)}}{\ge} f_{\ell(x_i)}^{(\iota(x_i))}(\operatorname{rank}(x_i)).$$

Here (d) is by monotonicity of ranks and (e) is by definition of index.

Combining these inequalities and using monotonicity of $f_{\ell(x_i)}$,

$$\operatorname{rank}(x_h) \geq f_{\ell(x_i)}\Big(f_{\ell(x_i)}^{(\iota(x_i))}(\operatorname{rank}(x_i))\Big) = f_{\ell(x_i)}^{(\iota(x_i)+1)}(\operatorname{rank}(x_i)).$$

After compression, the parent of x_i becomes x_h. So the new parent rank of x_i lies in a later interval than before. Thus either the level of x_i rises, or the level stays equal and the index rises by at least 1. In either case $\phi(x_i)$ drops by at least 1.

Therefore every traversed node is paid for either by a direct charge or by a unit drop in potential. There are only $O(L)$ direct charges. Each call to `find` has amortized cost $O(L)$.

Since `new-set` takes constant time and `union` is just two calls to `find`, any sequence of m operations on n elements takes total time $O((m+n)L)$.

Choosing L. Since ranks are bounded above by $\log n$ and each f_k is increasing, take

$$L = \min\{k \geq 0 : f_k(2) > \log n\}.$$

Then every node of rank at least 2 has level at most $L-1$. The first few values come from

$$\begin{aligned}
f_0(2) &= 3,\\
f_1(2) &= f_0^{(2)}(2) = 4,\\
f_2(2) &= f_1^{(2)}(2) = 8,\\
f_3(2) &= f_2^{(2)}(2) = f_2(8) = 2048,\\
f_4(2) &= f_3^{(2)}(2) = f_3(2048) = f_2^{(2048)}(2048).
\end{aligned}$$

Any standard inverse of such an *Ackermann-style hierarchy* differs only by a constant and is called an *inverse Ackermann function*, written $\alpha(n)$. So disjoint union with path compression takes

$$O((m+n)\alpha(n))$$

amortized time.

Theorem 23.2 ([Tar75])**.** *There exists a data structure supporting* `union` *and* `find` *over n elements in $\alpha(n)$ amortized time, where $\alpha(n)$ is the inverse Ackermann function.*

23.3 Additional notes and materials

Spring 2026 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

23.4 Exercises

Exercise 23.1. Recall that a (binary) min-heap maintains a collection of elements and supports insertion and removing the minimum element in $O(\log n)$ worst-case time. Here we will develop and analyze a more sophisticated data structure called a *binomial heap*, which obtains $O(1)$ amortized time for insertion and $O(\log n)$ amortized time for deletion.

At a high-level, a binomial heap maintains a family of rooted trees over its elements, where each tree is in min-heap order. (Each node is less than or equal to any of its children). We will call each of these individual trees "heaps". The heaps in a binomial heap have a specific structure based on the following notion of "ranks".

- A rank 0 heap consists of a single node.
- For $i > 0$, a rank-i heap consists of two rank $i - 1$ heaps, where one heap is the child of the root of the other.

Two rank-i-heaps can be merged into a rank-$(i + 1)$-heap in constant time. Binomial heaps maintain the invariant that there is at most one heap of any rank i.

Now we give a high-level sketch of the `insert` and `remove-min` subroutines. Both have a postprocessing step that fixes the heap invariant. You will flesh out the implementation details in part 4 below.

`insert(`x`)`

/ High-level sketch; cf. part 4. */*

1. Create a rank 0 heap with node x.
2. Restore the heap invariant by repeatedly merging heaps of the same rank.

`remove-min`

/ High-level sketch; cf. part 4. */*

1. Scan the roots of all the heaps. Let H be the heap with the smallest root, let x be the root.
2. Remove H from the collection of heaps, and add the sub-heaps corresponding to the children of x.
3. Fix the heap invariant by repeatedly merging heaps of the same rank.
4. Return x.

Implement and analyze the binomial heap data structure via the following steps.

1. How big is a heap of rank i?
2. Suppose there are n elements. How many heaps are in the binomial heap?
3. Prove that for $i > 0$, a heap of rank i consists of a root node with i children, where the jth child is a heap of rank $j - 1$.
4. Flesh out the implementation details for `insert` and `remove-min`[1]. Prove that, for your implementation, `insert(`x`)` has $O(1)$ amortized time and `remove-min` has $O(\log n)$ amortized time.[2]

[1] How do we store the heaps? How can we efficiently "merge" the heaps of the same rank? Etc.

[2] Note that one could also show that binomial heaps have $O(\log n)$-worst case time for both operations.

Chapter 24

Push-relabel

This chapter presents the *push-relabel* max flow algorithm, a different and (in a sense) dual approach to the augmenting path algorithms previously discussed. Besides being interesting in its own right, this topic serves two additional purposes for our class. First, it gives us new and competitive running times for the maximum flow problem. In particular we will obtain the following four different running times by variants of the same push-relabel framework:

1. $O(mn^2)$ for general capacities.
2. $O(n^3)$ for general capacities.
3. $O(n^2\sqrt{m})$ for general capacities.
4. $O(mn + n^2 \log(U))$ for integer capacities between 1 and U.

Of course the second running time dominates the first, and the third running time dominates the second. The slower algorithms are included because they each introduce new ideas that continue to be used in the faster algorithms. We omit from our discussion a version that runs in $O(mn \log(n^2/m))$ that requires some data structures (called link-cut trees, see Chapter 31) that we have not introduced; this algorithm is provided in Section 24.3 for the interested reader.

The second reason we present these algorithms is that their analyses use potential functions to great effect. This provides us with further examples of amortized analysis that, in contrast to Chapter 22, are not data structures.

Throughout this section, let $G = (V, E)$ be a directed graph and let $c : E \to \mathbb{R}_{\geq 0}$ be a fixed set of capacities. Let $s, t \in V$ be fixed source and sink vertices, respectively.

24.1 Push-Relabel Max Flow

The push-relabel framework for maximum (s, t)-flow was first introduced by Goldberg [Gol85] and further developed by Goldberg and Tarjan [GT88]. This approach departs

from the Ford-Fulkerson framework in the sense that it does not augment along paths and may not obtain a feasible flow until the very end. The two main components of the algorithm are a *preflow* and a set of *vertex labels*, which are relaxations of flows and distance labels (from s), respectively. These relaxations (defined below) are similar enough to flows and distance labels to allow us to define residual graphs and forwards/backwards/neutral edges, and similar to blocking flows, one can explore many different interesting ideas within this framework.

24.1.1 Preflows

An (s,t)-*preflow* is a relaxation of an (s,t)-flow where non-terminals are allowed to have excess flow. Formally, it is a vector $f : E \to \mathbb{R}_{\geq 0}$ subject to the following invariants.

1. *Capacity constraints:* For all edges $e \in E$, the flow on e is at most the capacity:
$$0 \leq f(e) \leq c(e).$$
2. *One-sided conservation constraints.* For all non-terminals $v \in V - \{s,t\}$, the total flow entering v is at least the total flow leaving v:
$$\sum_{(u,v)\in\delta^-(v)} f(u,v) \geq \sum_{(v,u)\in\delta^+(v)} f(v,u).$$

The (nonnegative) difference between the amount of flow entering a (non-terminal) vertex v and the amount of flow leaving it is called the *excess* or the *net flow* at v. For ease of notation, we denote it as

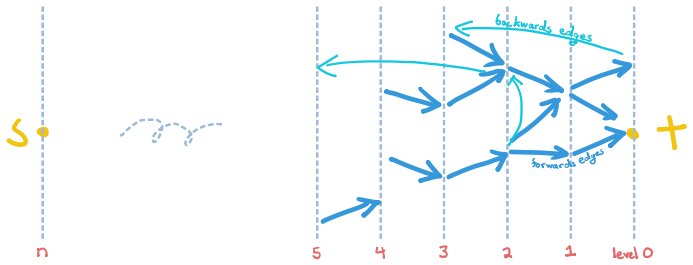
$$\hat{f}(v) \stackrel{\text{def}}{=} \sum_{e\in\delta^-(v)} f(u,v) - \sum_{e\in\delta^+(v)} f(v,u).$$

We define the residual graph $G_f = (V, E_f)$ of a preflow f in the same way as for flows: for each edge e with $f_e > 0$, we decrease the capacity on e by f_e, and increase the capacity in the opposite direction by f_e. This may introduce new (reversed) edges to and drop capacitated edges from the input graph.

Lemma 24.1. *Let f be a preflow. Then f decomposes to a path packing that contains $\hat{f}(v)$ units of fractional (s,v)-paths for every $v \in V - \{s\}$. In particular, f contains an (s,t)-flow of size $\hat{f}(t)$.*

Proof. Create a new auxiliary vertex t', and connect every vertex with positive excess (include the sink t) to t' with an edge with infinite capacity. Consider the (s, t')-flow f' where we send all the positive excess to t' via the auxiliary edges. This flow decomposes to a fractional (s, t')-path packing that includes $\hat{f}(v)$ paths that go through the auxiliary (v, t') edge. Removing this last edge gives the desired path decomposition. □

24.1.2 Labels



The push-relabel framework[1] maintains a preflow f and a set of *vertex levels* $\ell : V \to \mathbb{Z}_{\geq 0}$ such that:

(I) The sink has level 0 *and the source has level* n*:*

$$\ell(t) = 0 \text{ and } \ell(s) = n.$$

(II) Every residual edge goes down (towards t*) by at most one level:*

$$\ell(v) \geq \ell(u) - 1 \text{ for all } (u, v) \in E_f.$$

Initial preflow and labels. The initial preflow f starts with an empty flow and then saturates all the edges leaving the source s:

$$f(u, v) = \begin{cases} c(u, v) & \text{if } u = s \\ 0 & \text{otherwise.} \end{cases}$$

The level of s is set to n (as required) and all other vertices start at level 0:

$$\ell(v) = \begin{cases} n & \text{if } v = s \\ 0 & \text{otherwise.} \end{cases}$$

[1] Frank [Fra11] presents a slight variation of this, where s may have a level $\leq n$, which is particularly convenient for the Hao and Orlin [HO94] algorithm discussed later. The most important features are that the sink is at level 0, and each edge in the residual graph goes down at most one level.

Not that the initial preflow saturates and removes all edges leaving s from the residual graph, so we satisfy (II).

Forward, backward, and neutral edges. We say that a residual edge $(u, v) \in E_f$ is *forward* if $\ell(u) > \ell(v)$; i.e., the edge goes down the levels (towards t). By Item (II), (u, v) is forwards iff

$$\ell(u) = \ell(v) + 1.$$

Likewise we have a *neutral* edge when $\ell(u) = \ell(v)$, and a *backward* edge when $\ell(u) < \ell(v)$. We say that a vertex u "has a forward edge" if in particular there is a forward edge starting from u.

Levels as a proxy for distance. While $\ell : V \to \mathbb{Z}_{\geq 0}$ are not actually distance labels, invariants Items (I) and (II) install just enough structure ℓ to serve a similar role.

Lemma 24.2. *For all $x, y \in V$,*

$$\ell(x) - \ell(y) \leq d(x, y),$$

where $d(x, y)$ is the (unweighted) distance from x to y in the residual graph. In particular, $d(v, t) \geq \ell(x)$ for all vertices v.

Proof. Suppose there is an (x, y)-path of k edges. Each edge in the path can go down at most one level. The first inequality follows.

The second inequality follows from the first by observing that $\ell(t) = 0$. □

An interesting consequence of Lemma 24.2, applied to s (which has $\ell(s) = n$), is that $d(s, t) \geq n$ — which can only mean the following:

Corollary 24.3. *There is no (s, t)-path in the residual graph from s to t.*

This observation implies that the set of vertices reachable from s in the residual graph, as well as the set of vertices that can reach t in the residual graph, each induce an (s, t)-cut. In this sense push-relabel is dual to augmenting path algorithms — the preflow f always induces an (s, t)-cut (via its residual graph). This cut is not necessarily the minimum (s, t)-cut if f is not a feasible flow. Conversely, if f is also a flow (with no non-terminals with positive excess), then these (s, t)-cuts and f mutually certify one another as the minimum (s, t)-cut and maximum (s, t)-flows, respectively. In fact, a "level-set" induced by the levels ℓ will always induce a minimum (s, t)-cut, as follows.

Lemma 24.4. *Let f be an (s,t)-preflow and ℓ a set of levels satisfying the invariants Items (I) and (II). Suppose f is also an (s,t)-flow. Then f is a maximum (s,t)-flow. Moreover, there is an index $i \in \{1,\dots,n-1\}$ such that for $U = \{v : \ell(e) > i\}$, $\delta^+(U)$ induces a minimum (s,t)-cut.*

Proof. As discussed, f is maximum because there is no (s,t)-path in the residual graph. To extract the minimum (s,t)-cut from ℓ, observe that there exists an index $i \in \{1,\dots,n-1\}$ for which there are no vertices of level i. Let $U = \{v : \ell(e) > i\}$; since the ith level is empty, we have $\bar{U} = \{v : \ell(e) < i\}$. U contains s and $\bar{U}$ contains t, so $\delta^+(U)$ represents an (s,t)-cut in the input graph. By Item (II), there are no edges from U to $\bar{U}$ in the residual graph of f. This implies that in the input graph, $\delta^+(U)$ has capacity equal to the size of f. □

24.1.3 The generic push-relabel algorithm and analysis

There are many different push-relabel algorithms but they generally follow the same framework consisting of two types of operations, called *push* and *relabel*. Within this framework there are different strategies for employing these operations, some of which we discuss in greater detail below.

Recall that the only difference between a flow and a preflow is that a preflow allows intermediate vertices to carry positive excess. In fact, by Lemma 24.4, this is the only difference between a preflow and a *max* flow. Call a non-terminal vertex *active* if $\hat{f}(v) > 0$. Loosely speaking, push-relabel algorithms try to route the excess from active vertices, one vertex at a time, towards the sink t or the source s. (The latter implies that there was too much excess flow floating around for all of it to be routed to t.) Given an active vertex v, the algorithm executes one of the following two local operations at v.

1. We *push* flow along a forward edge e leaving v. Usually[2] we push as much flow as possible subject to the preflow constraints. This quantity is the minimum of the residual capacity of e and the excess at v. It is helpful in the analysis to classify each push as follows.
   (a) A *saturating push* is a push that uses up all of the capacity of the edge.
   (b) A *non-saturating push* is a push that does not use up all of the capacity of the edge, and (necessarily) uses all of the excess at the vertex.
2. We *relabel* the level of v to the next level. To preserve invariant (II), this is only permitted when v has no forward edges.

[2]Some scaling algorithms do not necessarily push the maximum amount.

The algorithm repeats the operations above until there are no more active vertices. At that point, by Lemma 24.4, f is a maximum flow.

It is not at all obvious that this algorithm should terminate. At a high level, we want to extinguish all of the active vertices. But repeatedly pushing flow out of one active vertex v, until v is no longer active, may activate many more vertices one level below! It's not clear that we're making much progress.

We will analyze the push-relabel framework by bounding each of the above operations — saturating pushes, non-saturating pushes, and relabelings — separately. Besides these "basic operations", one needs to account for the running time overhead for selecting an active vertex, identifying a forward edge from a given active vertix, and so forth. In most cases, it is easy to see how to use some simple data structures to facilitate these operations (e.g., tracking the active vertices in a list), and the bottleneck is just the total number of basic operations.

Relabelings.

Lemma 24.5. *For each vertex v, $\ell(v)$ is nonnegative, nondecreasing, and bounded above $2n-1$. Consequently, each vertex is relabeled at most $2n-1$ times and there are at most $O(n^2)$ relabelings in all.*

Proof. Indeed, if v is active, then there must be a path in the residual graph from v to s. But the level of s is fixed at n. So every active vertex v has level $\ell(v) \leq \ell(s) + n - 1 = 2n - 1$. Moreover, only active vertices get relabeled. □

Saturating pushes.

Lemma 24.6. *There are at most $n-1$ saturating pushes to any fixed edge. Therefore there are at most $O(mn)$ saturating pushes total.*

Proof. A saturating push along an edge $e = (u, v)$ removes u from the residual graph. Whenever $e = (u, v)$ is removed from the residual graph, we have $\ell(v) < \ell(u)$. The edge e only reappears after pushing along the opposite edge (v, u), which requires $\ell(u) < \ell(v)$. Since levels are monotonically increasing, this requires $\ell(u)$ to have increased by at least two. But $\ell(u)$ increases at most $2n - 1$ times in all. □

Non-saturating pushes.

The trickiest to analyze is non-saturating pushes and here we employ amortized analysis.

Lemma 24.7. *There are at most $O(mn^2)$ non-saturating pushes.*

Proof. Define a potential function by

$$\Phi = \sum_{v \text{ active}} \ell(v).$$

Each non-saturating push decreases Φ by one. A saturating push increases Φ by at most $O(n)$. A relabel increases Φ by one.

Since there are at most $O(mn)$ saturating pushes, and $O(n^2)$ relabels, the total increase to Φ is $O(mn^2)$. Moreover, Φ is initially zero, and always nonnegative. Charging each non-saturating push to a decrease in Φ, we conclude that there are at most $O(mn^2)$ non-saturating pushes. □

Thus push-relabel requires at most $O(mn^2)$ of the push/relabel operations. With some minor bookkeeping with simple data structures, it is easy to implement the algorithm in $O(mn^2)$ time.

The bottleneck is non-saturating pushes. The relabelings and saturating pushes – summing to $O(mn)$ total operations – are actually very fast. We now explore different variations of push-relabel that obtain better running times; each of these strategies are designed to limit the number of non-saturating pushes.

24.2 Top-down push-relabel

Section 24.1.3 gave a very general description and analysis of the push-relabel algorithm. We introduced the two basic operations, push and relabel, with pushes classified further as either saturating or non-saturating, and showed that *any* algorithm following these operations makes $O(mn^2)$ total operations. The bottleneck is the $O(mn^2)$ non-saturating pushes. There are at most $O(n^2)$ relabels and $O(mn)$ saturating pushes.

There are many possible strategies within this framework, and different approaches lead to better upper bounds (especially for non-saturating pushes) and different performance in practice. The main design decision is deciding which active vertex to select at a given moment in time. Here are some examples.

1. Select the active vertex v with the highest label $\ell(v)$.
2. Select the active vertex v with the lowest label $\ell(v)$.
3. Select active vertices v in FIFO order (e.g., add vertices to a queue as they become active).
4. Select active vertices v in LIFO order (e.g., add vertices to a stack as they become active).

5. Select the active vertex v with greatest excess $\hat{f}(v)$.
6. Select the active vertex v with least excess $\hat{f}(v)$.
7. Select an active vertex v uniformly at random.

In this section we consider the first strategy, which always addresses the active vertex of highest level. We call this the *top-down push-relabel* algorithm.

Lemma 24.8. *In top-down push-relabel, there are at most $O(n^2)$ non-saturating pushes from any fixed vertex v. Therefore there are at most $O(n^3)$ non-saturating pushes in all.*

Proof. Suppose we make a non-saturating push from a vertex v, making v inactive. Since v was the highest level active vertex, there is no way for v to receive flow and become active without some other vertex being promoted to a higher label. Each vertex gets promoted at most $O(n)$ times. □

The lemma above reduces the number of non-saturating pushs from $O(mn^2)$ to $O(n^3)$. We leave it to the reader to show that the algorithm can also be implemented in $O(n^3)$ time. (Here one needs to arrange some simple data structures to quickly find the top-level active vertex, identify a forwards edge to push flow on, etc.) Altogether we obtain the following running time.

Theorem 24.9. *There is a highest-level push-relabel algorithm that runs in $O(n^3)$ time.*

$O(n^3)$ is better than our best strongly polynomial algorithm in **??** among those that did not leverage sophisticated data structures. We note that the FIFO strategy also obtains a $O(n^3)$-running time; we leave the proof to the reader as exercise 24.1.

24.2.1 $O\left(n^2\sqrt{m}\right)$ non-saturating pushes

Cheriyan and Maheshwari [CM89] and Tunçel [Tun94] showed that for top-down push-relabel, one can obtain a better upper bound of $O(n^2\sqrt{m})$ non-saturating pushes.

Theorem 24.10 (Cheriyan and Maheshwari [CM89] and Tunçel [Tun94])**.** *The top-down push-relabel algorithm (appropriately implemented) runs in $O(n^2\sqrt{m})$ time.*

Proof. The following argument is due to Cheriyan and Mehlhorn [CM99]. We restrict our attention to showing that the highest-label rule makes at most $O(n^2\sqrt{m})$ non-saturating pushes; we leave it to the reader to fill in the details of a $O(n^2\sqrt{m})$-time implementation.

Define a potential Φ by

$$\Phi = \sum_{v \text{ active}} |w : \ell(w) \leq \ell(v)|.$$

This potential counts, for every active vertex v, the total number of vertices "below" or at the same level as v.

Φ is initially 0. A relabeling or a saturated push can increase Φ by at most n. So the total increase to Φ is at most $O(mn^2)$. On the other hand, a non-saturating push at a vertex v de-activates v and decreases Φ, by at least the number of vertices v at v's level, $|w : \ell(w) = \ell(v)|$. (In the worst case, the non-saturating push from v activates a new vertex w. Since $\ell(w) = \ell(v) - 1$, the difference in contribution between v and w is precisely the number of vertices at v's level.)

Call a level "small" if it has $< k$ vertices, and "big" if it has $\geq k$ vertices. Call a push "small" if it is from a vertex on a small level, and "big" if it is from a vertex at a big level.

Now, we always treat active vertices at the highest level. Let a "phase" be defined as a consecutive sequence of basic operations made by the algorithm on active vertices from the same level. Phases are terminated by increasing the highest active level (by relabeling an active vertex), or by decreasing the highest active level (by deactivating the last active vertex on the level with a push). Now there are only $O(n^2)$ relabels so we only increase the highest level $O(n^2)$ times. Each decrease can be attributed to a previous increase[3]. So there are at most $O(n^2)$ phases.

Let $k \in \mathbb{N}$ be a parameter to be determined. Call a non-saturating push "small" if the highest active level has $\leq k$ active vertices, and "big" if the highest active level has $> k$ active vertices. There are most k small non-saturating pushes per phase, hence $O(n^2 k)$ small non-saturating pushes overall. Meanwhile each big push decreases Φ by at least $O(k)$, so there are at least $O(mn^2/k)$ big pushes. Balancing terms at $k = \sqrt{m}$, we conclude that there are at most $O(n^2\sqrt{m})$ non-saturating pushes. □

24.3 Accelerating push-relabel with data structures

Goldberg and Tarjan [GT88] showed how to use dynamic data structures to accelerate the push-relabel running time algorithm. Goldberg and Tarjan specifically consider the FIFO strategy for selecting active vertices. However, it appears most strategies would work; the key idea is to use data structures drastically cut the computational work generated by non-saturating pushes. In fact, their argument does not actually

[3]Like an elevator.

decrease the total number of non-saturating pushes; rather, most of the non-saturating pushes are simulated efficiently by the data structures (similar to blocking flows).

Theorem 24.11. *With link-cut trees, the push-relabel algorithm can be made to run in $O(mn \log(n^2/m))$ time.*

Proof. For ease of exposition we will only show how to obtain an $O(mn \log(n))$ running time. The $O(mn \log(n^2/m))$ comes from a more careful application of these techniques. We refer the reader to Goldberg and Tarjan [GT88] or Goldberg's thesis [Gol87, Chapter 1] for these details.

We will use link-cut trees to make non-saturating pushes more efficient. Before discussing push-relabel, it is helpful to briefly recall the use of link-cut trees in accelerating blocking flows. The inspiration there was that whenever we find and augment along an (s, t), and remove a bottleneck edge from the residual graph, the remaining parts of the paths can still be useful. Rather than discard this progress, we maintain the explored edges in a dynamic tree. We use the `root` operation to quickly identify where the search left off, `link` to add an edge to the search, aggregate queries along the s-to-root path to identify bottleneck edges and route flow, and `cut` to delete edges as they are saturated and removed from the residual graph.See Chapter 31 for more details and explanations of the dynamic tree operations.

Let us return to push-relabel and first give some high-level motivation. We generally prefer to make saturating pushes or relabels as these already have good upper bounds. Of course only a non-saturating push may be available. When we execute a non-saturating push from (say) u to v, v becomes active, and we can keep searching at v for a saturating push or a relabeling. If indeed we can make then relabel v, or make a saturating push at v, then we can charge the preceding non-saturating push to this operation. But perhaps there is only a non-saturating push at v, and so we continue the search until we find something other than a non-saturating push. In this manner we find a path of non-saturating pushes terminated by either a saturating push or a relabeling. Of course the path may be long with many non-saturating pushes, so this approach is not yet efficient.

The idea is to store these non-saturating pushes (represented by their edges) in dynamic trees. As we search out non-saturating pushes (along forward edges), we keep these edges in a dynamic tree. We use the `root` operation to fast-forward to the leading non-saturating push-point, and then look for the next non-saturating push from there.

We can charge each newly discovered non-saturating push to inserting the edge in the dynamic forest. Meanwhile a forward edge $e = (u, v)$ is deleted from the dynamic forest data structure iff either

1. e is saturated.
2. v is relabeled.

In particular, a fixed edge is deleted at most $O(n)$ times. Thus an edge is also inserted at most $O(n)$ times, and we only have a "new" non-saturating push $O(n)$ times per edge.

This leads to an overall running time bounded by $O(mn)$ tree operations. We leave it to the reader to sort out the remaining implementation details, or otherwise refer the reader to the aforementioned references. □

24.4 Scaling push-relabel

The final version of push-relabel we present is a scaling algorithm due to Ahuja and Orlin [AO89]. Here we assume that the capacities $c : E \to [U]$ are integers between 1 and U (for some $U \in \mathbb{N}$); the running time will depend on $\log(U)$.

The high level idea as follows. For a fixed $\Delta > 0$, we maintain the invariant that

every vertex has excess at most 2Δ.

We only make non-saturating pushes of size exactly Δ. (That is, even if we could push more, we truncate the push to Δ units of flow.) We will always push an active vertex at the lowest possible level, which preserves the invariant that the maximum excess at most 2Δ.

We continue to push Δ units at a time until all active vertices have excess less than Δ. This means we can divide Δ in half while still maintaining the invariant that every vertex has excess at most 2Δ. We then continue with this smaller value of Δ.

Since we are repeatedly halving, it is convenient to maintain Δ as a power of 2. Initially we set $\Delta = 2^{\lfloor \log_2 U \rfloor}$. Meanwhile, the algorithm maintains integrality everywhere. Consequently, once we get to $\Delta < 1/2$, the invariant implies that every vertex has excess less than 1, which for integral values must be 0. That is, there are no active vertices once $\Delta < 1/2$. To get from $2^{\lfloor \log_2 U \rfloor}$ to below $1/2$, we halve Δ at most $O(\log(U))$ times before the algorithm terminates.

Now, consider the following potential functions which allows us to amortize non-saturating pushes.

$$\Phi = \frac{1}{\Delta} \sum_{v \in V} \hat{f}(v)\ell(v).$$

Lemma 24.12. *Relabeling and saturating pushes take $O(1)$ amortized time (w/r/t Φ). Non-saturating pushes take no amortized time.*

Proof. The first claims are easy; for non-saturating pushes, we observe that each non-saturating push moves Δ units of flow to a lower-level, □

The remaining operation to analyze – unique to scaling – is when we decrease Δ. Here we have the following.

Lemma 24.13. *Dividing Δ in half takes $O(n^2)$ amortized time.*

Proof. Observe the potential is at most $O(n^2)$ becuase $\hat{f}(v) \leq 2\Delta$ for all v. Halving Δ doubles the potential Φ, which gives the amortized running time. □

This leads to the following running time. The broad strokes of obtaining this running time are captured by the above lemmas; we leave it to the reader to flesh out the remaining implementation details (to identify the active vertex at lowest level, to identify a forwards edge at the active vertex, etc.).

Theorem 24.14 (Ahuja and Orlin [AO89])**.** *In a flow network with m edges, n vertices, and integer capacities between 1 and U, push-relabel with scaling (appropriately implemented) computes the maximum flow in $O(mn + n^2 \log(U))$ time.*

24.5 Additional notes and references

Preflows were introduced by Karzanov [Kar74], who employed them as part of a blocking flow subroutine. The push-relabel algorithm was pioneered by [Gol85], which was then improved by Goldberg and Tarjan [GT88]. Goldberg's thesis [Gol87, Chapter 1] is a good exposition of these developments. This lead to many follow-up works within the push-relabel framework, some of which we cover. There has also been experimental work on the algorithm [CG97; CGM98], by which push-relabel has acquired a reputation for being very good in practice.

The push-relabel algorithm is discussed in the algorithms textbook by Kleinberg and Tardos [KT06], and in books by Ahuja, Magnanti, and Orlin [AMO93], Schrijver [Sch03], Frank [Fra11], and Williamson [Wil19]; [Fra11; Wil19] also include the Hao-Orlin minimum cut algorithm from **??**. We also recommend the lecture notes by Blum and Gupta [BG13]. For the improved top-down push-relabel bound, we recommend a nice note by Tarjan [Tar13].

We did not discuss the minimum cut algorithm by Gabow [Gab95] which remains the best algorithm for small connectivities in multigraphs. Gabow's algorithm is based on Edmonds's [Edm75]'s directionless tree packing theorem. It is also a very nice example of matroid intersection, another topic unto itself. We may discuss Edmonds's

theorem [Edm75] and matroid intersection later in the course, at which point we will also discuss Gabow's algorithm [Gab95].

The rooted min-cut algorithm for simple graphs (**??**) is very recent [CQ21].

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten .pdf prepared before the lecture (and .note file).
- Handwritten .pdf annotated during the presentation (and .note file).
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

24.6 Exercises

Exercise 24.1. In Section 24.2, we described but did not analyze the "first-in first-out" strategy for selecting active vertices. (Vertices are placed in a queue as they are made active, from which we find the next active vertex.) Prove that this push-relabel algorithm (appropriately implemented) takes $O(n^3)$ to find the maximum flow.[4]

Exercise 24.2. One advantage of the relaxed definitions of preflows and labels is that we can make certain adjustments to the flow network that preserve the push-relabel invariants, so that we don't have to throw away all the computation we have already done. In this exercise, we will develop a "max flow data structure", so to speak,

[4]Hint: We already know that there are at most $O(mn)$ saturating pushes, which is at most $O(n^3)$. So the real question is why there are at most $O(n^3)$ non-saturating pushes.

that maintains the value of the max-flow under the following two changes to the network. The data structure starts with an instance of (s,t)-max flow and first runs the push-relabel algorithm to completion. The underlying graph then undergoes changes via the following operations.

1. `AddSourceEdge(`v`,`γ`)`: Add an edge (s,v) with capacity γ.
2. `RemoveSinkEdge(`v`, `γ`)`: Decrease the capacity of an edge (v,t) by γ. (You may assume that the capacity of the edge was previously $\geq \gamma$.).

The goal is to be able to recompute the value of the maximum (s,t)-flow faster than recomputing the maximum flow from scratch.

To keep things concrete let us analyze the following algorithm exactly. Initially we start with a capacitated and directed graph $G = (V,E)$ with m edges and n vertices, and. We run the top-down push-relabel algorithm until we obtain a max flow. Then we process a sequence of k `AddSourceEdge` and ℓ `RemoveSinkEdge` operations (possibly intermixed).

1. For each call to `AddSourceEdge(`v`,`γ`)`, we add the edge $e = (s,v)$ to the graph with capacity $c(e) = \gamma$. If $\ell(v) < \ell(s)$, we saturate e by setting $f_e = \gamma$. Note that this does not introduce a forwards edge and so we still satisfy Items (I) and (II). We resume the push-relabel flow algorithm with the existing preflow and labels until all active vertices have level n.
2. For each `DecreaseSinkEdge(`v`,`γ`)` we do the following. Let $e = (v,t)$. If $f_e > 0$, then we set $f_e = \min\{c(e) - \gamma, f_e\}$, possibly making v active. We decrease the capacity of e by γ. If the capacity of e is reduced to 0, then we remove e from the graph. This does not introduce any forwards edges and so we still satisfy Items (I) and (II). We resume the push-relabel flow algorithm with the existing preflow and labels until all active vertices have level n.

Let us interpret the above as a sort of data structure, that solves the initial flow problem in its initialization step, and then updates the flow as the two operations as described above. Prove the following amortized bounds on the number of basic operations executed by the algorithm.

1. `Init(`$G = (V,E)$`,`$c : E \to \mathbb{R}_{\geq 0}$`,`$s$`,`$t$`)` (where we compute the initial max flow) takes $O(mn^2)$ amortized basic operations.
2. `AddSourceEdge(`v`,`γ`)` takes $O(n^2)$ amortized basic operations.
3. `DecreaseSinkEdge(`v`,`γ`)` takes $O(n)$ amortized basic operations.

That is, after computing the initial flow, and then processing an intermixed sequence of k calls to `add-source-edge` ℓ calls to `decrease-sink-edge`, the total number of basic operations is at most $O((m+k)n^2 + \ell)$.

Here, we ask for "amortized basic operations" instead of "amortized time" because a full implementation would need to include some basic data structures to execute pushes and relabels in $O(1)$ and be able to quickly identify the next active vertex push or relabel. It is a bit long-winded to describe and analyze these data structures in detail, so we abstract them out, and ask you only to analyze the number of basic operations that are made.

(Food for thought: Can you get better bounds using the improved analysis of Section 24.2.1?)

Chapter 25

Randomized searching and sorting

25.1 First randomized algorithms

We introduce randomized algorithms via 3 basic problems: 3-SAT, sorting, and selection. For each we present randomized algorithms that are essentially optimal (if one does not mind that they are randomized). They are also very simple to describe and code. They might seem tricky to analyze, as the analysis involves probabilities and sorting through nondeterministic outcomes. We will see how focusing on *expected* performance renders the analysis surprisingly clean.

Let us briefly introduce the algorithms first as their striking simplicity may help motivate us to learn the mathematical tools we need to analyze them.

25.1.1 3-SAT.

We start with 3-SAT. The input to 3-SAT consists of a Boolean formula in conjunctive normal form (CNF) with 3 (distinct) variables per clause. For example,

$$f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$$

is a 3-SAT formula with $m = 4$ clauses and $n = 3$ variables. It is satisfied by the assignment $(x_1, x_2, x_3) = (\mathtt{t}, \mathtt{f}, \mathtt{t})$. ($\mathtt{t}$ denotes "true" and $\mathtt{f}$ denotes "false".) In the 3-SAT problem, we are given a 3-SAT formula $f(x_1, \ldots, x_n)$ with m clauses and n variables and want to find a satisfying assignment.

3-SAT is the quintessential NP-Complete search problem, and there is no polynomial time algorithm to solve it unless $P = NP$ [Coo71; Lev73]. But this does not prevent us from trying to *approximate* the problem. The goal is now to find an assignment that satisfies as many clauses as possible. Of course an exact algorithm for this maximization version implies a polynomial time algorithm for the decision version. Instead we will design algorithms that do not guarantee the maximum, but are competitive up to a multiplicative factor when compared to the optimum solution.

`random-SAT`$(f(x_1, \ldots, x_n))$

1. For each $i \in [n]$, draw $x_i \in \{\mathtt{t}, \mathtt{f}\}$ independently and uniformly at random.
2. Return $x_1, \ldots, x_n$.

Figure 25.1: A randomized approximation algorithm for SAT.

Given a SAT formula f, let OPT denote the maximum number of clauses that are satisfiable. For $\alpha \in [0, 1]$, an *α-approximation algorithm* for SAT is an algorithm that produces an assignment that satisfies at least $\alpha \operatorname{OPT}$ clauses. While obtaining an exact algorithm is NP-Hard, for fixed $\alpha < 1$, it is not necessarily NP-Hard to obtain an α-approximation algorithm for SAT.

Here is a simple randomized approximation algorithm for SAT. Given a formula $f(x_1, \ldots, x_n)$, for each variable x_i, flip a fair coin and assign $x_i = \mathtt{t}$ or $x_i = \mathtt{f}$ accordingly. (See Fig. 25.1 for pseudocode.) We will show that, on average, this random assignment satisfies at least $(7/8)m$ clauses out of m total. Moreover, we will be able to *derandomize* the above algorithm and obtain a deterministic algorithm that (always) satisfies at least $(7/8)m$ clauses.

The randomized algorithm is extremely simple and basically oblivious to the input. Surely it isn't very good. But in fact it is the best possible polynomial time algorithm unless $P = NP$. The *PCP theorem* states that for all constants $\epsilon > 0$, getting better than a $(7/8 + \epsilon)$-approximation to 3SAT is NP-Hard. "PCP" standards for *probabilistically checkable proofs*. The PCP theorem gives similar *hardness of approximation* results for many other problems besides SAT. The PCP theorem (as the name suggests) has strong connections to randomized algorithms. We will discuss related randomized topics, and maybe parts of the proof of the PCP theorem, later in the course.

Thus in this lecture we will show the first part of the following theorem. Topics in later chapters will hopefully shed some light on the second half of the theorem.

Theorem 25.1. *There is a polynomial time algorithm that given any 3-SAT formula computes an assignment that satisfies at least $\frac{7}{8}$th of the clauses. Moreover, for all $\epsilon > 0$, a polynomial time approximation algorithm with approximation ratio $\left(\frac{7}{8} + \epsilon\right)$ implies that $P = NP$.*

25.1.2 Sorting.

The next problem we discuss is sorting. The goal is to take an unordered list of n comparable elements (e.g., numbers) and return them as a list in sorted order.

`quick-sort(`$A[1..n]$`)`

/ For simplicity we assume all the elements are distinct. Otherwise, break ties consistently. */*

1. If $n \leq 1$ then return A.
2. Select $i \in [n]$ uniformly at random.
3. $B[1..k] \leftarrow$ recursively sort the set of elements less than $A[i]$.
4. $C[1..\ell] \leftarrow$ recursively sort the set of elements greater than $A[i]$.
5. Return the concatenation of B, $A[i]$, and C.

Figure 25.2: A randomized sorting algorithm.

The reader likely knows that the `merge-sort` algorithm runs in $O(n \log n)$, and that there is a $\Omega(n \log n)$-time lower bound for any sorting algorithm in the comparison model. Here we will study a randomized algorithm that is remarkably simple, called `quick-sort`, that is often the preferred one in practice. The idea is very simple: select an element *uniformly at random* out of the list to serve as a *pivot*. Divide the elements into those smaller and larger than the pivot, and recurse on both halves. See Fig. 25.2 for pseudocode.

What is the worst-case running time of `quick-sort`? It is important to clarify what we mean by "worst-case". Observe that the running time of `quick-sort` is proportional to the total number of comparisons made by the algorithm. It is certainly possible that the algorithm makes $\Omega(n^2)$ comparisons. (How?) So in a limited sense that algorithm has a worst-case $O(n^2)$ time.

As the algorithm is randomized, a more useful measure is the *average* number of comparisons. We will show that `quick-sort` takes $O(n \log n)$ time on average *against any input*. This is still a worst-case analysis in the sense that it holds for *all inputs*. (This is not to be confused with the performance of an algorithm against a *randomized input from a fixed distribution* — that is called *average case analysis*.) We will also show that the algorithm takes $O(n \log n)$ time with extremely high probability. We prove, in Section 25.4:

Theorem 25.2. *Given a list of n comparable elements,* `quick-sort` *returns the elements in a sorted list in $O(n \log n)$ expected time and with high probability.*

25.1.3 Selection.

The last problem we mention is selection. The input, similar to sorting, includes an unordered list of n comparable elements. Given an index $k \in [n]$, the goal is to find

`quick-select(`$A[1..n]$`,`k`)`

/ The goal is to find the rank k element in $A[1..n]$. We assume for simplicity that all the elements are distinct. */*

1. Randomly select $i \in [n]$ uniformly at random.
2. Compute the rank ℓ of $A[i]$. *// $O(n)$*
3. If $\ell = k$, then return $A[i]$.
4. If $\ell > k$, then recursively search for the rank k element among the set of $\ell - 1$ elements less than $A[\ell]$, and return it.
5. If $\ell < k$, then recursively search for the rank $k - \ell$ element among the set of $n - \ell$ elements greater than $A[\ell]$, and return it.

Figure 25.3: A randomized algorithm for selection.

the kth smallest element in the list.

The obvious solution is to sort the list, which takes $O(n \log n)$ time. Famously, one can do better: the "median-of-medians" divide-and-conquer algorithm of Blum, Floyd, Pratt, Rivest, and Tarjan [BFP+72] runs in $O(n)$ time. This algorithm is a bit tricky, both to describe and to analyze. Here then is a simpler alternative, which is similar to quick-sort: pick a pivot uniformly at random, and compute its rank ℓ. Depending on whether $k = \ell$, $k < \ell$, or $k > \ell$, either return the pivot, recurse on the subset of elements less than the pivot, or recurse on the subset greater than the pivot. See Fig. 25.3 for the pseudocode.

We will prove the following theorem which states that `quick-select` takes $O(n)$ time in expectation. Or rather, *you* will prove it, in exercise 25.12, employing the new tools gained from analyzing randomized SAT and sorting.

Theorem 25.3. `quick-select(`$A[1..n]$`,`k`)` *returns the rank k element in $O(n)$ time in expectation and with high probability.*

25.2 Basic probability

To analyze these algorithms we need to know some basic probability. Let us start with a familiar example.

Suppose we flip a coin, and then press pause as it is flipping in mid-air. Will it land heads or tails? Obviously, *we don't know yet.* Yet we can state without ambiguity that *half the time it will land heads*, and *half the time it will land tails.* What do we mean when we say that half the time it will land heads? There is of course only

one coin, and we can't split the coin in half. We are imagining that, if we repeat the experiment many times, we would expect half the coin tosses to come up heads.

This simple example, which we all understand thoroughly, points to a deeper feature of probability: *probability allows us to interpret fractional values as discrete ones*. Here, "half heads" does not mean that "half the coin will come up heads", which is total nonsense; rather it means that half *the time* the coin will come up heads.

The formal rules of probability are simple and intuitive. (The tricky part is adhering to them.) Likely the reader has had some acquaintance with random events and variables before, but we will still review the basics. Probability theory assumes an uncertain world where *events* occur with fixed (but not necessarily known) numerical probabilities. Each event A has a probability between 0 and 1, denoted

$$\mathbf{P}[A] \in [0, 1].$$

For every event A, there is the complementary event, $\bar{A}$, of A *not occurring*. We always have

$$\mathbf{P}[A] + \mathbf{P}\left[\bar{A}\right] = 1. \tag{25.1}$$

Joint events. For any two events A and B, one can define the *conjunctive event* that *both A and B* occurs: denoted

$$\text{"}A \wedge B\text{" or "}A \cap B\text{" or "}A \text{ and } B\text{".}$$

It is common to write $\mathbf{P}[A, B]$ as a shorting for $\mathbf{P}[A \wedge B]$.

In general, given A and B, there are four disjoint events induced by their combination:

1. $A \wedge B$: The event that both A and B occur.
2. $A \wedge \bar{B}$: The event that A occurs and B does not.
3. $\bar{A} \wedge B$: The event that A does not occur and B does.
4. $\bar{A} \wedge \bar{B}$: The event that neither A nor B occcurs.

The four joint events listed are mutually exclusive – at most one of them can be realized – and exhaustive – at least one of them will be realized. So we have the following identity:

$$\mathbf{P}[A \wedge B] + \mathbf{P}\left[A \wedge \bar{B}\right] + \mathbf{P}\left[\bar{A} \wedge B\right] + \mathbf{P}\left[\bar{A} \wedge \bar{B}\right] = 1$$

Suppose event A occurs with positive probability. Whenever A occurs, then of course exactly one of B occurs or $\bar{B}$ occurs. So we have

$$\mathbf{P}[A] = \mathbf{P}[A, B] + \mathbf{P}\left[A, \bar{B}\right].$$

If we divide both sides by $\mathbf{P}[A]$, we have

$$1 = \frac{\mathbf{P}[A, B]}{\mathbf{P}[A]} + \frac{\mathbf{P}\left[A, \bar{B}\right]}{\mathbf{P}[A]}.$$

This equation looks probabilistic – we have two nonnegative terms summing to 1. The first term on the right-hand side, $\mathbf{P}[A, B] / \mathbf{P}[A]$, can be interpreted as follows:

> *of the times that event A occurs, B also occurs $\mathbf{P}[A, B] / \mathbf{P}[A]$ fraction of the time.*

We call this the *conditional probability of event B conditional on event A*, denoted $\mathbf{P}[B \mid A]$ and defined as the ratio

$$\mathbf{P}[B \mid A] = \frac{\mathbf{P}[B, A]}{\mathbf{P}[A]}.$$

Likewise we have the conditional probabilities $\mathbf{P}\left[\bar{B} \,\middle|\, A\right]$, $\mathbf{P}\left[B \,\middle|\, \bar{A}\right]$, $\mathbf{P}\left[\bar{B} \,\middle|\, \bar{A}\right]$, and so forth.

We always have

$$\mathbf{P}[A \wedge B] \leq \min\{\mathbf{P}[A], \mathbf{P}[B]\}$$

(see exercise 25.1). It is generally *not* true that

$$\mathbf{P}[A \mid B] = \mathbf{P}[A] \tag{25.2}$$

for two events A and B. The equation above is equivalent to the one with A and B flipped, as well as

$$\mathbf{P}[A \wedge B] = \mathbf{P}[A]\,\mathbf{P}[B] \tag{25.3}$$

Events A and B are said to be *independent events* in the special case where Eqs. (25.2) and (25.3) holds.

Unions of events. We also have the *disjunctive event* that *either A or B* occurs, denoted

$$\text{“}A \vee B\text{” or “}A \cup B\text{” or “}A \text{ or } B\text{”}.$$

If $A \vee B$ occurs, then exactly one of the following events occurs:

$$A \wedge B,\ A \wedge \bar{B},\ \bar{A} \wedge B.$$

Consequently we have

$$\mathbf{P}[A \vee B] = \mathbf{P}[A \wedge B] + \mathbf{P}\left[A \wedge \bar{B}\right] + \mathbf{P}\left[\bar{A} \wedge B\right].$$

Recall that $\mathbf{P}[A] = \mathbf{P}[A \wedge B] + \mathbf{P}\left[A \wedge \bar{B}\right]$, and similarly for $\mathbf{P}[B]$. Adding $\mathbf{P}[A \wedge B]$ to both sides of the identity above gives

$$\mathbf{P}[A \vee B] + \mathbf{P}[A \wedge B] = \mathbf{P}[A] + \mathbf{P}[B].$$

This identity reflects a venn-diagram, so to speak, where the two regions A and B “sum” to their union $A \vee B$ and their intersection $A \wedge B$.

Dropping the nonnegative term $\mathbf{P}[A \wedge B]$ from the identity above gives rise to the following extremely useful *union bound*.

Lemma 25.4 (Union bound)**.** *For any two events A and B,*

$$\mathbf{P}[A \vee B] \leq \mathbf{P}[A] + \mathbf{P}[B],$$

with equality iff $\mathbf{P}[A \wedge B] = 0$.

Random variables. A *finite random variable* models an unrealized and uncertain object X that takes one of a finite set of values, $\{x_1, \ldots, x_k\}$.[1] For each outcome x_i, “X equals x_i” is an event, with a fixed probability, denoted $\mathbf{P}[X = x_i]$. These probabilities sum to 1:

$$\sum_{i=1}^{k} \mathbf{P}[X = x_i] = 1.$$

For example, we can describe a coin toss as a random variable $X \in \{\texttt{heads}, \texttt{tails}\}$. If the coin comes up heads, then $X = \texttt{heads}$. If the coin comes up tails, then $X = \texttt{tails}$. For a fair coin we have

$$\mathbf{P}[X = \texttt{heads}] = \mathbf{P}[X = \texttt{tails}] = \frac{1}{2}.$$

[1] One can also define continuous variable (e.g., that take values continuously between 0 and 1), where sums are replaced by variables.

Note that these two probabilities sum to 1.

If we have two random variables $X \in \{x_1, \dots, x_k\}$ and $Y \in \{y_1, \dots, y_\ell\}$, then their product (X, Y) forms a random variable in the set $\{(x_i, y_j) : i = 1, \dots, k,\ j = 1, \dots, \ell\}$. We have probabilities of the form

$$\mathbf{P}[X = x_i,\ Y = Y_j].$$

It is not generally true

$$\mathbf{P}[X = x_i,\ Y = y_k] = \mathbf{P}[X = x_i]\,\mathbf{P}[Y = y_k]$$

In the special case where the above holds for all x_i and y_j, then X and Y are said to be *independent*. This is equivalent to saying that for all x_i and y_j, the events $X = x_i$ and $Y = y_j$ are independent.

For example, suppose $X, Y \in \{\texttt{heads}, \texttt{tails}\}$ both describe coin tosses. If they described different coin tosses, then they would be independent random variables, and each combination of heads and tails would occur with probability .25. That is,

$$\begin{aligned}&\mathbf{P}[X = \texttt{heads},\ Y = \texttt{heads}] = \mathbf{P}[X = \texttt{heads},\ Y = \texttt{tails}]\\ &= \mathbf{P}[X = \texttt{tails},\ Y = \texttt{heads}] = \mathbf{P}[X = \texttt{tails},\ Y = \texttt{tails}] = \frac{1}{4},\end{aligned}$$

Thus X and Y are independent random variables. If they described the *same coin*, then we would have

$$\mathbf{P}[X = \texttt{heads},\ Y = \texttt{heads}] = \mathbf{P}[X = \texttt{tails},\ Y = \texttt{tails}] = \frac{1}{2},$$

while

$$\mathbf{P}[X = \texttt{heads},\ Y = \texttt{tails}] = \mathbf{P}[X = \texttt{tails},\ Y = \texttt{heads}] = 0.$$

Here, X and Y are *not* independent.

Averages. When a random variable X takes on real values, we can have a well-defined and quantitative notion of an "average", more formally called the *expected value*.

Definition 25.5. *Let $X \in \mathbb{R}$ be a real-valued random variable that has a finite set of possible values. Then the* expected value of X*, denoted $\mathbf{E}[X]$, is the weighted sum*

$$\mathbf{E}[X] \stackrel{\text{def}}{=} \sum_x \mathbf{P}[X = x] \cdot x;$$

where the sum is over all values of x where $\mathbf{P}[X = x] > 0$.

For continuous random variables, the sum would be replaced by an integral.

The average quantity of a random variable is very intuitive; the reader is likely used to discussing averages in the sense defined above. (e.g., the *average* midterm score.) The following identity, called *linearity of expectation*, is perhaps less intuitive; however it follows rather plainly from the definition of expectation.

Theorem 25.6. (Linearity of expectation.) *Let $X, Y \in \mathbb{R}$ be two random variables. Then*

$$\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y].$$

The proof of linearity of expectation is given as exercise 25.5. The reader may want to first consider the simple case where $X \in \{x_1, x_2\}$ takes on exactly two values, and $Y \in \{y_1, y_2\}$ takes on exactly two values. One can generalize to finite sets from there.

Observe that linearity of expectation does not make any assumptions about how X and Y are structured or related. This makes linearity of expectation extremely useful and often leads to surprising observations.

A simple example of linearity of expectation is as follows. Consider a population of people with various heights. Let X and Y be two quantities obtained by the following experiment. Draw one person uniformly at random. Let X be the length from the waist of this person to the top of their head. Let Y be the length from the waist of this person to the ground. $X + Y$ gives to the total height of the person. Note that X and Y are highly dependent, since they both measure the same (randomly drawn) person. Linearity of expectation says:

$$\underbrace{(\text{average height})}_{\mathbf{E}[X+Y]} = \underbrace{\begin{pmatrix}\text{average length}\\ \text{from waist up}\end{pmatrix}}_{\mathbf{E}[X]} + \underbrace{\begin{pmatrix}\text{average length}\\ \text{from waist down}\end{pmatrix}}_{\mathbf{E}[Y]}.$$

Of course, this makes total sense.

Conditional expected values. We also have conditional expected value analogous to how we have conditional probabilities. For a random variable X and an event A, $\mathbf{E}[X \mid A]$ denotes the expected value of X conditional on A occurring. Formally we have

$$\mathbf{E}[X \mid A] = \sum_x x \, \mathbf{P}[X = x \mid A].$$

The reader should verify that

$$\mathbf{E}[X] = \mathbf{E}[X \mid A] \, \mathbf{P}[A] + \mathbf{E}\left[X \,\middle|\, \bar{A}\right] \mathbf{P}\left[\bar{A}\right]$$

for all events A.

Now, suppose we have two random variables X and Y, and a real-valued function $f(X, Y)$ of these variables. Then by definition we have

$$\mathop{\mathbf{E}}_{X,Y}[f(X,Y)] = \sum_{(x,y)} f(x,y)\,\mathbf{P}[X = x, Y = y]. \tag{25.4}$$

Here we have annotated X, Y under the $\mathbf{E}[\cdots]$ to emphasize that the randomization is over X and Y jointly. On the other hand consider the following *nested* expected value,

$$\mathop{\mathbf{E}}_{X}\Big[\mathop{\mathbf{E}}_{Y}[f(X,Y) \mid X]\Big].$$

This version describes the average of an experiment where we first observe X, and then conditional on X, we observe Y and evaluate $f(X, Y)$. Formally the expression expands out to

$$\begin{aligned}\mathop{\mathbf{E}}_{X}\Big[\mathop{\mathbf{E}}_{Y}[f(X,Y) \mid X]\Big] &= \sum_{x} \mathop{\mathbf{E}}_{Y}[f(X,Y) \mid X = x]\,\mathbf{P}[X = x] \\ &= \sum_{x}\sum_{y} f(x,y)\,\mathbf{P}[Y = y \mid X = x]\,\mathbf{P}[X = x]. \end{aligned}\tag{25.5}$$

Since $\mathbf{P}[X = x, Y = y] = \mathbf{P}[Y = y \mid X = x]\,\mathbf{P}[X = x]$, the RHSs of Eqs. (25.4) and (25.5) are equal, hence

$$\mathop{\mathbf{E}}_{X,Y}[f(X,Y)] = \mathop{\mathbf{E}}_{X}\Big[\mathop{\mathbf{E}}_{Y}[f(X,Y) \mid X]\Big].$$

(See also exercise 25.6.)

A very nice trick, called the *law of iterated expectations*, is useful in the following situation. Suppose we had a random variable X for which we want to evaluate $\mathbf{E}[X]$. Suppose it is hard to analyze $\mathbf{E}[X]$ directly, but for some contextual reason there is a second random variable Y for which the conditional expectation $\mathbf{E}[X \mid Y]$ is better understood. Then $\mathbf{E}[X]$ might be computed indirectly via

$$\mathop{\mathbf{E}}_{X}[X] = \mathop{\mathbf{E}}_{X,Y}[X] = \mathop{\mathbf{E}}_{Y}\Big[\mathop{\mathbf{E}}_{X}[X \mid Y]\Big].$$

For a simple example (found on the internet), suppose we wanted to estimate the probability that it rains tomorrow. Thus let $X = 1$ if it rains tomorrow and 0 if not; $\mathbf{E}[X]$ is the probability that it rains. Suppose we also know the probability that it will rain today, as well as:

1. the probability of it raining tomorrow if it rains today, and
2. the probability of it raining tomorrow if it does not rain today.

Let $Y = 1$ if it rains today and 0 if not; in terms of X and Y, we are assuming that know $\mathbf{E}[Y] = \mathbf{P}[Y = 1]$, $\mathbf{E}[X \mid Y = 1]$, and $\mathbf{E}[X \mid Y = 0]$. Then we can obtain the probability of it raining tomorrow, $\mathbf{E}[X]$, via the law of iteration expectations, as

$$\mathbf{E}[X] = \mathop{\mathbf{E}}_{Y}\left[\mathop{\mathbf{E}}_{X}[X \mid Y]\right] = \mathbf{P}[Y = 1]\,\mathbf{E}[X \mid Y = 1] + \mathbf{P}[Y = 0]\,\mathbf{E}[X \mid Y = 0].$$

Another nice example appears soon in Section 25.4.

25.3 Randomized approximations for SAT

Let us test our new tools on the randomized algorithm for 3-SAT. Recall the very simple algorithm (Fig. 25.1) we want to analyze: given a 3-SAT formula $f(x_1, \ldots, x_n)$, independently flip a fair coin for each x_i and assign $x_i \in \{\mathtt{t}, \mathtt{f}\}$ accordingly. We claim that this algorithm gives a (7/8)th approximation (in expectation).

Consider a single clause; e.g., $(x_1 \vee \bar{x}_2 \vee x_3)$. Of all 8 ways to assign the three variables $(x_1, x_2, x_3$ in the example) values in $\{\mathtt{t}, \mathtt{f}\}$, there is only one way that does *not* satisfy the clause. That is,

each clause is satisfied with probability $7/8$.

Simple enough. However we are interested not in the outcome of a single clause, but the total number of clauses that are satisfied. That is to say we are analyzing many clauses and not just one. As a warm up, suppose we had two clauses, say C_1 and C_2. Let E_1 (resp. E_2) denote the event that C_1 (resp. C_2) is satisfied. Here the analysis varies depending on whether C_1 and C_2 share variables.

In the simplest case, suppose C_1 and C_2 have no variables in common. Then C_1 and C_2 depend on completely different coin tosses, so the events E_1 and E_2 are independent. Thus, the probability that they are both satisfied is

$$\mathbf{P}[E_1, E_2] = \mathbf{P}[E_1]\,\mathbf{P}[E_2] = \left(\frac{7}{8}\right)^2.$$

Likewise one can obtain probabilities that exactly one clause, or neither clause, is satisfied, arriving at the following table of joint probabilities.

$\mathbf{P}[\cdot, \cdot]$	E_2	$\bar{E}_2$
E_1	49/64	7/64
$\bar{E}_1$	7/64	1/64

By direct calculation[2] one will find that 7/4 clauses are satisfied (out of a maximum of 2) on average.

The calculations above were clean insofar as C_1 and C_2 are disjoint. What if they shared variables? Suppose for example they both had a variable x_3 in common; say,

$$C_1 = (x_1 \vee \bar{x}_2 \vee x_3) \text{ and } C_2 = (x_3 \vee x_4 \vee \bar{x}_5).$$

Now x_3 has greater importance: if $x_3 = \mathtt{t}$, then both C_1 and C_2 are immediately satisfied; if tails, then both C_1 and C_2, independently, need one of two coins to come up in their favor to be satisfied.

It is helpful in this example to map out the scenarios depending on the outcome of x_3, via *conditional expectations*. (We can *imagine* x_3 is flipped first, even if it is actually not.) For example, for $E_1 \wedge E_2$, we have

$$\mathbf{P}[E_1, E_2] = \mathbf{P}[E_1, E_2 \,|\, x_3 = \mathtt{t}]\, \mathbf{P}[x_3 = \mathtt{t}] + \mathbf{P}[E_1, E_2 \,|\, x_3 = \mathtt{f}]\, \mathbf{P}[x_3 = \mathtt{f}].$$

Now, we have $\mathbf{P}[x_3 = \mathtt{t}] = \mathbf{P}[x_3 = \mathtt{f}] = 1/2$. If $x_3 = \mathtt{t}$, then $E_1 \wedge E_2$ always occurs. Now consider the case where $x_3 = \mathtt{f}$. Since C_1 and C_2 have no overlap besides x_3, the events E_1 and E_2, conditional on $x_3 = \mathtt{f}$, are now independent. Thus

$$\mathbf{P}[E_1, E_2 \,|\, x_3 = \mathtt{f}] = \mathbf{P}[E_1 \,|\, x_3 = \mathtt{f}]\, \mathbf{P}[E_2 \,|\, x_3 = \mathtt{f}] = \left(\frac{3}{4}\right)^2.$$

Altogether we obtain

$$\mathbf{P}[E_1, E_2] = 1 \cdot \frac{1}{2} + \left(\frac{3}{4}\right)^2 \cdot \frac{1}{2} = 25/36.$$

Likewise, one can compute remaining values $\mathbf{P}\left[E_1 \wedge \bar{E}_2\right]$, $\mathbf{P}\left[\bar{E}_1 \wedge \bar{E}_2\right]$, and $\mathbf{P}\left[\bar{E}_1 \wedge \bar{E}_2\right]$ from the 2×2 table. Then by direct calculation one obtains the expected number of clauses that are satisfied in this scenario. If the reader works out the calculations they will find that the expected number is (again) 7/4.

Besides the scenarios above, we can also have the cases where C_1 and C_2 overlap at 2 variables, or all 3. We also need to consider scenarios where a variable x appears in C_1 while its negation $\bar{x}$ appears in C_2. Each of these situations require another series of calculations. The point is that its getting messier and messier, even with just two clauses.

Now imagine trying to analyze three clauses C_1, C_2, C_3. The number of ways the three clauses can relate grows combinatorially. In general, over m clauses $C_1, \ldots, C_m$,

[2] $2 \cdot (49/64) + 7/64 + 7/64 = 7/4$.

there are just too many possible scenarios to calculate everything exactly. The more general question is: *how do we precisely analyze a randomized mechanism that encompasses a combinatorial explosion of possibilities?*

The key insight is that we are only interested in the *average* number of clauses satisfied. Observe that the average, being a reductive aggregate statistic, does not reveal much information about the fine-grained complexities about the clauses. Perhaps all these details are not necessary to assess the average either.

For each clause C_i, let $Y_i \in \{0, 1\}$ be the random variable indicating whether or not C_i is satisfied:

$$Y_i = \begin{cases} 1 & \text{if } C_i \text{ is satisfied,} \\ 0 & \text{otherwise.} \end{cases}$$

Thus $\sum_{i=1}^m Y_i$ is the number of clauses satisfied, and we seek the quantity $\mathbf{E}[\sum_{i=1}^m Y_i]$.

As established above, it is easy to see that

$$\mathbf{E}[Y_i] = 7/8$$

for each i. The difficulty is in understanding how to analyze the sum of the Y_i's *jointly* as they may be strongly connected with each other. Thank goodness, then, for *linearity of expectation: the expected sum equals the sum of expectations.* By linearity of expectation, we have

$$\mathbf{E}\left[\sum_{i=1}^m Y_i\right] = \sum_{i=1}^m \mathbf{E}[Y_i] = \frac{7}{8}m.$$

In the final analysis we pay no attention to the intricate relationships among the C_i's. Such is the power of linearity of expectations, to help us see in the aggregate what is far too complicated to understand in detail.

It is easy to extend the above to k-SAT for any $k \in \mathbb{N}$, where each clause has exactly k variables, and obtain the following.

Theorem 25.7. *For all $k \in \mathbb{N}$, there is a randomized $(1 - 1/2^k)$-approximation algorithm for k-SAT.*

Derandomization. Its hard to imagine a more random algorithm than flipping a coin for each variable. Can we obtain a (7/8)-approximation factor *deterministically*? The answer is yes, and perhaps more surprisingly, we will use the randomized algorithm as a guide.

As before, for each clause C_i, let Y_i indicate whether or not a clause is satisfied. Let $Z = \sum_{i=1}^{m} Y_i$. When all the variables are assigned uniformly at random, $\mathbf{E}[Z] = (7/8)m$.

Among many other decisions, a deterministic algorithm must decide whether to set $x_1 = \mathtt{t}$ or $x_1 = \mathtt{f}$. We already know that a uniformly random choice is pretty good on average. Generally speaking, if an average of choices is good, then at least one of those choices ought to be good as well. We just need a way to distinguish the better choice.

Let us be more precise. Suppose all the x_i's are assigned values independently and uniformly at random. By conditional expectations we have

$$\mathbf{E}[Z] = \frac{1}{2}\mathbf{E}[Z \mid x_1 = \mathtt{t}] + \frac{1}{2}\mathbf{E}[Z \mid x_1 = \mathtt{f}].$$

Rearranging, we have

$$\max\{\mathbf{E}[Z \mid x_1 = \mathtt{t}], \mathbf{E}[Z \mid x_1 = \mathtt{f}]\} \geq \frac{1}{2}(\mathbf{E}[Z \mid x_1 = \mathtt{t}] + \mathbf{E}[Z \mid x_1 = \mathtt{f}]) \geq \mathbf{E}[Z].$$

That is, fixing either $x_1 = \mathtt{t}$ or $x_1 = \mathtt{f}$, and then randomly flipping the coins for the remaining variables, will preserve the expected number of satisfied clauses.

To decide which choice is better, we need to be able to compute the conditional expectations $\mathbf{E}[Z \mid x_1 = \mathtt{t}]$ and $\mathbf{E}[Z \mid x_1 = \mathtt{f}]$. Fortunately this is easy to do. Consider the case $x_1 = \mathtt{t}$. By linearity of expectation we have

$$\mathbf{E}[Z \mid x_1 = \mathtt{t}] = \sum_{i=1}^{m} \mathbf{E}[Y_i \mid x_1 = \mathtt{t}].$$

For a particular Y_i, we have

1. $\mathbf{E}[Y_i \mid x_1 = \mathtt{t}] = 1$ if x_i appears (unnegated) in the ith clause, and
2. $\mathbf{E}[Y_i \mid x_1 = \mathtt{t}] = 1 - 2^{-k_i}$ otherwise, where k_i is the number of undecided variables remaining.[3]

Thus we can calculate $\mathbf{E}[Z \mid x_1 = \mathtt{t}]$ and similarly $\mathbf{E}[Z \mid x_1 = \mathtt{f}]$ exactly, and identify the choice of x_1 maximizing the conditional expectation.

Fixing x_1 to be this value from now on, we can now identify the best choice for x_2 in the same way, and continue in such a fashion to make deterministic choices for all x_i's.

[3]For 3-SAT, $k_i = 2$ if $\bar{x}_i \in C_i$, and $k_i = 3$ otherwise.

To make this precise, suppose we have already identified values $a_1, \ldots, a_k \in \{\mathtt{t}, \mathtt{f}\}$ for $x_1, \ldots, x_k$, respectively, such that

$$\mathbf{E}[Z \mid x_1 = a_1, \ldots, x_k = a_k] \geq (7/8)m.$$

Consider x_{k+1}. We have

$$\begin{aligned}\mathbf{E}[Z \mid x_1 = a_1, \ldots, x_k = a_k] = \frac{1}{2}\mathbf{E}[Z \mid x_1 = a_1, \ldots, x_k = a_k, x_{k+1} = \mathtt{t}] \\ + \frac{1}{2}\mathbf{E}[Z \mid x_1 = a_1, \ldots, x_k = a_k, x_{k+1} = \mathtt{f}],\end{aligned}$$

hence setting either $x_{k+1} = \mathtt{t}$ or $x_{k+1} = \mathtt{f}$ preserves the expected value of Z. We can compute both conditional expectations explicitly, and identify the better choice, setting this value to a_{k+1} accordingly. Continuing in this fashion, we will eventually identify n values $a_1, \ldots, a_n$ such that

$$f(a_1, \ldots, a_n) = \mathbf{E}[Z \mid x_1 = a_1, \ldots, x_n = a_n] \geq (7/8)m,$$

as desired.

The algorithm we have described is entirely deterministic. It uses the estimates of imagined randomized experiments to make its deterministic choices, and the estimates can calculated deterministically thanks in part to linearity of expectation.

25.4 Randomized sorting

We now move on to sorting. Recall the simple `quick-sort` algorithm from the introduction, and given in Fig. 25.2. To recap, `quick-sort` is a recursive algorithm where, for each subproblem, we select one of the elements uniformly at random as a pivot. We divide the remaining elements into those that are smaller and greater than the pivot, and recursively sort both groups.

Intuitively, we are hoping that each pivot roughly divides the input in half. If that were always the case, then we would have a $O(n \log n)$ running time by the usual divide-and-conquer analysis. However the random pivot may be bad and break the input into extremely uneven parts, in which case we have made little progress from a divide-and-conquer point of view.

It appears difficult to analyze the running time when progress varies wildly on random choices. It gets more convoluted when one thinks about how a good or bad pivot early effects all the running times thereafter. More generally, one difficulty in analyzing a randomized *algorithm* is that sequences of random decisions generate overwhelmingly many "butterfly effects" to consider.

Fortunately we are not trying to map out all the probabilistic outcomes with complete precision. We are only interested in analyzing the running time on *average*. We will leverage linearity of expectation to greatly simplify the analysis for the following theorem.

Theorem 25.8. *Given a list of n comparable elements,* `quick-sort` *returns elements in a sorted list in $O(n \log n)$ expected time.*

Proof. For each $i, j \in [n]$ with $i < j$, let X_{ij} be equal to 1 if the rank i element (i.e., the ith smallest element) is compared to the rank j element, and 0 otherwise. $\sum_{i<j} X_{ij}$ represents the total number of comparisons made by the algorithm, hence the running time up to a constant factor. We want to upper-bound $\mathbf{E}\left[\sum_{i<j} X_{ij}\right]$.

Consider X_{ij} for fixed $i < j$. Observe that the rank i and rank j numbers are compared to each other iff either is selected as the pivot before any element of rank between i and j. Since the pivots are selected uniformly at random, this occurs with probability $2/(j - i + 1)$. That is,

$$\mathbf{E}[X_{ij}] = \mathbf{P}[X_{ij} = 1] = \frac{2}{j - i + 1}.$$

Consider now the sum $\sum_{i<j} X_{ij}$. While each X_{ij} was simple to analyze alone, the different X_{ij}'s are not at all independent. Fortunately we do not need to map out their myriad interactions; we are only interested in the X_{ij}'s *in the aggregate* and *on average*. Enter *linearity of expectation*. We have

$$\mathbf{E}\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}\right] \overset{\text{(a)}}{=} \sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] \overset{\text{(b)}}{=} \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n}\sum_{k=1}^{n-i} \frac{2}{k} \leq O(n \log n),$$

as desired. Here (a) applies linearity of expectation: the average sum is equal to the sum of averages. (b) is from our analysis for a single X_{ij} above. □

Bounding the running time with high probability. We've now shown that `quick-sort` terminates in $O(n \log n)$ time *on average*. Next we will show that the running time is at most $O(n \log n)$ with *high probability*. Here, "high probability" means that the probability of error is at most $1/n^c$ for some fixed constant $c > 0$; that is, polynomially small in the input size. (Below we will prove a probability of error of $1/n^2$, but the exponent could have been made arbitrarily large by increasing the hidden constant in the $O(n \log n)$ running time.)

Our proof will require the following lemma.

Lemma 25.9 (Markov's inequality)**.** *Let $X \geq 0$ be a* nonnegative *random variable, and $\alpha \geq 0$. Then*

$$\mathbf{P}[X \geq \alpha] \leq \mathbf{E}[X]/\alpha.$$

Proof. Since X is nonnegative we have

$$\mathbf{E}[X] = \mathbf{E}[X \mid X \geq \alpha]\,\mathbf{P}[X \geq \alpha] + \mathbf{E}[X \mid X < \alpha]\,\mathbf{P}[X < \alpha] \geq \alpha\,\mathbf{P}[X \geq \alpha]$$

for all $\alpha \geq 0$. □

Markov's inequality is very intuitive. Consider for example $\alpha = 2\,\mathbf{E}[X]$. Then Markov's inequality states that for nonnegative X, the probability that X is at least twice its average is at most 50%. This should be as obvious as the fact that no more than half the population can be twice as wealthy as the average individual. Or that no more than half the class can get at least twice the average score on the midterm, no matter how low the average. Etc.

We now present the high probability bound for `quick-sort`.

Theorem 25.10. `quick-sort` *runs in $O(n \log n)$ time with high probability.*

Proof. Fix an element e. This element e will appear in a series of recursive subproblems until e is selected as a pivot. Let the *depth* of e, denoted D_e, be the number of recursive subproblems containing e before reaching the base case. Observe that the sum of depths over all the elements, $\sum_e D_e$, bounds the number of comparisons made by the algorithm.

Fix e. We will show that with high probability, D_e is at most $O(\log n)$. More precisely, we will prove that

$$\mathbf{P}[D_e \geq 32 \ln n] \leq 1/n^3. \tag{25.6}$$

Assuming (25.6) holds for any e, we then have

$$\mathbf{P}\left[\max_e D_e \geq 32 \ln n\right] \overset{\text{(a)}}{\leq} \sum_e \mathbf{P}[D_e \geq 32 \ln n] \leq n \cdot \frac{1}{n^3} = \frac{1}{n^2}$$

by (a) the union bound. This establishes that with high probability, *all* elements have depth $O(\log n)$. In this event the running time is $O(\sum_e D_e) = O(n \log n)$, as desired.

It remains to prove (25.6) for a fixed element e. For $i = 0, 1, 2, \ldots$, let X_i be the number of elements in the subproblem containing e after i recursive calls. Here $X_0 = n$ since initially there are n elements. For depths i after e is selected as a pivot we set $X_i = 0$. Thus $D_e \geq i$ only if $X_i \geq 1$.

We want to upper bound $\mathbf{E}[X_i]$ for each i (and eventually show that $\mathbf{E}[X_i] = 1/\operatorname{poly}(n)$ for $i = O(\log n)$). As mentioned above, $X_0 = n$. Consider X_1. An exact estimate for $\mathbf{E}[X_1]$ is somewhat involved as it depends on the rank of e. A lazier upper bound can be obtained as follows.

For a given subproblem of k elements, call a pivot "good" if it is one of the middle $k/2$ elements, and otherwise "bad". A pivot is good with probability $1/2$, and separates e from at least $k/4$ elements. Applying this logic to the first pivot, where $k = X_0 = n$, we have

$$\begin{aligned}\mathbf{E}[X_1] &= \frac{1}{2}\mathbf{E}[X_1 \,|\, \text{first pivot is good}] + \frac{1}{2}\mathbf{E}[X_1 \,|\, \text{first pivot is bad}] \\ &\leq \frac{1}{2}\cdot\frac{3}{4}n + \frac{1}{2}n \leq \frac{7}{8}n.\end{aligned}$$

More generally, for each index i, *conditional on* X_{i-1}, we have

$$\mathbf{E}[X_i \,|\, X_{i-1}] \leq \frac{1}{2}\cdot\frac{3}{4}X_{i-1} + \frac{1}{2}X_{i-1} = \frac{7}{8}X_{i-1}.$$

We now claim, by induction on i, that $\mathbf{E}[X_i] \leq (7/8)^i n$. The base case $i = 0$ is immediate. For $i > 0$, we have

$$\mathop{\mathbf{E}}_{X_i}[X_i] \overset{\text{(b)}}{=} \mathop{\mathbf{E}}_{X_{i-1}}\left[\mathop{\mathbf{E}}_{X_i}[X_i \,|\, X_{i-1}]\right] = \mathop{\mathbf{E}}_{X_{i-1}}\left[\frac{7}{8}X_{i-1}\right] \overset{\text{(c)}}{=} \left(\frac{7}{8}\right)^i n.$$

Here, in (b), we applied the law of iterated expectations. (c) is by our induction hypothesis.

Now, let $k = 32\ln n$. We have

$$\mathbf{E}[X_k] \leq \left(1 - \frac{1}{8}\right)^k n \overset{\text{(d)}}{\leq} e^{-k/8}n = e^{-4\ln n}n = \frac{1}{n^3}.$$

Here, in (d), we applied the inequality $1 + x \leq e^x$ which holds for all x. Finally, by Markov's inequality, we have

$$\mathbf{P}[D_e \geq k] \leq \mathbf{P}[X_k \geq 1] \leq \mathbf{E}[X_k] \leq 1/n^3,$$

as claimed in Eq. (25.6). This completes the proof. □

25.5 Additional notes and materials

Quicksort is also covered in [MR95, Chapter 1]. This particular proof of the high probability bound for `quick-sort` is from [Har19]. It is implicitly similar to a more standard proof using *concentration inequalities*, which will be introduced later.

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:
- Handwritten .pdf prepared before the lecture (and .note file).
- Handwritten .pdf annotated during the presentation (and .note file).
- Recorded video lecture.

Fall 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. We covered the randomized algorithm for 3-SAT (and not derandomization) and `quick-select` (expected running time only), as well as a randomized algorithm for minimum cut which will be discussed later. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

25.6 Exercises

Additional exercises may be found it [MR95, Chapter 1].

Exercise 25.1. Prove or disprove: For any two events A, B,

$$\mathbf{P}[A \wedge B] \leq \min\{\mathbf{P}[A], \mathbf{P}[B]\}.$$

Exercise 25.2. Prove or disprove: For any two events A and B, if $\mathbf{P}[A] + \mathbf{P}[B] > 1$, then $\mathbf{P}[A \wedge B] > 0$.

Exercise 25.3. Let A and B be two events. Prove that the following three identities are all equivalent:

$$\mathbf{P}[A \,|\, B] = \mathbf{P}[A], \qquad \mathbf{P}[B \,|\, A] = \mathbf{P}[B], \qquad \mathbf{P}[A, B] = \mathbf{P}[A]\,\mathbf{P}[B].$$

(That is, if A and B satisfies any one of the identities above, then it automatically satisfies the other two.)

Exercise 25.4. Let A and B two events. Prove or disprove that A and B are independent iff $\bar{A}$ and $\bar{B}$ are independent.

Exercise 25.5. Prove linearity of expectation (Theorem 25.6).

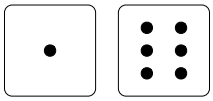
Exercise 25.6. Prove or disprove: for any two random variables X, Y, and real-valued function $f(X, Y)$, we have

$$\mathop{\mathbf{E}}_{X}\Big[\mathop{\mathbf{E}}_{Y}[f(X,Y) \,|\, X]\Big] = \mathop{\mathbf{E}}_{Y}\Big[\mathop{\mathbf{E}}_{X}[f(X,Y) \,|\, Y]\Big].$$

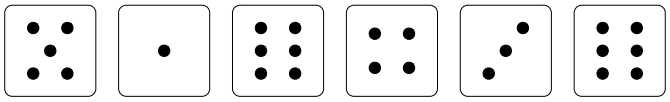
Exercise 25.7. Let A and B be two events with $\mathbf{P}[A] + \mathbf{P}[B] = 1$. Prove or disprove: $\mathbf{P}[A \vee B] = 1$ iff $\mathbf{P}[A \wedge B] = 0$.

Exercise 25.8. Suppose you only have access to a coin that flips heads with a known probability p, and tails with (remaining) probability $1 - p$. Describe and analyze a protocol that uses a limited number of tosses of this biased coin in expectation (the smaller the better) to simulate 1 coin toss of a fair coin. (The expected number of biased coin tosses you make may depend on p.)

Exercise 25.9. Recall that when we roll six-sided dice, the dice samples an integer between 1 and 6 uniformly at random. Let us call an unordered pair of dice "lucky" if one of them is a 1 and the other is a 6.



If we roll 6 independent six-sided dice, how many lucky pairs do we expect? Note that a single dice may appear in more than one lucky pair. For example, the following roll of six dice has 2 lucky pairs amongst them.



Exercise 25.10. Suppose you repeatedly flip a coin that is heads with fixed probability $p \in (0, 1)$.

1. What is the expected number of coin flips until you obtain one heads?[4] Prove your answer.
2. What is the expected number of coin flips until you obtain two heads? Prove your answer.
3. For general $k \in \mathbb{N}$, what is the expected number of coin tosses until you obtain k heads? Prove your answer.

Exercise 25.11. This exercise is about how for many intents and purposes, we approximately have the extremely convenient identity, "$1 + x \approx e^x$".

1. Prove that for all $x \in \mathbb{R}$, $1 + x \leq e^x$.

   Hint: At $x = 0$, both sides are equal. What are their respective rates of change moving away from 0?
2. Prove that for all $x \leq 1$, $e^x \leq 1 + x + x^2$.

[4] If the first toss is heads, that counts as one coin flip. If the first toss is tails and the second toss is heads, that counts as two coin tosses. Etc. It may be helpful to first think about a fair coin, where $p = 1/2$.

Exercise 25.12. Recall the `quick-select` algorithm introduced in Section 25.1.3. The goal of this exercise is to prove that `quick-select` takes $O(n)$ time in expectation. Below we present two different approaches which offer two different perspectives. We ask you to submit solutions to just one of the two approaches, though we strongly recommending trying to solve and understanding both. Both analyses should use linearity of expectation and we ask you to point this out explicitly.

1. *Approach 1.* Analyze `quick-select` similarly to `quick-sort`, based on the sum of indicators X_{ij}.

   One approach is to reduce to a separate analysis for each of the following 4 classes of pairs:

   (a) X_{ij} where $i < j < k$,

   (b) X_{ij} where $i < k < j$,

   (c) X_{ij} where $k < i < j$, and

   (d) X_{ij} where either $i = k$ or $j = k$.

   For each case, show that the expected sum is $O(n)$.

2. *Approach 2.* The following approach can be interpreted as a randomized divide and conquer argument. We are arguing that with constant probability, we decrease the input by a constant factor, from which the fast (expected) running time follows.

   (a) Consider a coin that flips heads with probability p. Suppose we repeatedly flip the coin until it comes up heads. What is the expected number of coin tosses until it comes up heads? Prove your answer.[5]

   (b) Consider again `quick-select`. Consider a single iteration where we pick a pivot uniformly at random and throw out some elements. Prove that with some constant probability p, we either sample the kth element or throw out at least $1/4$ of the remaining elements.

   (c) For each integer i, prove that the expected number of iterations (i.e., rounds of choosing a pivot) of `quick-select`, where the number of elements remaining is in the range $[(4/3)^i, (4/3)^{i+1})$, is $O(1)$.

   (d) Fix an integer i, and consider the amount of time spent by `quick-select` while the number of elements remaining is greater than $(4/3)^{i-1}$ and at most $(4/3)^i$. Show that that the expected amount of time is $\leq O((4/3)^i)$

[5] There is a slick proof.

(e) Finally, use the preceding part to show that the expected running time of `quick-select` is $O(n)$.

Exercise 25.13. Last week you decided to clean up your toolshed, but you compulsively made the following silly mistake. Originally you had n matching pairs of nuts and bolts of different sizes, but you decided to organize the nuts and bolts separately into a box full of nuts and a box full of bolts, splitting up all the pairs of nuts and bolts in the process. So now you have n nuts, and n bolts, such that each nut fits a distinct bolt and vice-versa, but you don't know which nut goes with which bolt. The bolts all look pretty similar, so you can't compare two bolts directly with each other and tell which one is bigger. Likewise you cannot compare the nuts to each other. However, you can try to fit one nut to one bolt, and see if either:



1. The nut fits the bolt.
2. The nut is too big for the bolt.
3. The nut is too small for the bolt.

We will treat one of these nut-to-bolt tests as a single operation. Your goal is to match up all nuts and bolts. Of course you can compare every pair of nut and bolt in $O(n^2)$ time, but can you do better?

Design and analyze an algorithm, as fast as possible, to recover all matching pairs of nuts and bolts.

Exercise 25.14. This exercise is about a simple randomized algorithm for *verifying* matrix multiplication. Suppose we have three $n \times n$ matrices A, B, C. We want to verify if $AB = C$. Of course one could compute the product AB and compare it entrywise to C. But multiplying matrices is slow: the straightforward approach takes $O(n^3)$ time and there are (more theoretical) algorithms with running time roughly $O(n^{2.37\cdots})$. We want to test if $AB = C$ in closer to n^2 time.

The algorithm we analyze is very simple. Select a point $x \in \{0,1\}^n$ uniformly at random. (That is, each $x_i \in \{0,1\}$ is an independently sampled bit.) Compute $A(Bx)$ and Cx, and compare their entries. (Note that it is much faster to compute $A(Bx)$ than AB.) If they are unequal, then certainly $AB \neq C$ and we output `false`. Otherwise we output `true`. Note that the algorithm is always correct if $AB = C$, but could be wrong when $AB \neq C$. We will show that if $AB \neq C$, the algorithm is correct with probability at last $1/2$.

1. Let $y \in \mathbb{R}^n$ be a fixed nonzero vector, and let $x \in \{0,1\}^n$ be drawn uniformly at random. Show that $\langle x, y \rangle \stackrel{\text{def}}{=} \sum_{i=1}^n x_i y_i \neq 0$ with probability at least $1/2$. [6]

2. Use the preceding result to show that if $AB \neq C$, then with probability at least $1/2$, $ABx \neq Cx$.[7]

3. Suppose we want to decrease our probability of error to (say) $1/n^2$. Based on the algorithm above, design and analyze a fast randomized algorithm with the following guarantees.

   - If $AB = C$, then it always reports that $AB = C$.
   - If $AB \neq C$, then with probability of error at most $1/n^2$, it reports that $AB \neq C$.

   (Your analysis should include the running time as well.)[8]

Exercise 25.15. Let P be a set of n distinct points in the plane. Consider the problem of computing the minimum distance $\|p - q\|$ between any two distinct points $p, q \in P$. (If $|P| \leq 1$, the minimum distance is defined as $+\infty$.) For simplicity we assume the points are in general position, so that all pairwise distances are unique. You may have learned a deterministic divide-and-conquer algorithm that runs in $O(n \log n)$ time. Here we will analyze the running time of a different randomized algorithm. To help build intuition, first solve the following problem.



1. (3 points) Let $a_1, \ldots, a_n \in \mathbb{R}$ be n distinct numbers presented in a uniformly random order. Imagine tracking the minimum as you examine the numbers one by one in (the randomized) order. What is the expected number of times the minimum changes?

We assume black box access to a data structure for the following "threshold" version of the minimum distance problem. Fix a threshold $\rho > 0$. The data structure takes $O(1)$ time to initialize, and starts with an empty point set. It supports the following operation:

[6] *Hint: Suppose for simplicity that the last coordinate of y is nonzero. It might help to imagine sampling the first $n-1$ bits and computing the partial sum $S_{n-1} = \sum_{i=1}^{n-1} x_i y_i$ first, before sampling x_n and adding $x_n y_n$. Formally your analysis may involve some conditional probabilities. (And what about the case where $y_n = 0$?)*

[7] Even if you haven't solved part 1 you may assume it to be true.

[8] *Hint:* How can you decrease the probability of error to 1/4? 1/8? 1/16?

`insert(`x`)`: Insert a point $x \in \mathbb{R}^2$ into the underlying point set. If the minimum distance in the underlying point set is (strictly) less than ρ, then return `true`; otherwise return `false`. This operation takes $O(1)$ expected time.[9]

This subroutine, combined with part 1, suggests the following algorithm. Here we assume that one can do math operations (add, subtract, square, etc.) in $O(1)$ time, and that we can sample a random permutation of n items in $O(n)$ time.

1. Let $p_1, p_2, \ldots, p_n$ permute P uniformly at random. *// $O(n)$ time.*
2. Let $r_1 = +\infty$ and $r_2 = \|p_1 - p_2\|$.

/ We maintain the invariant that r_i is the minimum distance over $p_1, \ldots, p_i$. */*

3. Start a new instance of the threshold data structure with threshold r_2. Insert p_1 and p_2.
4. For $i = 3, \ldots, n$:
   A. Insert p_i into the threshold data structure. If the data structure returns true:
      1. Let $r_i = \min\{\|p_i - p_j\| : j < i\}$.
      2. Replace the threshold data structure with a new one with radius r_i. Insert $p_1, \ldots, p_i$ into the data structure.
   B. Otherwise set $r_i = r_{i-1}$.
5. Return r_n.

Now analyze this algorithm via the following steps.

2. (3 points) For each index $i > 2$, analyze the exact probability that $r_i < r_{i-1}$.
3. (4 points) Finally, bound the expected running time of the algorithm.

Exercise 25.16. You have a sequence of n switches $S_1, \ldots, S_n$ that jointly control m light bulbs $L_1, \ldots, L_m$. Each switch can be "up" or "down", and this controls whether the light bulbs are on or off.

[9] We can implement this data structure by building a "hash table over grid cells". Initially we have an empty dictionary for an empty point set. Given a point $x = (x_1, x_2)$, compute the key `key(`x`)` $= (\lfloor x_1/(\rho/\sqrt{2}) \rfloor, \lfloor x_2/(\rho/\sqrt{2}) \rfloor$ in $O(1)$ time. (Rounding is not a standard operation in all models of computation, but here we assume it takes $O(1)$ time.) If this key is already occupied by a point $y \in P$, then $\|x - y\| \le \rho$, and we return `true`. Otherwise consider the 8 "neighboring" keys/cells where we add or subtract at most one or zero to each of the key's coordinates. Assuming the minimum distance was $\ge \rho$ before inserting x, each of these 8 cells can have at most 1 point. If the distance between x and any of the $O(1)$ points in neighboring cells is $< \rho$ from x, return `true`.

Each light bulb L_i, is associated with two sets of switches $A_i, B_i \subseteq [n]$. The switches in A_i turn on the light bulb when they are "up" and the switches in B_i turn on the light bulb then they are "down".

More precisely, for each $j \in A_i$, having switch S_j "up" automatically turns on the light bulb. (It only takes one of these switches to be "up" to turn on the light bulb.) For each $j \in B_j$, turning the switch "down" automatically turns on the light bulb. (Again, it only takes one of these switches to be "down" to turn on the light bulb.)

Thus, for a light bulb L_i, the light bulb L_i lights up if and only if either (a) some switch in A_i is flipped up or (b) some switch in B_i is flipped down. A_i and B_i are generic subsets of switches, not necessarily disjoint, and their union does not necessarily include all the switches. We do assume, however, that $|A_i| + |B_i| \geq 2$ for all i. We assume that the sets A_i and B_i are given explicitly for each i (for simplicity; otherwise they can be obtained by inspection).

Your algorithm can flip switches "up" and "down". For the sake of running times, assume that flipping a single switch takes $O(1)$ time, and inspecting whether a single light bulb is on or off takes $O(1)$ time. The light bulbs turn on and off instantly when you flip a switch.

For each of the following decision problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. Decide if there exists a way to flip the switches to turn on all the light bulbs.

2. Decide if there exists a way to flip the switches to turn on at least three-fourths of the light bulbs.

Exercise 25.17. Let $G = (V, E)$ be an undirected graph with m edges and n vertices, vertex weights $w : V \to \mathbb{R}_{>0}$, and maximum (unweighted) vertex degree Δ. We let $W = \sum_{v \in V} w(v)$ denote the total weight of all the vertices.

Recall than an *independent set* is a set of vertices $S \subseteq V$ such that no two vertices in S are connected by an edge. Computing the maximum cardinality independent set is NP-Hard. In fact, for all $\epsilon > 0$, it is NP-Hard to get a $1/n^{1-\epsilon}$-approximation to the maximum cardinality independent set. (Of course, computing the maximum *weight* independent set is no easier.)

Consider the following algorithm that always returns a *maximal* independent set:

`randomized-greedy`$(G = (V, E), w)$

1. Order the vertices $v_1, \ldots, v_n$ uniformly at random.
2. $S \leftarrow \emptyset$.
3. For $i = 1, \ldots, n$:
   A. If $S + v_i$ is independent, then set $S \leftarrow S + v_i$.
4. Return S.

We will analyze this algorithm by proving two claims. For both proofs, explain the reasoning for each step clearly.

1. (5 points) Consider a vertex $v \in V$ with degree d_v. Prove that $v \in S$ with probability $1/(d_v + 1)$.
2. (5 points) Prove that the expected weight of the returned independent set S is at least $W/(\Delta + 1)$. (Here you can assume part 1 as a fact even if you can't prove it.)

Exercise 25.18. Prove Turán's theorem:

> **Turan's theorem** *Any undirected graph G with m edges and n vertices has an independent set of size (at least)*
>
> $$\frac{n^2}{2m + n}.$$

(You may want to solve Exercise 25.17 first.)

Exercise 25.19. There are a number of implementations of binary search trees offering $O(n)$ space and $O(\log n)$ (possibly amortized) time for insertions, deletions, and search queries. Here we develop a simple, randomized data structure, called *randomized treaps*, that offers the same performance guarantees in expectation.

A treap is a binary search tree over a set of n comparable items that each have an additional scalar *priority*. We define it recursively. The root is always the minimum priority item. The left subtree is a treap over all the items whose key (not priority) is less than that of the root. The right subtree is a treap over all the items whose key (again, not priority) is greater than that of the root. For simplicity we assume all keys and priorities are distinct. Note that each item has a unique place in the tree.

A randomized treap is a treap over n comparable items where each item x_i is independently assigned a priority p_i drawn uniformly from the range $[0, 1]$.

- To search for an item, we just search the binary tree as usual.
- To insert a new item, we first search for and verify that the item is not in the tree. Then we sample a new priority for the item, identify the unique place for the new item in the treap, and rebuild the subtreap rooted by the new item. (You may assume that one can do this in linear time in the number of items in the subtree.)
- To delete a new item, we search for it in the tree, and rebuild the subtree rooted by the node. (Again, in linear time in the number of nodes in the subtree.)

Analyze the search operation and prove that it takes $O(\log n)$ time in expectation. (Insertion and deletion can be analyzed somewhat similarly; in the interest of time, we are only asking you to analyze search. Your analysis should be self-contained and from first principles; we will not award many points for saying "It's just like X" for some topic X that this may remind you of.)

Exercise 25.20. Let $G = (V, E)$ be an undirected graph. Recall that a set of vertices $S \subseteq V$ is *independent* if no pair of vertices in S is connected by an edge. An independent set is *maximal* if $S + v$ is not independent for any $v \in V \setminus S$; equivalently, S is a dominating set.

It is easy to create a maximal independent set greedily; starting from $S = \emptyset$, go through all the vertices one by one, adding any vertex that retains independence. This problem is about computing a maximal independent set in *parallel* in very few rounds of parallel computation. Maximal independent set is a very valuable primitive in parallel algorithms.

Consider the following algorithm:

1. $I \leftarrow \emptyset$
2. While G is not empty:
   - A. In parallel, remove all isolated vertices and add them to I.
   - B. In parallel, mark each vertex v independently with probability $1/2 \deg(v)$, where $\deg(v)$ is the degree of v in the current (remaining) graph.
   - C. For every edge $e = (u, v)$ in parallel:
     1. If u and v are both marked, unmark the lower degree vertex, breaking ties arbitrarily.
   - D. Add all marked vertices to I. Delete all marked vertices, all neighbors of marked vertices, and all incident edges from G.

3. Return I.

Our goal is to prove that this algorithm outputs a maximal independent set in $O(\log m)$ rounds in expectation. It is easy to see that the output is a maximal independent set: each round, the "unmarking" step ensures that all the marked vertices are independent. When we add the marked vertices to I, we remove the marked vertices and all its neighbors from G to ensure that any future vertices added to I are independent.

We have broken down the proof into five steps. Note that even if you have a tough time proving one of the steps, you can assume it to be true in subsequent steps.

The proof is a "heavy/light" argument. Call a vertex v *heavy* if at least one third of v's neighbors have degree less than or equal to $\deg(v)$, and *light* otherwise. Call an edge $\{u, v\}$ *heavy* if one of its endpoints is heavy, and *light* otherwise.

1. (2 points) Prove that every initially marked vertex remains marked with constant probability.

2. (2 points) Prove that any heavy vertex v is removed from G with constant probability.

3. (2 points) Prove that a constant fraction of edges are heavy. *(Hint: suppose you assign each edge to its higher-degree endpoint...)*

4. (2 points) Prove that each iteration removes a constant fraction of the edges in expectation.

5. (2 points) Finally, prove that the algorithm outputs a maximal independent set after $O(\log m)$ rounds of parallel computation (i.e., in $O(\log m)$ time in the PRAM model) in expectation.

Exercise 25.21. Consider the problem of computing the longest path (with distinct vertices) in a graph. Of course the problem is NP-Hard and we cannot expect a polynomial time algorithm. Meanwhile one can find a path of length k in $O\left(n^{k-1}m\right)$ time in a relatively straightforward fashion, which is polynomial time only for constant k. One can also find the longest path in a DAG (directed acyclic graph) with dynamic programming (which you can assume as a black box).

Our goal is to develop faster algorithms that allow for (slightly) larger values of k. Consider first the randomized algorithm that repeats the following randomized experiment.

1. Order the vertices uniformly at random, and let H be the DAG retaining only the arcs going "down" from a higher ranked vertices to lower ranked vertices.
2. Compute the longest path in H (which is a DAG).

We repeat the experiment above $O(n)$ and return the longest path found over all the randomized sub-DAG's H.

1. (4 points) Show that for $k \leq O(\log(n)/\log\log n)$, this algorithm will find a path of length k (if it exists) with constant probability.

Now we develop a second randomized algorithm that can handle larger values of k.

2. (2 points) Suppose each vertex in G is assigned one of k colors. We say that a path is a *rainbow* path if it has at most one vertex of each color. Write (just) the recursive spec of a dynamic programming algorithm computing the longest rainbow path in $\text{poly}\left(m, n, 2^k\right)$ time.

   (The recursive spec should be strong/complete enough to support a correct proof by induction.)

3. (4 points) Assume you have a subroutine computing a longest rainbow path in $O\left(2^\ell m\right)$ time in a graph with ℓ colors. Design and analyze a randomized polynomial time algorithm that computes a path of length k (if one exists) for $k = O(\log n)$. (I.e., an algorithm whose dependence on k is on the order of $2^{O(k)}$.)

Exercise 25.22. By now we have studied many algorithms that can find a path from s to t, and some variants of (s,t)-path problems that are NP-hard. But what if there's earthquakes?

Let $G = (V, E)$ be a directed graph and $s, t \in V$. We model earthquakes by assuming that each edge $e \in E$ gets destroyed independently with a fixed and known probability. For each edge $e \in E$, let p_e denote the probability of an edge e being destroyed in an earthquake.

We say that an (s,t)-walk *survives* an earthquake if all of its edges survive. The high-level goal is to compute the (s,t)-walk that is most likely to survive the earthquake.

1. (2 points) For an (s,t)-path with (distinct) edges $e_1, \ldots, e_k$, what is the probability that the path survives an earthquake (as a function of the edge-wise probabilities p_e)?

2. (8 points) Consider the problem of computing the (s, t)-walk with the highest probability of surviving an earthquake. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Chapter 26

Randomized Rounding

Many of the problems we are interested in are inherently discrete, and unfortunately many discrete problems are NP-Hard. One general class of NP-hard problems are *integer programs* (IPs), which are linear programs where the variables are required to be integers.

One example is a *covering integer program*, which is an optimization problem of the form

$$\begin{aligned} \text{minimize } & \sum_{j=1}^{n} c_j x_j \\ \text{over } & x \in \mathbb{Z}_{\geq 0} \\ \text{s.t. } & \sum_{j=1}^{n} A_{ij} x_j \geq b_i \text{ for } i \in [m], \end{aligned} \tag{26.1}$$

where $A \in \mathbb{R}_{\geq 0}^{m \times n}$, $b \in \mathbb{R}_{>0}^{m}$, and $c \in \mathbb{R}_{>0}^{n}$. Here, each variable $x_j \in \mathbb{Z}_{\geq 0}$ models a binary decision. Each c_j can be interpreted as the *cost* of taking $x_j = 1$. For $i \in [m]$, the constraint

$$\sum_{j=1}^{n} A_{ij} x_j \geq b_i \text{ for } i \in [m]$$

is a *covering constraint*, saying we must set $x_j = 1$ for enough variables x_j so that the sum of A_{ij} over these variables is at least b_i.

In the set cover problem, we have a family of n sets $\mathcal{F} = \{S_1, \ldots, S_n\}$, where each set S_i is a subset of m points $\mathcal{U}$. The goal is to compute the minimum cardinality collection of sets, $S_{i_1}, \ldots, S_{i_k}$ where $i_1, \ldots, i_k \in [n]$, such that

$$S_{i_1} \cup S_{i_2} \cup \cdots \cup S_{i_k} = \mathcal{U}.$$

The set cover problem is NP-Hard.

Set cover is the special case of a CIP where $c = \mathbb{1}$, $b = \mathbb{1}$, and $A \in \{0,1\}^{m \times n}$. The coordinates $j \in [n]$ correspond to sets, and the coordinates $i \in [m]$ correspond to points. Set j covers point i iff $A_{ij} = 1$.

Besides covering integer programs, there are also packing integer programs, which are maximization problems subject to packing (i.e., $\leq$) constraints, of the form

$$\begin{aligned} \text{maximize } & \sum_{j=1}^{n} b_j y_j \\ \text{over } & y \in \mathbb{Z}^n_{\geq 0} \\ \text{s.t. } & \sum_{j=1}^{m} A_{ij} y_j \leq c_j \text{ for } i \in [m] \end{aligned}$$

where $A \in \mathbb{R}^{m \times n}_{\geq 0}$, $b \in \mathbb{R}^n_{>0}$, and $c \in \mathbb{R}^m_{>0}$. Here each b_j can be understood as the "profit" of taking $y_j = 1$, and we want to maximize the total profit. We are constrained by the n packing constraints. The knapsack problem is a special case of the covering integer programs where $m = 1$.

In general, integer programs maximize or minimize a set of integer variables over a linear objective, subject to linear equality and inequality constraints.[1] The basic appeal of integer programs is that they are very flexible for modeling discrete optimization problems. Unfortunately, their broad applicability also makes them NP-Hard.

Integer programs are NP-Hard because the output is required to be discrete. If we allow the variables to vary *continuously* over the reals, we instead have a *linear program* (LP). For example is a linear program for covering problems similar to the covering integer program above. The difference is that each variable x_j is allowed to take any nonnegative value.

$$\begin{aligned} \text{minimize } & \sum_{j=1}^{n} c_j x_j \\ \text{over } & x \in \mathbb{R}^n_{\geq 0} \\ \text{s.t. } & \sum_{j=1}^{n} A_{ij} x_j \geq b_i \text{ for } i \in [m]. \end{aligned}$$

Any solution feasible to the CIP (26.1) is also feasible for the LP above. As such, the LP is called a *relaxation* of the CIP. It implies that the optimum value of the LP is less than or equal to the optimum value for the integer program.

[1] Integer programs may have both packing (i.e., $Ax \leq b$) and covering (i.e., $Ax \geq b$) constraints. "Packing integer programs" and "covering integer programs" refer to the special case that have only packing constraints or only covering constraints, respectively.

Unlike integer programs, linear programs are polynomial time solvable (!). This allows for the following general approach to discrete optimization: given an IP formulation of the problem, instead solve the corresponding LP. The LP provides a *fractional solution* satisfying the same constraints, which we can treat as a clue towards a good integer solution. The goal becomes to convert the fractional solution to an integer solution, while maintaining feasibility and the objective value. There are several strategies to *round* a fractional solution to an integer solution and they are covered in courses on approximation algorithms (e.g., [WS11; Vaz01]). We will study a technique called *randomized rounding*.

In this chapter, we use randomized rounding to obtain approximation algorithms for max SAT (Section 26.1), set cover (Section 26.2), and CIPs (Section 26.3).

26.1 SAT

Recall the max-SAT problem from Chapter 25: given a boolean formula $f(x_1, \dots, x_n)$ in CNF, the goal is to find an assignment of $x_1, \dots, x_n$ to $\{\mathtt{t}, \mathtt{f}\}$ that satisfies the maximum number of clauses. There we showed that a random assignment gives a 7/8-approximation for 3-SAT. Moreover the algorithm can be derandomized, and the approximation factor is best possible unless $P = NP$.

More generally a random assignment gives a $(1 - 2^{-k})$ approximation for k-SAT, for any $k \in \mathbb{N}$. The following table lists the approximation factors for the first few values of k.

k	**oblivious APX**
1	1/2
2	3/4
3	7/8
4	15/16
5	31/32
$\vdots$	
k	$1 - 1/2^k$

The approximation factor gets better and better as k increases. Meanwhile, the first row, $k = 1$, is goofy, since 1-SAT is trivial. 2-SAT is not as trivial but there is a polynomial time algorithm for this problem as well. So oblivious rounding is not so great for very small values of k.

This is relevant as we now consider the more general form of max-SAT, where we are given a formula in CNF where each clause can have any number of clauses. We can still apply our oblivious randomized algorithm that flips a coin for every variable.

If every clause had *at least* k variables, then we obtain a $1 - 1/2^k$ approximation ratio. But the presence of single-variable clauses, which should cause the least trouble, means we only have a 1/2-approximation ratio in general. We will use linear programming to improve the approximation ratio.

An integer program for SAT. We first translate max-SAT to an integer program. Fix a formula $f(x_1, \dots, x_n)$ of n variables, consisting of m clauses $C_1, \dots, C_m$. Let OPT denote the maximum number of satisfiable clauses. Let "$x_j \in C_i$" indicate that the symbol x_j appears in C_i (without negation) and "$\bar{x}_j \in C_i$" indicate that $\bar{x}_j$ appears in C_i.

Consider now the following integer program.

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{m} z_i \text{ over } y \in \{0,1\}^n,\ z \in \{0,1\}^m \\ \text{s.t. } & \sum_{j : x_j \in C_i} y_j + \sum_{j : \bar{x}_j \in C_i} (1 - y_j) \geq z_i \text{ for all clauses } C_i. \end{aligned}$$

The integer program has $\{0,1\}$-variables for each variable x_j and each clause C_i. Intuitively:

1. For $j = 1, \dots, n$, $y_j \in \{0,1\}$ indicates whether we set $x_j = \texttt{t}$ ($y_j = 1$) or $x_j = \texttt{f}$ ($y_j = 0$).
2. For $i = 1, \dots, m$, $z_i \in \{0,1\}$ indicates whether we satisfy the ith clause ($z_i = 1$) or not ($z_i = 0$).

The integer program seeks to maximize the number of satisfied clauses, represented by $\sum_{i=1}^m z_i$. For each clause C_i, the corresponding constraint implies we can only set $z_i = 1$ if $y_j = 1$ for some $x_j \in C_i$ or $y_j = 0$ for some $\bar{x}_j \in C_i$.

A linear program for SAT. We cannot solve integer programs in polynomial time but we can solve linear programs. The integer program above relaxes to a linear program by allowing each variable y_j or z_i to lie anywhere in the interval $[0,1]$.

$$\begin{aligned} \text{maximize } & \sum_{i=1}^{m} z_i \text{ over } y_1, \dots, y_n, z_1, \dots, y_m \in \mathbb{R} \\ \text{s.t. } & \sum_{j : x_j \in C_i} y_j + \sum_{j : \bar{x}_j \in C_i} (1 - y_j) \geq z_i \text{ for all clauses } C_i \\ & 0 \leq z_i \leq 1 \text{ for all } i = 1, \dots, m \\ & 0 \leq y_j \leq 1 \text{ for all } j = 1, \dots, n \end{aligned}$$

Let OPT_{LP} denote the optimum value of the LP. Since the LP is a relaxation of the preceding IP, we have $\text{OPT}_{\text{LP}} \geq \text{OPT}$.

Rounding the LP solution. Let $y_1, \dots, y_n$ and $z_1, \dots, z_m$ be an optimum solution to the LP. We know that the objective value $\text{OPT}_{\text{LP}} = \sum_i z_i$ is very good. Our goal now is to convert the y_j's into *discrete* decisions while keeping the objective value as close to OPT_{LP} as possible.

The basic question is: how do we interpret a *fractional* value such as $y_1 = .5$? The LP seems to suggest that we should set x_1 to be one-half true and one-half false. Of course the are no half values in boolean algebra and "one-half true and one-half false" is total nonsense. A different interpretation is that of a randomized experiment where we set $x_1 = \texttt{t}$ *half the time*, and $x_1 = \texttt{f}$ the other half. Consider the following *randomized rounding* algorithm:

`Randomized rounding for SAT`

1. Let $y_1, \dots, y_n$ be an optimum solution to the LP for max-SAT.
2. For each variable x_j, independently, randomly set $x_j = \texttt{t}$ with probability y_j, and $x_j = \texttt{f}$ otherwise.

Next we analyze the expected number of clauses satisfied by randomized rounding. By linearity of expectation this boils down to analyzing the probability of satisfying each individual clause.

Lemma 26.1. *Each clause C_i is satisfied with probability at least $(1 - 1/e)z_i$.*

Proof. A clause C_i is not satisfied iff we randomly set $x_j = \texttt{f}$ for all $x_j \in C_i$, and set $x_j = \texttt{t}$ for all $\bar{x}_j \in C_i$. Thus

$$\mathbf{P}[C_i \text{ not satisfied}] = \prod_{j : x_j \in C_i} (1 - y_j) \prod_{j : \bar{x}_j \in C_i} y_j.$$

Now, by the inequality $1 + x \leq e^x$ (for all x), we can simply the RHS as

$$e^{-\left(\sum_{j : x_j \in C_i} y_j + \sum_{j : \bar{x}_k \in C_i} (1 - y_j)\right)} \leq e^{-z_i}.$$

Here the inequality follows from the LP constraint for C_i. Finally, by convexity[2] of $f(x) = e^{-x}$, we have

$$e^{-z_i} \leq (1 - z_i)e^0 + z_i e^{-1} = 1 - (1 - 1/e)z_i,$$

as desired. □

[2] $f(x)$ is convex if $f(ta + (1-t)b) \leq tf(a) + (1-t)f(b)$ for all a, b and all $t \in [0, 1]$.

By linearity of expectation, the expected number of clauses that are satisfied equals the sum of probabilities of each clause being satisfied. Therefore, by Lemma 26.1, we will satisfy at least $(1-1/e)\,\mathrm{OPT}_{\mathrm{LP}} \geq (1-1/e)\,\mathrm{OPT}$ clauses in expectation. In conclusion:

Theorem 26.2. *There is a $(1-1/e)$-approximation algorithm for max-SAT.*

Our bound is only interesting when there are clauses with one or two variables; otherwise oblivious rounding is still better.

The best of both worlds. Part of the problem is that Lemma 26.1 is not tight for small k. Take for example $k=1$. Obviously, if $|C_i|=1$, then C_i is satisfied with probability z_i, not $(1-1/e)z_i$. For $k=2$, we have the following better analysis.

Lemma 26.3. *If $|C_i|=2$, then C_i is satisfied with probability $\geq 3z_i/4$.*

Proof. Suppose for simplicity that $C_i = x_1 \vee x_2$. (It will be obvious how to generalize the analysis to other pairs of variables.) We have

$$\begin{aligned}\mathbf{P}[C_i \text{ not satisfied}] = (1-y_1)(1-y_2) &\overset{\text{(a)}}{\leq} \left(\frac{(1-y_1)+(1-y_2)}{2}\right)^2 \\ &\leq \left(1-\frac{z_j}{2}\right)^2 \overset{\text{(b)}}{\leq} (1-z_j)\left(1-\frac{0}{2}\right)^2 + z_j\left(1-\frac{1}{2}\right)^2 \\ &= 1 - z_j + \frac{z_j}{4} = 1 - \frac{3}{4}z_j.\end{aligned}$$

(a) is by AM-GM. (b) is by convexity of $f(x) = \left(1-\frac{x}{2}\right)^2$. □

Below we list the probability of clause being satisfied by the oblivious and LP rounding strategies, as a function of the number of variables in the clause, k. Observe that the average of the probabilities is at least $(3/4)z_i$ for all k, as indicated in the column on the right.

k	**oblivious**	**LP**	**average**
1	1/2	z_i	$\geq (3/4)z_i$
2	3/4	$(3/4)z_i$	$\geq (3/4)z_i$
3	7/8	$(1-1/e)z_i$	$\geq (3/4)z_i$
4	15/16	$(1-1/e)z_i$	$\geq (3/4)z_i$
5	31/32	$(1-1/e)z_i$	$\geq (3/4)z_i$
$\vdots$			
k	$1-1/2^k$	$(1-1/e)z_i$	$\geq (3/4)z_i$.

The table suggests that we might be able to merge the oblivious sampling and LP rounding algorithms to obtain a 3/4-approximation ratio. But the two approaches seem like polar opposites: one approach is completely oblivious, and the other strongly depends on the formula (implicitly via the LP solver). But here's a trick: just pick one of the two strategies uniformly at random. The "averaging" will work itself out in the analysis.

Hybrid algorithm for max-SAT.

1. With probability 1/2, return a uniformly random assignment.
2. Otherwise solve and randomly round the LP.

Theorem 26.4. *The hybrid algorithm gives a* (3/4)*-approximation algorithm for max-SAT.*

We leave the analysis to the reader as exercise 26.1.

26.2 Set cover

In *set cover*, we are given m points $[m] = \{1, \dots, m\}$, and n sets $S_1, \dots, S_n \subseteq [m]$. The goal is to

$$\textit{find the minimum number of sets } S_{i_1}, \dots, S_{i_k} \textit{ such that } S_{i_1} \cup \cdots \cup S_{i_k} = [m].$$

Some natural extensions including adding costs for sets, pointwise demands that require points to be covered by multiple sets, and coefficients $A_{ij} \in [0, 1]$ that indicate the amount of coverage a set S_i gives to point j. These will be considered when we discuss CIPs later. Set Cover is NP-Hard. Instead we will design an approximation algorithm for set cover, via randomized rounding.

We first write an LP relaxation for set cover. For each set S_j we introduce a variable x_j that models our decision to take S_j in our set cover. Consider the following LP.

$$\text{minimize } \sum_{j=1}^{n} x_j \text{ over } x \in \mathbb{R}^n_{\geq 0} \text{ s.t. } \sum_{S_j \ni i} x_j \geq 1 \text{ for all } i \in [m].$$

The objective, $\sum_{j=1}^{n} x_j$, is the (fractional) number of sets in our (fractional) set cover. For each point $i \in [m]$, we require at least one fractional set among the family of sets that cover that point.

Randomized rounding. Let us consider the following randomized rounding algorithm

1. Let $x_1, \dots, x_n$ be an optimum solution to the set cover LP.
2. For each set S_j, independently, take S_j with probability x_j.

Let $F \subseteq \{S_1, \dots, S_m\}$ be the random family of sets produced by randomized rounding. We have two questions to address:

1. How big is F, relative to OPT?
2. Is F a set cover?

For the first question, we have

$$\mathbf{E}[|F|] = \sum_j \mathbf{P}[S_j \in F] = \sum_j x_j = \text{OPT}_{\text{LP}},$$

which is very good indeed. Now, is F a set cover? Fix a point $i \in [m]$. The expected number of sets covering i is

$$\mathbf{E}[\#\text{ sets in } F \text{ covering } i] = \sum_{S_j \ni i} \mathbf{P}[S_j \in F] = \sum_{S_j \ni i} x_j \geq 1.$$

So i expects to have at least 1 set in F containing it. But this does not imply that F is a set cover, that F covers *all* of $[m]$ *simultaneously*, with any nonnegligible probability.

A better question asks for the probability that F covers i, or rather, the probability that F doesn't cover i. F does not cover i iff it fails to sample any of the sets covering i, hence

$$\mathbf{P}[F \textit{ doesn't} \text{ cover } i] = \prod_{S_j \ni i} (1 - x_j) \leq e^{-\sum_{S_j \ni i} x_j} \leq e^{-1}.$$

So each point is covered with constant probability $1 - 1/e$.

The question is: how do we increase the probability that *all points are covered*, simultaneously?

Scaling and rounding. The problem above is that having each point covered with constant probability does *not* imply that all points are covered with constant, or any nonnegligible, probability. However, having each point covered with *high probability* does imply, *by the union bound*, that all points are covered with *high probability*. Now, how can we ensure that each point is covered with high probability? By scaling up x before rounding.

Randomized rounding for set cover.

1. Let $x_1, \dots, x_n$ be an optimum solution to the set cover LP. *Let* $\alpha = 2\log(m)$.
2. For each set S_j independently, take S_j with probability $\min\{1, \alpha x_j\}$.

As before, let F denote the random collection of sets returned by the algorithm.

Lemma 26.5. *Each point i is covered with probability $1 - 1/m^2$.*

Proof. If $x_j \geq 1/\alpha$ for any set S_j covering i, then i is covered deterministically. Otherwise, by similar calculations as before, we have

$$\mathbf{P}[F \text{ doesn't cover } i] = \prod_{S_j \ni i} (1 - \alpha x_j) \leq e^{-\sum_{S_j \ni i} \alpha x_j} \leq e^{-\alpha} = 1/m^2. \tag{26.2}$$

□

Lemma 26.6. *F is a set cover with probability at least $1 - 1/m$.*

Proof. By the union bound, we have

$$\mathbf{P}[F \text{ is not a set cover}] \leq \sum_i \mathbf{P}[F \text{ doesn't cover } i] \leq \frac{1}{m},$$

as desired. □

Theorem 26.7. *Randomized rounding is a $O(\log m)$-approximation algorithm for set cover.*

Proof. We have $\mathbf{E}[|F|] = 2\log(m)\,\mathrm{OPT}$, and by Markov's inequality, $|F| \leq 4\log(m)\,\mathrm{OPT}$ with probability $1/2$. F is also a set cover with probability $1 - 1/m$. By the union bound, with probability of error at most $1/2 + 1/m$, F is a set cover of size at most $4\log(m)\,\mathrm{OPT}$. □

26.3 Covering integer programs

Lastly, wxe consider covering integer programs. They are discrete problems generalizing set cover, of the form

$$\text{minimize } \langle c, x \rangle \text{ over } x \in \mathbb{Z}^n_{\geq 0} \text{ s.t. } Ax \geq b.$$

where $c \in \mathbb{R}^n_{>0}$, $A \in \mathbb{R}^{m \times n}_{\geq 0}$, and $b \in \mathbb{R}^m_{>0}$. The matrix notation expands out to the following.

$$\begin{aligned} \text{minimize } & \sum_{j=1}^n c_j x_j \\ \text{over } & x \in \mathbb{Z}^n_{\geq 0} \\ \text{s.t. } & \sum_{j=1}^n A_{ij} x_j \geq b_i \text{ for } i \in [m]. \end{aligned}$$

We assume without loss of generality that $A_{ij} \leq b_i$ for all i. (Why?) We also assume without loss of generality that m is at least some constant; say, 4. (Otherwise add a few empty constraints.)

The algorithm we analyze is essentially the same as for set cover. We scale up x by a $O(\log m)$ factor, and then randomly round each coordinate of x, independently, to integer values.

1. Let $\alpha = 8 \log(m)$.
2. For each $j \in [n]$, independently, let

$$z_j = \begin{cases} \lceil \alpha x_j \rceil & \text{with probability } \alpha x_j - \lfloor \alpha x_j \rfloor, \\ \lfloor \alpha x_j \rfloor & \text{otherwise.} \end{cases}$$

The algorithm is very similar to set cover. However, the coefficients A_{ij} make the analysis more difficult. For set cover, we were able to give an exact formula for the probability that a point i is uncovered. (Equation (26.2).) Here it is not so simple, and we instead appeal to the *multiplicative* (or *relative*) Chernoff bound introduced last chapter. The proof is deferred to Section 26.A.

Theorem 26.8. *Let $X_1, \ldots, X_n \in [0,1]$ be independent random variables, and $\epsilon \in [0,1]$.*

- *For $\mu \geq \mathbf{E}[X_1 + \cdots + X_n]$,*

$$\mathbf{P}[X_1 + \cdots + X_n \geq (1+\epsilon)\mu] \leq e^{-\epsilon^2 \mu/3}.$$

- *For $\mu \leq \mathbf{E}[X_1 + \cdots + X_n]$,*

$$\mathbf{P}[X_1 + \cdots + X_n \leq (1-\epsilon)\mu] \leq e^{-\epsilon^2 \mu/2}.$$

The main part of the analysis is to show that a single constraint is satisfied with high probability, as follows.

Lemma 26.9. *The ith constraint is satisfied with probability at least* $1 - 1/m^2$.

Proof. By scaling, we may assume that $b_i \leq 1$, hence $A_{ij} \leq 1$ for all j. We may assume that $\sum_j A_{ij} \lfloor \alpha x_j \rfloor < 1$, since otherwise the constraint is satisfied deterministically.

For each coordinate j, let $X_j = A_{ij}(z_j - \lfloor \alpha x_j \rfloor)$ for each j. Let $\mu = \mathbf{E}\Big[\sum_{j=1}^n X_j\Big]$. We have

$$\mu = \sum_{j=1}^n A_{ij}(\alpha x_j - \lfloor \alpha x_j \rfloor) \geq \alpha - \sum_{j=1}^n A_{ij} \lfloor \alpha x_j \rfloor.$$

This implies that both

$$\mu \geq \alpha \left(1 - \sum_{j=1}^n A_{ij} \lfloor \alpha x_j \rfloor \right), \text{ and } \mu \geq \alpha - 1 \geq 8 \log(m).$$

We have

$$\begin{aligned} \mathbf{P}[z \text{ fails constraint } i\ ] &= \mathbf{P}\left[\sum_{j=1}^n A_{ij} z_j \leq 1 \right] \\ &= \mathbf{P}\left[\sum_{j=1}^n X_j \leq 1 - \sum_{j=1}^n A_{ij} \lfloor \alpha x_j \rfloor \right] \\ &\leq \mathbf{P}\left[\sum_{j=1}^n X_j \leq \frac{\mu}{\alpha} \right]. \end{aligned}$$

By the Chernoff inequality, we have

$$\mathbf{P}\left[\sum_{j=1}^n X_j \leq \frac{\mu}{\alpha} \right] \leq e^{-(1-1/\alpha)^2 \mu/2}.$$

Now,

$$\frac{1}{2}\Big(1 - \frac{1}{\alpha}\Big)^2 \mu \geq \frac{1}{2}\Big(1 - \frac{1}{\alpha}\Big)^3 \alpha \geq \frac{\alpha}{4} \geq 2 \log(m).$$

Thus

$$\mathbf{P}[z \text{ fails constraint } i] \leq e^{-(1-1/\alpha)^2 \mu/2} \leq e^{-2\ln(m)} = \frac{1}{m^2},$$

as desired. □

Theorem 26.10. *With constant probablity, the randomized rounding algorithm returns a feasible solution of cost at most* $O(\ln(m))\,\mathrm{OPT}_{LP}$.

Proof. By Markov's inequality, we have

$$\sum_j c_j z_j \leq 2\,\mathbf{E}\left[\sum_j c_j z_j\right] = 2\alpha\,\mathrm{OPT}_{\mathrm{LP}}$$

with probability of error at most $1/2$. Each constraint i is satisfied with probability of error at most $1/m^2$.

By the union bound, the probability of z not being a feasible solution of cost at most $2\,\mathrm{OPT}_{\mathrm{LP}}$ is at most

$$\frac{1}{2} + \mathbf{P}[z \text{ fails any constraint}] \leq \frac{1}{2} + \sum_i \mathbf{P}[z \text{ fails constraint } i] \leq \frac{1}{2} + \frac{1}{m},$$

as desired. □

26.4 Additional notes and materials

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:

- Handwritten .pdf prepared before the lecture (and .note file).
- Handwritten .pdf annotated during the presentation (and .note file).
- Recorded video lecture.

Fall 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

26.5 Exercises

Exercise 26.1. Complete the proof of Theorem 26.4.

Exercise 26.2. Extend the approximation algorithm for set cover to positive costs. For each set, there is a positive cost $c_j > 0$. The goal is to compute the minimum cost collection of sets that covers all the points.

Exercise 26.3. Design and analyze a deterministic 3/4-approximation algorithm for max-SAT.[3]

Exercise 26.4. Consider an instance of (weighted) set cover defined by sets $S_1, \ldots, S_n \subseteq [m]$ and costs $c_i > 0$ for each set S_i. The goal is to compute the minimum cost collection of sets covering $[m]$. We saw that solving the LP and then randomly rounding gives a $O(\log m)$ approximation. Here we consider a special case where all the sets are small and obtain a better approximation factor by a standard extension of randomized rounding called *alterations*.

Let $\Delta \in \mathbb{N}$ be such that $|S_j| \leq \Delta$ for all j. Consider the algorithm `round-and-fix` for which some pseudocode is given below. `round-and-fix` is similar to randomized rounding and has two stages. The first stage solves the LP and then *rounds* the solution scaled up by some factor $\alpha \geq 1$. It is possible that some of the elements $i \in [m]$ may not be covered. In the second stage, we *fix* each uncovered element by (deterministically) taking the cheapest set that covers it.

[3] You may first want to design and analyze a deterministic $(1 - 1/e)$-approximation algorithm for max-SAT.

`round-and-fix(sets` $S_1, \dots, S_n \subseteq [m]$`, costs` $c \in \mathbb{R}^n_{>0}$`,` $\alpha \geq 1$`)`

1. let $x \in [0,1]^n$ solve the set cover LP
2. let $F \subseteq \{S_1, \dots, S_n\}$ sample each set S_i independently with probability $\min\{1, \alpha x_i\}$
3. for each $i \in [m]$
   A. if i is not covered by F
      1. add the cheapest set covering i to F
4. return F

Show that for an appropriate choice of α, this algorithm returns a $O(\log \Delta)$ approximation to the set cover instance (in expectation). (It is possible to get $\log \Delta + \log\log \Delta + O(1)$ with care.)

Exercise 26.5. The defining characteristic of LPs is that the objective and all linear constraints are given by linear functions. It is natural to generalize this notion and consider mathematical programs where the objective and linear constraints are all given by low-degree polynomials; say, bounded by a degree d. Let us call these "degree d polynomial programs". Linear programs are degree 1 polynomial programs.

Prove that degree d polynomial programs are NP-Hard to solve for $d \geq 3$. To this end, pick a suitable NP-Hard problem, and design a degree 3 polynomial program that can be rounded to a discrete solution without any loss.[4]

Exercise 26.6. For CIPs, we could assume without loss of generality that $A_{ij} \leq b_j$ for all i, j. (In fact this assumption was critical for applying the Chernoff bound.) Suppose now that we had $\lambda A_{ij} \leq b_j$ for all i, j for a parameter $1 \leq \lambda \leq \log(n)$. Design and analyzing a $O(\log(m)/\lambda)$-approximation algorithm for this setting.

Exercise 26.7. In CIPs we allowed each x_j to be as large as we want. Suppose we added the constraint $x_j \leq 1$ for all j. Would the randomized rounding algorithm from Section 26.3 still obtain a $(1 - 1/e)$-approximation ratio? Why or why not?

Exercise 26.8. Recall the set cover problem for which we obtained a randomized $O(\log n)$-approximation. Here we consider a (maximum weight) set *packing* problem, defined as follows.

[4] Better yet, prove the same for $d \geq 2$.

Let $[m]$ be a set of points, and let $S_1, \dots, S_n \subseteq [m]$ be n subsets of $[m]$. Let $b_1, \dots, b_n > 0$ represent the profit of $S_1, \dots, S_n$, respectively. We say that a collection of sets $\mathcal{F} = \{S_{j_1}, \dots, S_{j_k}\}$ is a *set packing* if they are all disjoint. The total profit of such a set packing is defined as the sum of profits $b_{j_1} + \cdots + b_{j_k}$ of the corresponding sets.

The goal is to compute a set packing of maximum profit, but the problem is NP-Hard. Here we consider the following (perhaps unusual) approximation criteria. Let OPT denote the maximum profit of any set packing. For $\alpha \geq 1$, we say that a collection of sets $S_{j_1} + \cdots + S_{j_k}$ is an α-packing if each point is covered by at most α sets S_{j_h}. We say that a randomized collection of sets $\mathcal{F}$ is a *randomized approximate α-packing* if

1. The expected total profit of $\mathcal{F}$ is at least OPT.
2. With high probability, $\mathcal{F}$ is an α-packing.

Design and analyze a polynomial time algorithm that outputs a randomized approximate α-packing for α as small as possible.[5]

Exercise 26.9. Our department has a large catalog of classes and a handful of course requirements you have to satisfy to graduate. The requirements seem to be particularly complicated for undergraduates. Each requirement is defined by a subset of classes and a number specifying the number of courses you have to take from this subset.

Formally, let the courses be indexed 1 through n. We have L requirements $(S_1, k_1), \dots, (S_L, k_L)$, each consisting of a set $S_i \subseteq [n]$ and an integer $k_i \in \mathbb{N}$. For each i, you have to take at least k_i classes from S_i. Let $T \subseteq [n]$ be the set of classes you've already taken. Let $m = \sum_i |S_i|$ denote the total size of all the sets.

Now we have two possible interpretations of the rules. In the first version, a single class can only count towards a single requirement. In the second version, a single class can count towards any number of requirements. For example, suppose you are required to take $(k_1 = 2)$ classes from $S_1 = \{A, B, C\}$ and $(k_2 = 2)$ classes from $S_2 = \{C, D, E\}$, and you have already taken $T = \{B, C, D\}$. In the first version, you do not have enough classes to graduate; in the second version, you do have enough classes to graduate.

Here we have related but slightly different problems for the two systems.

[5] Here we are interested in the approximation factor α – the smaller and closer to 1 the better – and not the exact polynomial running time.

1. (5 points) Suppose a single class can only count towards a single requirement. Consider the problem of deciding if you have already taken enough classes to graduate. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Now suppose a single class can count towards any number of requirements. Consider the problem of identifying the minimum number of additional classes that need to be taken to satisfy all the requirements. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 26.10. Imagine we have n unknown nonnegative numbers $x_1, \dots, x_n \in \mathbb{R}_{\geq 0}$, with each x_i hidden in a box B_i. Each box is labeled by a number y_i with the promise that

$$y_i \leq x_i \leq \alpha y_i$$

for a fixed and known parameter $\alpha \geq 1$.



y_1 y_2 y_3 y_4 y_5 y_6 y_7

We want to approximate the sum of the x_i's while opening as few boxes as possible. For given parameters $\epsilon, \delta \in (0,1)$, design and analyze a randomized algorithm that computes a $(1 \pm \epsilon)$-approximation of the sum with probability of error at most δ, opening as few boxes as possible (up to constant factors).

26.A Proof of the multiplicative Chernoff bound

Theorem 26.8. *Let $X_1, \dots, X_n \in [0,1]$ be independent random variables, and $\epsilon \in [0,1]$.*

- *For $\mu \geq \mathbf{E}[X_1 + \cdots + X_n]$,*

$$\mathbf{P}[X_1 + \cdots + X_n \geq (1+\epsilon)\mu] \leq e^{-\epsilon^2 \mu/3}.$$

- *For $\mu \leq \mathbf{E}[X_1 + \cdots + X_n]$,*

$$\mathbf{P}[X_1 + \cdots + X_n \leq (1-\epsilon)\mu] \leq e^{-\epsilon^2 \mu/2}.$$

Proof. We prove the claim for a slightly weaker constant in the exponent. Let $t \in [0,1]$ be a parameter TBD. We have

$$\mathbf{P}[X_1 + \cdots + X_n \geq (1+\epsilon)\mu] = \mathbf{P}\Big[e^{t(X_1+\cdots+X_n)} \geq e^{t(1+\epsilon)\mu}\Big] \overset{\text{(a)}}{\leq} \frac{\mathbf{E}\Big[e^{t(X_1+\cdots+X_n)}\Big]}{e^{t(1+\epsilon)\mu}}$$

by (a) Markov's inequality. Next analyze $\mathbf{E}\Big[e^{\epsilon X_1+\cdots+X_n}\Big]$. We first have

$$\mathbf{E}\Big[e^{t(X_1+\cdots+X_n)}\Big] = \prod_{i=1}^{n} \mathbf{E}\Big[e^{tX_i}\Big]$$

by independendence of the X_i's. For each i, we have

$$e^{tX_i} \leq 1 + tX_i + t^2X_i^2 \leq 1 + \Big(t + t^2\Big)X_i$$

where we recall that $0 \leq t, X_i \leq 1$ and apply the inequality $e^x \leq 1 + x + x^2$ for $x \leq 1$. Consequently

$$\mathbf{E}\Big[e^{tX_i}\Big] \leq \mathbf{E}\Big[1 + \Big(t + t^2\Big)X_i\Big] \leq 1 + \Big(t + t^2\Big)\,\mathbf{E}[X_i] \leq e^{(t+t^2)\,\mathbf{E}[X_i]},$$

where the last inequality follows from $1 + x \leq e^x$ for all x. Now we have

$$\mathbf{E}\Big[e^{t(X_1+\cdots+X_n)}\Big] \leq \prod_{i=1}^{n} \mathbf{E}\Big[e^{tX_i}\Big] \leq \prod_{i=1}^{n} \mathbf{E}\Big[e^{(t+t^2)\,\mathbf{E}[X_i]}\Big] \leq e^{(t+t^2)\mu},$$

hence

$$\mathbf{P}[X_1 + \cdots + X_n \geq (1+\epsilon)\mu] \leq e^{(t+t^2)\mu - t(1+\epsilon)\mu} = e^{t(t-\epsilon)\mu}.$$

The RHS is minimized by setting $t = \epsilon/2$, giving $e^{-\epsilon^2\mu/4}$.

The proof for the second inequality is similar and we provide a sketch highlighting a differences. Let $t \in (0,1]$ by a parameter TBD. We have

$$\begin{aligned}\mathbf{P}[X_1 + \cdots + X_n \leq (1-\epsilon)\mu] &= \mathbf{P}\Big[e^{-t(X_1+\cdots X_n)} \geq e^{-(1-\epsilon)t\mu}\Big] \\ &\leq \mathbf{E}\Big[e^{-t(X_1+\cdots+X_n)}\Big]e^{(1-\epsilon)t\mu}\end{aligned}$$

Note the introduction of the negative sign to reverse the inequality before applying Markov's inequality. As before, we apply independence to break the expectation into individual moments $\mathbf{E}\Big[e^{-tX_i}\Big]$, which are each bounded above by

$$\mathbf{E}\Big[e^{-tX_i}\Big] \leq 1 - \Big(t - t^2/2\Big)\,\mathbf{E}[X_i] \leq e^{-\left(t-t^2/2\right)\mathbf{E}[X_i]},$$

where we applied the inequality $e^x \leq 1 + x + x^2/2$ for $x \leq 0$. This leads to

$$\mathbf{P}[X_1 + \cdots + X_n \leq (1-\epsilon)\mu] \leq e^{(t^2/2-\epsilon t)\mu};$$

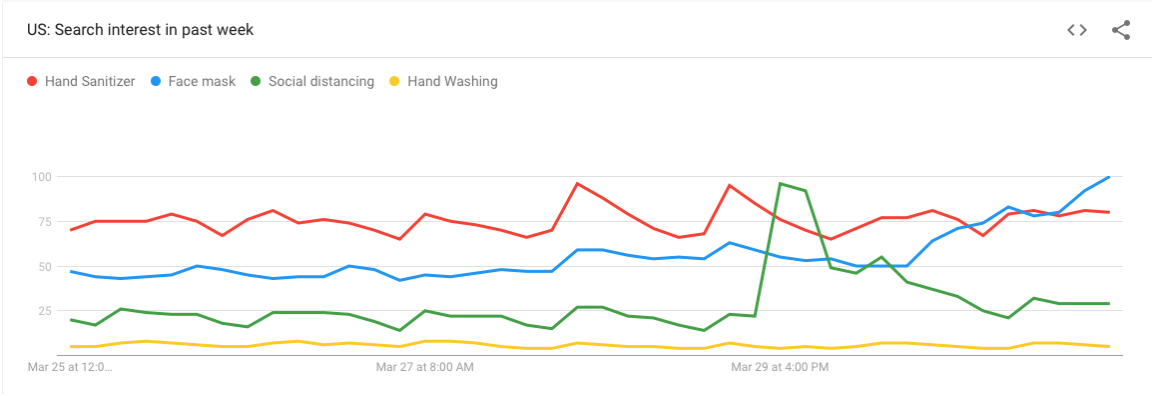
The RHS is minimized by $t = \epsilon$, giving the desired bound. □

Chapter 27

Hashing and heavy hitters

27.1 Introduction

Google has an interesting web page called "Google trends", which tracks surging search queries around the world in real time. In Spring 2021, there was even a subpage of search trends specifically related to the Covid-19.



Google tracks the trending search queries not just for the sake of curiosity. Its goal is not only to serve queries, but to serve queries *fast*. The best way to serve something quickly is to have it ready before it is even asked for. By keeping track of the 'heavy hitter" search terms - a few search terms that make up a disproportionate amount of the search traffic - Google can cache the answers to most search requests before they are even made.

Google currently serves billions[1] of queries a day. Given the sheer magnitude of Google's search traffic, and the diversity of search queries, it is not obvious how to

[1]Maybe 7 billion? See https://www.internetlivestats.com/google-search-statistics/

identify the most popular search queries. Certainly one cannot simply have a tally for each search term, since there are too many search terms out there to be stored. More generally, it is prohibitively expensive to maintain any data structure proportional to the input size. Somehow we need an approach that takes *sublinear* space.

Streaming models. We study the heavy hitter problem in the *streaming model* of computation. In the streaming model, the input is a sequence of items presented to the algorithm *one at a time*. We assume the algorithm has much less memory than the size of the input. In particular, it cannot simply write down everything and solve the problem offline. For example, suppose the stream had m items in the stream, and the algorithm was allowed only $O(\sqrt{m})$ space. When the space is so much smaller than the input size, each time an item from the stream is given to the algorithm, the algorithm needs to fairly selective about what parts of the item (if any) it wants to remember.



The heavy hitters problem. We formalize the heavy hitters problem as follows. We have m elements in a stream $e_1, \ldots, e_m$, where each element is from some large universe $[n] = \{1, \ldots, n\}$. Elements can repeat. The *absolute frequency* of an element e, denoted f_e, is the number of times the element appears in the stream. The *relative frequency*, denoted p_e, is the fraction of the stream that the element appears in. In a stream of m elements, the relative frequency of an element is the absolute frequency divided by m.

Ideally we would track all of the elements exactly. This is impossible with less than $\min\{n, m\}$ bits[2]. Here, and unlike standard algorithmic settings, upper bounds like $O(m)$ or $O(n)$ bits are not good enough. So we relax our requirements and formulate problems that are more *tractable* but still *useful*. We first relinquish exactness and consider *approximations*, where we allow for some error that we can analyze and control. We also give up on deterministic computation, and design *randomized* algorithms that have some small — analyzed and controlled — *probability of failure*.

[2]One can formalize this impossibility as follows. There are $\binom{m+n-1}{n-1}$ ways to make a frequency vector of n items where the total sum is m. (*Why?*) Suppose we claim that we can describe any combination of counts with k bits. Each k-bit string can describe by only one of these $\binom{m+n-1}{n-1}$

ϵ-heavy hitters. Given a fixed parameter $\epsilon \in (0, 1)$, an *ϵ-heavy hitter* is an element with relative frequency $\geq \epsilon$. The heavy hitters problem is to identify all of the ϵ-heavy hitters for an input parameter $\epsilon > 0$. Note that there can only be $(1/\epsilon)$-many ϵ-heavy hitters, which preserves some hope that we can identify all of them with space proportional to $1/\epsilon$, rather than m.

We also consider a closely related problem of *(approximate) frequency estimation*. Given a fixed error parameter $\epsilon > 0$, the goal is to estimate every element's relative frequency up to an additive error of ϵ. Equivalently, we want to estimate the absolute frequency of each element up to an ϵm-additive factor. At first it might seem impossible to estimate n-many counts with $o(n)$ space. We have already argued that m exact counters is impossible. Allowing for ϵ-relative error, however, means that 0 is a satisfactory estimate for all but at most $\lfloor 1/\epsilon \rfloor$ elements.

If we had a data structure that can estimate the absolute frequency of each element up to an additive error parameter ϵ, then one can use such a data structure with (approximately) maintain a list of all the heavy hitters. See exercise 27.7. Of course frequency estimation gives more information than just who are the heavy hitters. By knowing their frequencies up to some small error, one can also rank them (approximately) from most to least frequent, such as in Google trends.

Conversely, if we knew *a priori* which of the elements are the ϵ-heavy hitters, then ϵ-frequency estimation is trivial. Namely, we would maintain a counter for each of the $\lfloor 1/\epsilon \rfloor$-many ϵ-heavy hitters. All other elements are ignored and assigned frequency 0. Of course this approach is not possible since we do not know the heavy hitters.

Surprisingly we will pursue a strategy that is actually quite similar. We will allocate $w = O(1/\epsilon)$ counters, hoping to use one counter for each heavy hitter (plus a few extra for safe measure). Although we do not know the heavy hitters, we will use *randomized hash functions* to isolate them implicitly.

---

outcomes, so we must have $2^k \geq \binom{m+n-1}{n-1}$, hence $k \geq \log\left(\binom{m+n-1}{n-1}\right)$. Here log denotes $\log_2$. We also have

$$\binom{n+m-1}{n-1} = \left(\frac{n+m-1}{n-1}\right)^{n-1} \text{ and also } \binom{n+m-1}{n-1} = \binom{n+m-1}{m} \geq \left(\frac{n+m-1}{m}\right)^{m}$$

Thus

$$k \geq (n-1)\log\left(1 + \frac{m}{n-1}\right) \text{ and } k \geq m \log\left(1 + \frac{n-1}{m}\right).$$

Slightly better low bounds can be obtained via *Stirling's approximation*, which is also related to entropy.

27.2 Hashing

Likely the reader has used hash tables before, and may be aware that they use *hash functions* to randomly map keys to slots in an array. (We discuss hash tables in detail in the following chapter.)

Loosely speaking, a hash function is a randomly constructed function $h : [n] \to [k]$ where the values $h(i)$ are (in a qualified sense) randomly distributed through $[k]$. A *collision* is a pair of distinct indices $i_1 \neq i_2 \in [n]$ such that $h(i_1) = h(i_2)$. In most applications, n is much (much, much) larger than k. In this case, there are necessarily many collisions (see exercise 27.3). A goal of hash functions is to distribute these collisions "fairly".

Ideal hash functions. One way to construct a hash function, for example, is to sample, for each $i \in [n]$, a value $h(i) \in [k]$ independently and uniformly at random. This produces an "ideal hash function", defined as follows.

Definition 27.1. *An* ideal hash function $h : [n] \to [k]$ *is a uniformly random function* $h : [n] \to [k]$*. That is, each* $h(i)$ *is drawn from* $[k]$ *independently and uniformly at random.*

An ideal hash function h is particularly easy to reason about. For example, for every input i and possible output $j \in [k]$, we have

$$\mathbf{P}[h(i) = j] = \frac{1}{k}.$$

More generally, for any ℓ distinct inputs $i_1, \ldots, i_\ell \in [n]$ and ℓ possible outputs $j_1, \ldots, j_\ell \in [\ell]$, we have

$$\begin{aligned}
&\mathbf{P}[h(i_1) = j_1,\, h(i_2) = j_2,\, \ldots,\, h(i_\ell) = j_\ell] \\
&= \mathbf{P}[h(i_1) = j_1]\, \mathbf{P}[h(i_2) = j_2] \cdots \mathbf{P}[h(i_\ell) = j_\ell] \\
&= \frac{1}{k^\ell}.
\end{aligned}$$

Ideal hash functions are a good model to keep in mind when designing randomized algorithms. Assuming the hash values are completely independent simplifies calculations. In reality, however, ideal hash functions are very expensive to make and store. Indeed, one has to have $n \log k$ bits to be able to describe all of the possible functions from $[n]$ to $[k]$ (as there are k^n such functions, and we must pay the logarithm of this quantity). This is particularly ill-suited to our streaming setting where n is astronomical and the goal is to use space sublinear to the input size.

Universal hash functions. Fortunately, in most applications, only a *limited amount of randomization* is actually required. For the current discussion, we only require "universal" hash functions that have "ideal pairwise collision probabilities", in the following sense.

Definition 27.2. *A random hash function $h : [n] \to [k]$ is* universal *if for any distinct indices $i_1 \neq i_2 \in [n]$, we have*

$$\mathbf{P}[h(i_1) = h(i_2)] \leq \frac{1}{k}.$$

In contrast to ideal hash functions, universals hash functions can be constructed compactly, as described in the following theorem.

Theorem 27.3. *Consider the randomly constructed function $h : [n] \to [k]$*

$$h(x) = (ax + b \mod p) \mod k,$$

where p is a prime number larger than n, $a \in \{1, \ldots, p-1\}$ is drawn uniformly at random, and $b \in \{0, \ldots, p\}$ is drawn uniformly at random. Then h is a universal hash function.

The proof of Theorem 27.3 is given as exercise 27.2. Here we will assume Theorem 27.3 and focus on its application to the heavy hitters problem.

27.3 Hash tables with chaining

As a first example of hasing, we consider hash tables that use linked lists to handle collisions.The basic framework is as follows. We have n distinct keys $k_1, \ldots, k_n$, from a universe of integers $\{1, \ldots, U\}$. We allocate an array $A[1..m]$ of size m. (Eventually we will set $m = O(n)$, but for the moment we leave it as a variable to explore the tradeoffs between smaller and larger m.)

We randomly construct a hash function $h : [U] \to [m]$. Here we analyze the setting where h is a universal hash function, but later we will also explore stronger notions of independence. Exercise 27.4 explores the setting where h is an ideal hash function.

We hash the n keys into A. At each slot $A[i]$, we build a linked list over all the keys k_j such that $h(k_j) = i$. To find a key k, we go to the linked list stored at $A[h(k)]$, and scan the linked list looking for key k. A high level diagram of the scheme is given in Fig. 29.2. Clearly, the running time of each `get` and `set` will be proportional to the length of the list at the hashed array index. Thus most of our analysis will focus on the lengths of these lists.

We first recall the definition of a *universal hash function.*

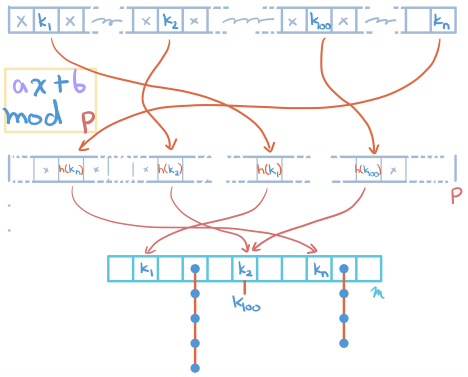


Figure 27.1: Hash tables with universal hashing and chaining.

Definition 27.4. *A randomly constructed function $h : [n] \to [m]$ is* universal *if, for any two indices $i_1 \neq i_2$, we have*

$$\mathbf{P}[h(i_1) = h(i_2)] = \frac{1}{m}.$$

We also remind the reader that a universal hash function can be constructed as a random function of the form $h(x) = (ax + b \mod p) \mod m$, where p is a prime number larger than the maximum possible key.

Now we present the expected running time of a hash table with chaining and universal hash functions, as a function of m and n. We encourage the reader to attempt the proof themselves.

Theorem 27.5. *Consider chaining with n keys, an array $A[1, ..., m]$, and a universal hash function $h : [U] \to [m]$. Then each* `get` *and* `set` *takes $O(1 + n/m)$ time in expectation. In particular, for $m = O(n)$, hash tables with chaining takes $O(n)$ total space and $O(1)$ time per operation in expectation.*

Proof. The time to insert a key k is proportional to the number of collisions with k (plus $O(1)$). The expected number of collisions

$$\mathbf{E}[|k' : h(k') = h(k)|] \overset{(a)}{=} \sum_{k' \neq k} \mathbf{P}[h(k') = h(k)] \overset{(b)}{=} \sum_{k' \neq k} \frac{1}{m} \overset{(c)}{=} \frac{n-1}{m}$$

Here (a) is by linearity of expectation. (b) is by universality. (c) is because there are $n - 1$ other keys. □

`hashed-counters(`$\epsilon > 0$`)`

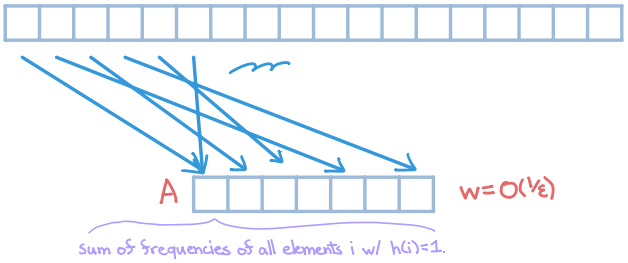
1. Allocate an array of size $A[1..w]$ for $w = \lceil 2/\epsilon \rceil$
2. Sample a universal hash function $h : \{1, \dots, n\} \to \{1, \dots, w\}$
3. For each item e in the stream
   A. $A[h(e)] \leftarrow A[h(e)] + 1$

Figure 27.2: Hashing into a $O(1/\epsilon)$ counters.

27.4 Using hashing to approximate frequencies

Let us now return to the frequency estimation problem. We have elements from the set $[n]$ arriving in a stream. We assume we know n *a priori* but not the length of the stream, which might as well be infinite. In the analysis, we imagine pausing the stream at a fixed point in time after m elements have arrived, and analyze the algorithm at that point in time.

The goal is to estimate the absolute frequency of each element up to an (ϵm)-additive factor. The crux of the problem is that total space usage should be (more or less) independent of the length of the stream, m, or the number of elements, n. We mentioned briefly above that if we knew the heavy hitters, then we could just maintain a counter for each one. Since there are at most $1/\epsilon$ heavy hitters, this approach would satisfy our space constraints. Of course we do not know the heavy hitters. In the following, we will use hash functions to, in effect, *guess* the heavy hitters.



We first create an array of counters $A[1..w]$ with $w = \lceil 2/\epsilon \rceil$ entries. Note that $2/\epsilon$ is extremely small compared to the total length of the stream, or the distinct number of keys. We also sample a universal hash function $h : \{1, \dots, n\} \to \{1, \dots, w\}$. For each element e presented by the stream, we increase $A[h(e)]$ by 1. In turn, for each element e, we treat $A[h(e)]$ as an estimate for f_e. See Fig. 27.2 for pseudocode.

$A[h(e)]$ never underestimates f_e, and the hope is that it does not overestimate f_e by too much. The risk of error comes from other elements' frequencies possibly adding more than ϵm to $A[h(e)]$. Here the intuition is that the "noise" coming from other frequencies is spread out by the hash function over $2/\epsilon$ entries, so we would only expect $\epsilon m/2$ error for each element e. To translate "expected error" to "probability of error", we use Markov's inequality, as follows.

Lemma 27.6. *For each element e, with probability $\geq 1/2$, we have*

$$f_e \leq A[h(e)] \leq f_e + \epsilon m.$$

Proof. We have $A[h(e)] \geq f_e$ always because $A[h(e)]$ is a sum of frequencies of elements with hash code $h(e)$, which of course includes e. The expected additive error is bounded above by

$$\mathbf{E}[A[h(e)]] - f_e \overset{\text{(a)}}{=} \sum_{d \neq e} f_d \, \mathbf{P}[h(d) = h(e)] \overset{\text{(b)}}{\leq} m/w \leq \frac{\epsilon}{2} m. \tag{27.1}$$

Here (a) is by linearity of expectation. (b) is because h is universal. Now we have

$$\mathbf{P}[A[h(e)] \geq f_e + \epsilon m] \overset{\text{(c)}}{\leq} \mathbf{P}[A[h(e)] - f_e \geq 2 \, \mathbf{E}[A[h(e)] - f_e]] \overset{\text{(d)}}{\leq} \frac{1}{2}$$

Here (c) plugs in the inequality obtained in (27.1). (d) applies Markov's inequality, where we note that $A[h(e)] - f_e \geq 0$. □

Given that it is impossible to track frequencies exactly in sublinear space, it is suprising that we can now count every element's frequency with extremely small space, *sometimes and with small additive error*. The algorithm, including the construction of the universal hash function, is extremely simple.

`count-min`($\epsilon > 0,\ \delta > 0$)

1. build $d = \lceil \log 1/\delta \rceil$ instances $(A_1, h_1), \ldots, (A_d, h_d)$ of `hashed-counters`(ϵ) over the stream
2. to query an element e:
   A. return $\min_{i=1,\ldots,d} A_i[h_i(e_i)]$

Figure 27.3: The `count-min` data structure

27.5 Amplification

Section 27.4 shows how to estimate each element with fairly small error with constant probability of error. Our goal now is to reduce the error probability enough to even take the union bound over all of the elements, and thus estimate all frequencies up to ϵm-additive error.

The idea is to use repetition, and one analogy is coin tossing. The goal is to flip enough coin tosses to get at least one heads with very high probability. With one coin toss, the probability that it is tails is $1/2 = .5$. With two coin tosses, the probability that both come up tails is still $1/4 = .25$. But with 100 coin tosses, the probability that all 100 coin tosses come up tails is $1/2^{100} \approx .00000000000000000000000000007886...$.



The point is that independent trials magnify the probability of success exponentially. For a specified probability of error $\delta \in (0, 1)$, the algorithm `count-min` below makes $\lceil \log 1/\delta \rceil$ independent instances of `hashed-counters`(ϵ). For each element e, it uses the minimum estimate over all of the instances of `hashed-counters`. The overall data structure fails for an element e only if *every* instance of `hashed-counters` fails, which by the analogy with coins, is exceedingly unlikely.

We now have the following smaller error probability for each element e.

Lemma 27.7. *For each element e, with probability $\geq 1 - \delta$, we have*

$$\min_{i=1,\ldots,d} A_i[h_i(e)] \leq f_e + \epsilon m.$$

Proof. We have

$$\mathbf{P}\left[\min_{i=1,\ldots,d} A_i[h_i(e)] > f_e + \epsilon m\right] \overset{(a)}{=} \prod_{i=1}^{d} \mathbf{P}[A_i[h_i(e)] > f_e + \epsilon m] \overset{(b)}{\leq} \frac{1}{2^d} \leq \delta.$$

Here (a) is by independence of each $A_i[h_i(e)]$. (b) is by Lemma 27.6. □

For δ set to a polynomial of $1/n$, the probability of error becomes low enough to take a union bound over all elements in $[n]$, as follows.

Theorem 27.8. *Given a stream of elements from the range* $[n]$, `count-min(`ϵ`,`$1/n^2$`)` *has the following guarantee at any fixed point in the stream.*

Suppose m *elements have been presented in the stream. With probability at least* $1 - 1/n$, `count-min(`ϵ`,`$1/n^2$`)` *overestimates the total frequency of each element with additive error at most* ϵm *and total space* $O(\log(n)/\epsilon)$.

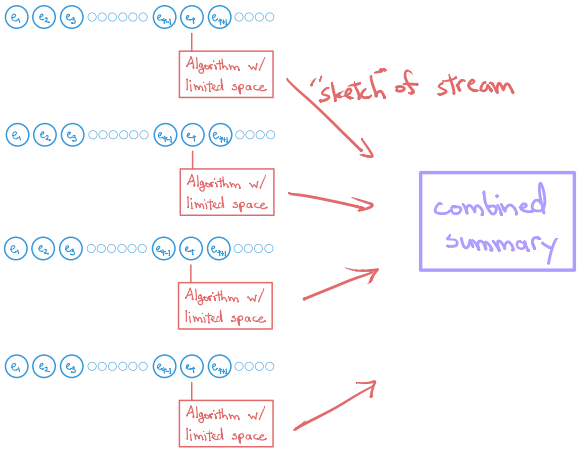
Proof. By Lemma 27.7, we have probability of error $\leq 1/n^2$ for each element e. Taking the union bound over all n elements in the stream, we have probability of error $\leq 1/n$. □

Remark 27.9. More precisely, the space usage of `count-min(`ϵ`,`$1/n^2$`)` is that of $O(\log(n)/\epsilon)$ counters. Here we assume each counter takes $O(1)$ space for simplicity.

27.6 Extensions

27.6.1 Crossing streams

One can extend the streaming model to multiple streams in the following *distributed* model of computation. Here we have several streams simultaneously, each served by an algorithm using sublinear space. The goal is to solve the heavy hitters problem over the *combined streams*.



`count-min` has the convenient property of being a *sketch*. To handle multiple streams, we have an instance of `count-min` for each stream arranged so that they are *all using the same hash functions*. To combine their results, we simply sum up the arrays A_i of hashed sums entry-wise. The result is an instance of `count-min` over the combined streams.

27.6.2 Turnstile streams

Consider the more general model where each item in the stream consists of an element e and a value Δ, signifying that we should increase the frequency count for e, f_e by Δ. Δ is allowed to be negative, with the restriction that the frequency f_e of each element (which is now the sum of $\Delta's$ for that element) remains nonnegative. This model is sometimes called a "turnstile stream", in the sense that a turnstile counting the number of people in an amusement park is always nonnegative because each decrease corresponds to a person who entered the park earlier.

`count-min` adapts immediately to turnstile stream, by simply adding Δ to $A_i[h_i(e)]$ for each instance (A_i, h_i) of `hashed-counters`. The additive error is now ϵ times the sum of all Δ's in the stream.

27.7 Takeaways

- There are many basic and useful problems – heavy hitters with sublinear space being on of our first examples – that are too difficult or even impossible to compute exactly and deterministically. Instead we consider *randomized approximation algorithms* that are potentially more scalable. This requires *quantitative analysis* to address the approximation factor in addition to algorithm design.

- `count-min-sketch` uses *hashing* to try to distribute the heavy hitters across an array. It does not know which are the heavy hitters, but relies on *randomization* to separate the heavy hitters (most of the time) in an *oblivious* fashion.

- *Ideal hash functions*, while easy to reason about, are prohibitively expense. Luckily, weaker hash functions with *limited randomness* often suffice, and are easily constructed. `count-min-sketch` requires only *universal hash functions*. Universal hash functions can be implemented very easily.

- *Linearity of expectation*, combined with universal hash functions, implies that the noise seen by a particular element is evenly spread out *on average*. *Markov's*

inequality allowed us to argue that the noise encountered by an element is close to the average, most of the time.

- `count-min` *amplifies* the probability of success by taking the minimum over many independent trials. A particular element is miscounted if and only if *all* independently trials miscount the element, which happens with vanishingly small probability.
- The error probability drops so rapidly that we can apply the *union bound* over all of the elements after just $O(\log n)$ trials.
- `count-min` does not give *unbiased estimates* of the counters. Instead, `count-min` tries to be *within a prescribed error with high probability*. It is *consistent*. It is more important to be *consistent* then *unbiased*, since we can (psychologically) adjust for the bias. Many real-world apparatus are designed on this principle.

27.8 Additional notes and references

The `count-min` data structure is from [CM05]. Additional notes can be found in [Nel20; Che14].

Spring 2026 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2025 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2023 lecture materials. Click on the links below for the following files:

- Handwritten .pdf prepared before the lecture (and .note file).
- Handwritten .pdf annotated during the presentation (and .note file).
- Recorded video lecture.

Fall 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

27.9 Exercises

Exercise 27.1. Let $h : [n] \to [k]$ be *any* fixed function.

1. Prove that the number of collisions is
$$\geq \frac{n(n-k)}{2k}$$
2. Show that the above inequality is tight when k divides n.

Exercise 27.2. Show that the construction given in Section 27.2 is indeed a universal hash function, using the steps listed below.

To recall the construction, we randomly construct a function $h : [n] \to [k]$ as follows. First, let p be any prime number $> n$. Draw $a \in \{1, \ldots, p-1\}$ uniformly at random, and draw $b \in \{0, \ldots, p-1\}$ uniformly at random. We define a function $h(x)$ by

$$h(x) = ((ax + b) \mod p) \mod k.$$

1. Let $x_1, x_2 \in [n]$ with $x_1 \neq x_2$, and let $c_1, c_2 \in \{0, \dots, p-1\}$ with $c_1 \neq c_2$. Show that the system of equations

$$\begin{aligned} ax_1 + b &= c_1 \mod p \\ ax_2 + b &= c_2 \mod p \end{aligned}$$

uniquely determines $a \in \{1, \dots, p-1\}$ and $b \in \{0, \dots, p-1\}$.[3]

- Step 1 implies that the map $(a, b) \mapsto (ax_1 + b \bmod p, ax_2 + b \bmod p)$ is a bijection between $\{1, \dots, p-1\} \times \{0, \dots, p-1\}$ and $\{(c_1, c_2) \in \{0, \dots, p-1\} : c_1 \neq c_2\}$.

2. Let $x_1, x_2 \in [n]$ with $x_1 \neq x_2$, and let $c_1, c_2 \in \{0, \dots, p-1\}$ with $c_1 \neq c_2$. Show that

$$\mathbf{P}[ax_1 + b = c_1 \mod p,\ ax_2 + b = c_2 \mod p] = \frac{1}{p(p-1)}.$$

(Here the randomness is over the uniformly random choices of a and b.)

3. Fix $x_1, x_2 \in [n]$ with $x_1 \neq x_2$, and $c_1 \in \{0, \dots, p-1\}$. Show that

$$\sum_{\substack{c_2 \in \{1,\dots,p\} \\ c_2 \neq c_1 \mod p \\ c_1 = c_2 \mod k}} \mathbf{P}[ax_1 + b = c_1 \mod p,\ ax_2 + b = c_2 \mod p] \leq \frac{1}{pk}.$$

- The LHS represents $\mathbf{P}[ax_1 + b = c_1 \mod p \text{ and } h(x_2) = h(x_1)]$.[4][5]

4. Finally, show that $\mathbf{P}[h(x_1) = h(x_2)] \leq \frac{1}{k}$.

Exercise 27.3. Let $h : [n] \to [\ell]$ be an ideal hash function, with $\ell \geq n$. What is the *exact probability* that h has no collisions (i.e., h is injective)?

Exercise 27.4. Consider the particular case of hash tables with chaining with $k = n$ and an *ideal* hash function $h : [n] \to [n]$. Let $A[1..n]$ be the cells of the hash table.

[3] Here it is helpful to know that division is well-defined on the set of integers modullo p when p is prime. More precisely, "a/b" is defined as the unique integer c such that $bc = a$.

[4] Here we note that for $x_1 \neq x_2$, $ax_1 + b = ax_2 + b_2 \mod p$ iff $a = 0$.

[5] *Hint:* You may want to show that the number of values $c_2 \in [p]$ such that $c_1 = c_2 \mod k$ is $\leq \frac{p-1}{n}$.

1. Consider a particular array slot $A[i]$. Show that for $\ell \in \mathbb{N}$, the probability that $A[i]$ has $\geq \ell$ items hashed to it is

$$\mathbf{P}[\text{at least } \ell \text{ items being hashed to } A[i]] \leq \frac{1}{\ell!}.$$

2. Show that, with probability of error $\leq 1/n^2$, the maximum length is at most $O(\log(n)/\log\log n)$.[6]

Exercise 27.5. The goal of this exercise is to show how to get constant time access for n keys with $O(n)$ space, using only universal hash functions. We will require the following fact that we ask you to prove.

1. Let $h : [n] \to [k]$ be a *universal* hash function. Show that for $k \geq n^2$, h has no collisions with probability $\geq 1/2$.

Now we describe the data structure. We first allocate an array $A[1..n]$ of size n. We have one universal hash function h_0 into $[n]$. If we have a set of (say) k collisions at an array cell $A[i]$, rather than making a linked list of length k, and we build another hash table, with a new universal hash function h_i, of size k^2, with no collisions (per part 1). (We may have to retry if there is a collision.) If the total size (summing the lengths of the first arrray and each of the second arrays) comes out to bigger than (say) $5n$, we try again.

2. For each $i = 1, \ldots, n$, let k_i be the number of keys that hash to the ith cell. We have

$$(\text{sum of array sizes of our data structure}) \leq n + \sum_{i=1}^{n} k_i^2.$$

   Show that[7]

$$\sum_{i=1}^{n} k_i^2 \leq n + O(\text{total \# of collisions (w/r/t } h_0)).$$

3. Show that

$$\mathbf{E}[\text{total \# of collisions (w/r/t } h_0)] \leq n/2.$$

[6]The simple lower bound of $\ell! \geq (\ell/2)^{\ell/2}$ may be helpful. It is implicit in the $O(\cdots)$ notation that your bound need only hold for n sufficiently large.

[7]Here a "collision" is an unordered pair of keys with the same hash. The $O(\cdots)$ means you can choose whatever constant you find convenient; 2 is possible.

4. Show that

$$\mathbf{P}[(\text{sum of all array sizes}) > Cn] \leq 1/2$$

for some constant $C > 0$. ($C = 5$ is possible.)

Taken together, steps 1 to 3 above show that this approach will build a "perfect" hash table over the n keys in $O(n)$ space with probability of success at least $1/2$, using only universal hash functions. Even if it fails to work, we can then keep repeating the construction until it succeeds. This approach works better in *static settings*, when the set of keys is fixed.

Exercise 27.6. Let $x_1, \ldots, x_n$ be n numbers drawn independently and uniformly at random from the range $[0, 1]$. Obviously they can be sorted in $O(n \log n)$ time with `merge-sort` or `quick-sort`. Design and analyze an algorithm sorting these random numbers in linear time in expectation.

Exercise 27.7. The `count-min` data structure allows us to estimate the relative frequency of each element up to an ϵ-additive factor with probability of error $\leq 1/\operatorname{poly}(n)$ with $O(\log(n)/\epsilon)$ space. [8] The original motivation, however, was to also obtain a list of ϵ-heavy hitters. Design and analyze an algorithm that maintains a list of elements, with at any *particular* point in time,[9] with probability of error $\leq 1/n^2$:

1. Contains all of the ϵ-heavy hitters.
2. Only includes $(\epsilon/2)$-heavy hitters.

Your space usage should be comparable to the space used by the `count-min` data structure.[10]

Additional remark. The question asks for *one data structure* that satisfies *both the criteria* simultaneously. That is, you should maintain a list S that (a) contains all ϵ-heavy hitters, *and* (b) only includes $(\epsilon/2)$-heavy hitters. The tricky part is that

[8] Here the elements are integers from $[n] = \{1, \ldots, n\}$, where n is known, and $\epsilon \in (0, 1)$ is an input parameter.

[9] To clarify, what we mean by "particular point in time" is as follows. You have a data structure that is processing data over time. Suppose we suddenly paused the stream and asked you to report your list of heavy hitters. Your algorithm should succeed then and there with probability of error $\leq 1/n^2$. For this criteria, you do not need to know the length of the stream.

[10] You may want to use the `count-min(`ϵ`,`δ`)` data structure as a black box, but you should be clear about your choice of parameters ϵ and δ.

`count-min` only approximates the frequencies. You may want to account for the fact that an instance of `count-min(`ϵ`,`δ`)` may overestimate the relative frequency of an element by as much as ϵ, which can make a very infrequent element look like an ϵ-heavy hitter.

Exercise 27.8. In this exercise, we develop a refined analysis that can reduce the additive error substantially in (arguably realistic) settings when the stream is dominated by heavy hitters.

Let S denote the sum of frequency counts of all elements that are *not* ϵ-heavy hitters:

$$S = \sum_{e:p_e<\epsilon} f_e.$$

Note that $S \leq m$, and S might be much less than m when the stream is dominated by heavy hitters.

Show that, by increasing the parameter w (in the `count-min` data structure) by a constant factor, and still using universal hash functions, one can estimate the frequency of every element with additive error at most ϵS with high probability in $O(\log(n)/\epsilon)$ space.[11]

Exercise 27.9. Consider the streaming model where we have elements $e_1, e_2, \ldots$ presented one at a time by a stream. A natural task is to sample a fixed number of elements uniformly at random from the stream. Usually, sampling (say) 1 item from a set of m elements is easy: randomly generate a number k between 1 and m, and return the kth element form your set. Sampling in streaming is trickier because we cannot hold the entire stream in memory, and don't know the length of the stream.

1. Consider the following randomized streaming algorithm that selects one element s from the stream:

   `sample-one`

   `/*` *m counts the number of elements in the stream so far, and s is the "sample" of 1 element from the stream.* `*/`

   1. $m \leftarrow 0$, $s \leftarrow$ `nil`.
   2. For each element e presented by the stream:
      A. $m \leftarrow m + 1$.
      B. With probability $1/m$:
         1. $s \leftarrow e$.

[11] *Hint*: It might be helpful to think about the special case of $S = 0$.

For $i \in \mathbb{N}$, let e_i denote the ith element in the stream. For $m \in \mathbb{N}$ let s_m denote the value of s after the mth iteration. Show that for all i and m,

$$\mathbf{P}[s_m = e_i] = \begin{cases} 0 & \text{if } m < i \\ 1/m & \text{if } m \geq i. \end{cases}$$

That is, for each m, s_m is a uniformly random element out of $\{e_1, \ldots, e_m\}$.[12]

2. Now let $k \in \mathbb{N}$ be a fixed parameter. (e.g., $k = 3$.) Suppose you want to sample a set of k elements from the stream without replacement. Design and analyze an algorithm generalizing `sample-one` that maintains a sample S of k elements drawn uniformly at random from the stream. That is, for $m \geq k$, your algorithm should have a set S of k elements, where any particular set of k elements is equally likely (i.e., with probability $1/\binom{m}{k}$). For $k = 1$, your algorithm should coincide with `sample-one` above.[13]

Exercise 27.10. This exercise gives a different and interesting application of hashing to string matching. In string matching, we have a long text string $T[1..n]$, and a smaller search string $S[1..k]$, and we want to decide if S occurs in T. For simplicity we assume these are bit strings, but it is easy to generalize to larger alphabets.

The naive approach directly compares $S[1..k]$ to each length-k substring $T[i, ..., i + k - 1]$, and takes $O(nk)$ time. A more sophisticated algorithm due to Knuth, Morris, and Pratt compiles S into a deterministice finite automaton A_S of size $O(k)$, and uses the automaton A_S to search T in $O(n)$ time. Here we take a different approach based on randomization.

Now, a k-bit string $x \in \{0, 1\}^k$ can be thought of as an integer

$$2^{k-1}x_1 + 2^{k-2}x_2 + \cdots + 2x_{k-1} + x_k$$

[12] Fix i. For $m < i$ the probability 0 since e_i hasn't event appeared in the stream. Now, what about $m = i$? What about $m = i + 1$?

[13] One way to frame your analysis is as follows. For $m \geq k$, let S_m denote the (randomized) sample S after m iterations. Prove the following statement by induction on $m - k$:

For all $m \geq k$, and all sets $X \subseteq \{e_1, \ldots, e_m\}$ of k elements,

$$\mathbf{P}[S_m = X] = \frac{1}{\binom{m}{k}}.$$

In our argument, you may have two cases depending on whether or not $e_m \in X$.

between 0 and $2^k - 1$. One might try to compute all the integers for the k-bit substrings $T[1..k], T[2,, k+1], ..., T[n-k+1..n]$ of T and compare each to the integer form of S. But this is really no different than the naive approach of direct comparison, since the integers are k-bits long.

Suppose instead we took these integers modulo a prime p drawn from a range $\{2, \dots, q\}$, for sufficiently large q. Consider the hash function $h : \{0,1\}^k \to \mathbb{Z}_{\geq 0}$ defined by

$$h(x_{1..k}) = 2^{k-1}x_1 + 2^{k-2}x_2 + \cdots + 2x_{k-1} + x_k \mod p$$

for a randomly selected prime number p. h is a *rolling* hash function: as we "shift" the hash function by 1-bit from (say) bits $1, ..., k$ to bits $2, ..., k+1$, we need only update

$$h(x_{2..k+1}) = 2(h(x_{1..k}) - 2^{k-1}x_1) + x_{k+1} \mod p$$

with a constant number of arithmetic operations, instead of $O(k)$ operations to compute it from scratch.

The goal is to use the rolling hash function to design and analyze a fast randomized algorithm for string matching. The problem of course is collisions between distinct substrings, and the probability of collision depends on the random selection of p. The following facts about prime numbers may be helpful:

- By the prime factorization theorem, every integer can be represented uniquely as a product of prime numbers.
- By the prime number theorem, there are $(1 - o(1))n/\ln n$ primes between 1 and n.
- There is a deterministic polynomial time algorithm for verifying if a number is prime, and a faster randomized algorithm that succeeds with high probability. In particular, there is a deterministic algorithm that runs in $\tilde{O}(k^6)$ time for k-bit integers.

1. Let $x, y \in \{0,1\}^k$ be two distinct k-bit strings interpreted as integers between 0 and $2^k - 1$. Observe that $x = y \mod p$ iff p divides $|x - y|$, which is again an integer between 0 and $2^k - 1$. Prove that there are at most $\log_2(|x - y|)$ distinct prime numbers dividing $|x - y|$.
2. Suppose p is a random prime number from the range $\{2, \dots, q\}$ for some value q. How large does q need to be to guarantee that p does not divide $|x - y|$ with probability (say) at least $1/2$?

3. Using the observations from the previous 2 problems, design and analyze a randomized algorithm searching for S in T that runs in $O(n \operatorname{poly}(\log k))$ time (the faster the better). (Your algorithm should always be correct, and take $O(n \operatorname{poly}(\log k))$ time in expectation. You can assume that arithmetic operations modulo p takes $O(\log p)$ time.)[14]

Exercise 27.11. It's no easy task to find a good apartment near Purdue.[15] Among other factors, it's gotta be cheap, either close to campus or a bus stop, close to at least two good coffee shops, close to at least one good bar, and possibly cat or dog friendly.

Each Spring there's a big rush where apartments are put on the market and quickly gobbled up. The usual predicament is as follows: you see that a nice apartment is available. Should you take it, or wait for another offering? You really want to get the *best* apartment this year.

We model this problem as follows. There are n apartments, each assigned its own value $x_i > 0$. They are presented in uniformly random order, online and one at a time. With each x_i, you have to take it or leave it, on the spot, before getting to see any of the remaining items. The total number of apartments, n, is known. Your goal is to pick the item i with the maximum value x_i.

The goal is to design and analyze an online algorithm that maximizes the probability of selecting the maximum value apartment. You've committed to the following strategy: for a parameter k to be determined, you will observe but pass on each of the first k apartments. Then you will choose the first apartment that is better than any of these first k apartments.

Design and analyze such an online algorithm, optimizing the choice of k and analyzing the probability of identifying the best apartment.

[14] *Hint:* You might first try to get a $O\left(n \log n \log^{O(1)} k\right)$ time algorithm that succeeds with high probability. (This will still get most of the points.) Getting it down to $O\left(n \log^{O(1)} k\right)$ time is a little trickier.

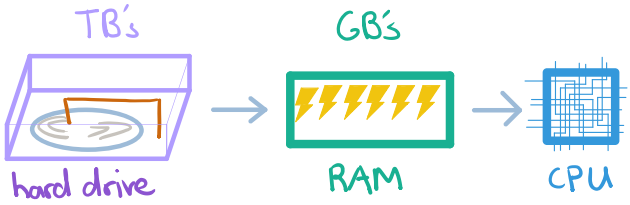
[15] According to the 2023 Campus Student Housing Project (https://www.tippecanoe.in.gov/1224/2023-Campus-Student-Housing-Project):

- From 2015 through the end of 2022, Campus Student Housing Occupancy rates increased by 8.12%. In that same time period, Purdue Undergraduate admissions increased by 28.65%.
- Market Asking Rent for Campus Student Housing increased by 33.6% from 2015 to the end of 2022.

Chapter 28

Online algorithms

28.1 Caching



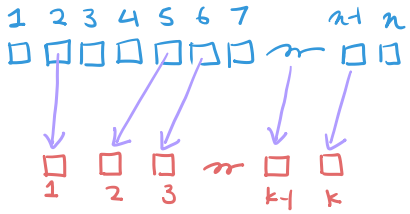
At a very high level, a computer consists of memory holding data and a CPU that computes the data. In one generic step, it loads a few bytes of data from memory into the CPU registers, does some calculations on the registers, and writes the result back into memory.

Memory is supplied by a series of devices with different tradeoffs between memory size and latency. The biggest device is the hard disk, which can hold terabytes of nonvolatile memory but is slow to read and write. Next we have RAM, which holds gigabytes of volatile memory, and is much faster than the the hard drive. Finally the CPU also has its own local (L_1 and L_2) cache, which holds at most a few megabytes of memory, but is extremely fast to access due to its proximity to the CPU.

At the end of the day, the memory is backed by the hard disk, and the RAM and local cache act as faster intermediate layers. When the CPU needs a piece of memory x, it first checks the local cache. If the local cache does not have x, then it checks the RAM. If x is not in RAM, then finally the computer retrieves x from disk. The RAM and the cache are both populated with x as it travels from hard disk to CPU. In the likely event that we read or write x again, it will already be cached in faster memory.

Hitting the hard disk throttles the computation. A good *cache strategy* tries to maximize the odds of the requested data already sitting in fast memory, so we can avoid hitting the disk. The main design decision is in choosing which data to *evict*

from fast memory. In the example, when x was placed in memory, we needed to evict another item y to make room for x. Different strategies for choosing y can make a big difference in performance.



The model. We design and analyze cache strategies in the following simplified model. We assume there are n pieces of data of identical size, and k cache slots that can each hold one piece of data. In general, n is much larger than k. The input is an *online* sequence of data requests, in the form of indices out of $[n]$. Each request i is served from cache, so if i is not already in one of the k slots, then we have to choose which of the k slots to place it in. The general goal is to minimize the number of cache misses.

A salient point is that the requests are made *online*: we have to choose which item to evict without any knowledge of future requests.

The only design decision is in the *eviction policy*: when the requested data x is not in cache, which item y should be evicted to make space for x? Here are a few approaches:

- *Least frequently used (LFU):* Evict the item y that has been accessed the fewest number of times (since being put in cache).
- *Least recently used (LRU):* Evict the item y with the oldest time of last access.
- *Not recently used (NRU), a.k.a. 1-bit LRU:* Whenever an item is accessed, mark it. Evict unmarked items. If all cache items are marked, then remove all marks and try again.

LRU can be seen as a special case of NRU.

Competitive analysis. We typically analyze algorithms from a worst-case point of view. For caching, the worst-case perspective would try to bound the total number of cache misses. In the online model, however, the worst-case is unbounded: the adversary can always request an item out of cache and force a cache miss.

So maybe there is no good cache strategy because the data requests are simply impossible to cache. But *if there is a good cache strategy, then we can find a comparably good one?* Suppose that for a fixed sequence of data requests $i_1, i_2, \cdots \in \mathbb{N}$, it was possible to have m cache misses *in hindsight*. Is it possible for us, operating online, to also get m misses? $100m$ misses? mk misses? etc. In competitive analysis, we want to minimize the competitive ratio,

$$\text{competitive ratio} \stackrel{\text{def}}{=} \frac{\text{our misses}}{\text{OPT misses}}$$

where OPT is the minimum number of misses in hindsight.

Least frequently used (LFU). Least frequently used has an unbounded competitive ratio. We leave the proof to the reader as an exercise.

Exercise 28.1. Prove that LFU has unbounded competitive ratio. (That is, for all $L > 0$, give a sequence of requests for which LFU obtains a competitive ratio $\geq L$.)

Not-recently-used (NRU) and least-recently-used (LRU). Next we analyze the not-recently-used and least-recently-used eviction strategies.

Theorem 28.1. *NRU and LRU have competitive ratio at most k.*

Proof. LRU is a special case of NRU, so we only need to prove the bound for NRU. Consider a sequence of requests

$$i_1, i_2, i_3, \cdots \in [n].$$

We split the sequence whenever NRU resets of all of its marks. We call the contiguous subsequence of requests between resets a *run*.

Each run has k distinct items. The first item after a run is distinct from the k distinct items in the run. Thus: *any* eviction strategy must miss either (a) one of the last $k-1$ distinct requests of the run, or (b) the first item after the run. That is, OPT makes at least 1 mistake per run.

On the other hand, NRU makes k mistakes per run. All put together, we have

$$\frac{\text{\# NRU misses}}{\text{\# OPT misses}} \leq \frac{k(\text{\# runs})}{\text{\# runs}} = k,$$

as desired. □

Randomized NRU. Lastly we consider a *randomized* variation of the NRU algorithm. It starts from the NRU framework: marking items as they are accessed, evicting unmarked items, and unmarking all items when all items are marked. The key distinction is that when evicting an unmarked item, it chooses one uniformly at random.

Theorem 28.2. *Randomized NRU has competitive ratio at most $2\ln(k)$ in expectation.*

Proof. As before, consider a sequence of requests

$$i_1, i_2, i_3, \cdots \in [n],$$

and split the sequence into *runs* whenever all the marks are reset. There are k distinct items per run. Of these k items we distinguish two types:

1. "New" items that were not in the previous run.

2. "Repeat" items that were also in the previous run.

We define a potential function Φ equal to the number of items that are in OPT's cache, but not in the NRU cache.

Fix a single run. Let Φ_{IN} and Φ_{OUT} be the value of Φ at the beginning and the end of the run, respectively. Let ℓ be the number of new items in the run; that leaves there are $k - \ell$ repeat items in the run.

Consider OPT. At the beginning of the run, out of ℓ new pages not in the NRU cache, at most Φ_{IN} of them are in OPT's cache. So OPT misses at least $\ell - \Phi_{\text{IN}}$ new pages.

Now consider Φ_{OUT}. Φ_{OUT} is the number of items in OPT's cache but not in the NRU cache at the end of the run. But the NRU cache is filled with the k items in the run. So OPT must have kicked out these Φ_{OUT} items, and incurred Φ_{OUT} cache misses, during the run.

So OPT has at least $\ell - \Phi_{\text{IN}}$ misses, and at least Φ_{OUT} misses. Average these together, and OPT has at least

$$\frac{\ell + \Phi_{\text{OUT}} - \Phi_{\text{IN}}}{2}$$

misses in the run.

Now consider NRU. The NRU cache has ℓ cache misses for the ℓ new items. Consider the repeat items, which occupy $k - \ell$ of the k slots, all unmarked, at the beginning of the run. We claim that for $i = 1, \ldots, k - \ell$, the ith repeat item is missed with probability at most $\frac{\ell}{k-(i-1)}$.

Consider the first repeat item. In the worst case, all ℓ new items come first, and they evict ℓ of the k items uniformly at random. The first item is evicted, hence missed, with probability ℓ/k.

Now consider the ith repeat item. When this request arrives, the cache contains:

(a) (At most) ℓ new items, all marked.

(b) The first $i - 1$ repeat items, all marked.

(c) $k - \ell - (i - 1)$ items out of the $k - (i - 1)$ from the previous run that have not appeared in this run, all unmarked.

Consider the unmarked items in the third category. Of the $k - (i - 1)$ items from the previous run that have not appeared in this run, exactly ℓ have been evicted. By

symmetry, each has an equally likely chance of being evicted. This includes the ith repeat item. Thus the ith repeat item has a $\frac{\ell}{k-(i-1)}$ probability of being evicted, hence missed, when it is requested.

By linearity of expectation,

$$\begin{aligned}\mathbf{E}[\#\text{ misses on repeats}] &= \sum_{i=1}^{k-\ell} \mathbf{P}[\text{missing on the } i\text{th repeat item}] \\ &\overset{(a)}{\leq} \sum_{i=1}^{k-\ell} \frac{\ell}{k+1-i} \leq \ell \ln(k).\end{aligned}$$

To review, in a single run, OPT makes at least $.5(\ell + \Phi_{\text{OUT}} - \Phi_{\text{IN}})$ mistakes, and randomized NRU makes at most $\ell \ln(k)$ mistakes in expectation.

Let L denote the total number of new items over the entire sequence. Over all the requests, the number of OPT misses is at least $L/2$ plus half of the total change in Φ. Initially $\Phi = 0$ and at the end it is nonnegative so the total change is nonnegative. Thus OPT makes at least $L/2$ mistakes.

Meanwhile the randomized NRU has $L \ln(k)$ cache misses in expectation. Thus the *expected* competitive ratio is bounded by

$$\mathbf{E}[\text{competitive ratio}] \leq \frac{L \ln(k)}{L/2} = 2 \ln(k),$$

as desired. □

28.2 Buy-or-rent

Someone without a car can use Lyft or Uber to get around. But after a lot of taxi fares, buying a car becomes more appealing. Should you buy, or continue to rent? It's tricky because you don't entirely know what the future holds.

Another example is with skis. The ski resort rents skis, but its not very cheap. You can also buy skies, which is more expensive than renting, but could save you money in the long run if you ski a lot. Should you rent or buy skis? (IMO its much harder to project future ski use then car use.)

Decisions like this pop up all over a computer. Should you keep a hard drive spinning when not in use? It takes a while to spin up a hard drive from rest. On the other hand it takes energy to keep it spinning and ready. The right decision depends on future data access patterns that you don't know. Something similar can be said for any component with a high startup cost.

For fun we will frame the problem in terms of skis. Suppose it costs k dollars to buy skis, and 1 dollar a day to rent skis. Let ℓ be the total number of days skied, in hindsight. The tricky part is that we don't know ℓ. Every day we decide to buy or rent skis, not knowing if we will ever ski again.

If we knew ℓ, the decision would be trivial:

1. If $\ell \leq k$, then rent every day.
2. If $\ell \geq k$, then buy on the first day.

Thus OPT $= \min\{k, \ell\}$ in hindsight. Our goal is to develop an *online strategy* competitive with OPT.

As a warmup, we challenge the reader to think of a 2-competitive algorithm.

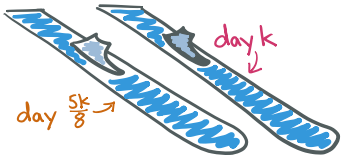
Exercise 28.2. Design and analyze an online, $(2 - o(1))$-competitive algorithm for the ski-rental problem on the previous page.

In fact 2 is optimal for deterministic algorithms:

Exercise 28.3. Prove that $2 - o(1)$ is the optimal competitive ratio for any deterministic algorithm for the ski-rental problem on the preceding page.

Half-skis. As a thought experiment, supposed we relaxed the rules so that you could buy one ski at a time for $k/2$ dollars, and rent 1 ski at a time for $1/2$ dollars. (You still need 2 skis each day). Note that the offline optimal strategy does not change: you should buy or rent (both) depending on if $\ell \geq k$ or not. Thus OPT $= \min\{k, \ell\}$. But here one can get a competitive ratio better than 2. Let us assume k is divisible by 8 for simplicity.

1. On day $5k/8$, buy one ski.
2. On day k, buy a second ski.



For $\ell < 5k/8$, we are optimal since we are only renting. For $\ell = 5k/8$, we pay

$$\frac{5k}{8} + \frac{k}{2} = \frac{9k}{8} = \frac{9}{5}\,\text{OPT}\,.$$

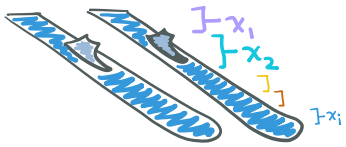
For $\ell = k$, we pay

$$\frac{5k}{8} + \frac{1}{2}\frac{3k}{8} + k = \frac{29}{16}k \leq \frac{29}{16}\,\text{OPT}\,.$$

For ℓ in between $5k/8$ and k, the competitive ratio is only better than at $5k/8$ or at k.

The takeaway is that operating "fractionally" – here we allow for "half" purchases – leads to a better ratio.

Liquid skis. Suppose we pushed the thought experiment further and allowed you to buy or rent arbitrary fractions of a pair of skis. For each day i, let x_i denote the fractional skis bought on day i. Let y_i be the fractional skis rented on day i. We require 1 total ski on each day i:



$$x_1 + \cdots + x_i + y_i \geq 1 \text{ for all } i \in [\ell].$$

Our goal is to minimize the total cost,

$$k(x_1 + \cdots + x_\ell) + y_1 + \cdots + y_\ell.$$

Observe that the optimum policy in hindsight is the same, hence $\text{OPT} = \min\{k, \ell\}$.

Here we will analyze the following heuristic. Let $\delta > 0$ be a parameter TBD. Suppose we commit to spending $1 + \delta$ units per day until we've completely bought the skis. Note that $y_i = 1 - x_1 - \cdots - x_i$ so x_i determines y_i. For day 1,

$$1 + \delta = kx_1 + (1 - x_1) \implies x_1 = \frac{\delta}{k-1}$$

For day 2,

$$1 + \delta = kx_2 + (1 - x_1 - x_2) \implies x_2 = \frac{1}{k-1}(\delta + x_1).$$

In general, on the ith day,

$$1 + \delta = kx_i + (1 - x_1 - \cdots - x_i) \implies x_i = \frac{1}{k-1}(\delta + x_1 + \cdots + x_{i-1}).$$

For the first k days, OPT pays 1 dollar per day, while we pay δ. So for $\ell \leq k$, we have a competitive ratio of $1 + \delta$. We now choose δ to ensure we own a full set of skis after k days, which guarantees that the competitive ratio is $1 + \delta$ for all ℓ. We have

$$\begin{aligned} 1 = x_1 + \cdots + x_k &= \frac{\delta}{k-1}\left(1 + \left(\frac{k}{k-1}\right) + \left(\frac{k}{k-1}\right)^2 + \cdots + \left(\frac{k}{k-1}\right)^{k-1}\right) \\ &= \delta\left(\left(\frac{k}{k-1}\right)^k - \left(\frac{k}{k-1}\right)\right). \end{aligned}$$

Rearranging, we have

$$\delta = \frac{1}{\left(\frac{k}{k-1}\right)^k - \left(\frac{k}{k-1}\right)},$$

and

$$\lim_{k\to\infty} \delta = \frac{1}{e-1} \approx .582.$$

Thus the competitive ratio $1+\delta$ converges to $\frac{e}{e-1} \approx 1.582$.

Random skis. Our improved competitive ratio is artificial since of course we cannot buy skis in arbitrary fractions. We convert the continuous strategy above to a fractional one by *randomized rounding*. The high-level idea is that we commit to buying in one of the first k days, and for $i \in [k]$, interpret x_i as the probability of buying skis on day i. This can be implemented by as follows.

1. Pick $\alpha \in [0,1]$ uniformly at random.
2. Simulate the fractional algorithm, giving values $x_1, x_2, \cdots \in [0,1]$.
3. On day i, if $x_1 + \cdots + x_i \geq \alpha$, buy skis.

The expected cost from buying skis is

$$\mathbf{E}[\text{cost buying}] = k \sum_{i=1}^{\ell} \mathbf{P}[\text{buy on day } i] = k(x_1 + \cdots + x_\ell).$$

The probability we rent on day i is

$$\mathbf{P}[\text{rent on day } i] = \mathbf{P}[\alpha > x_1 + \cdots + x_i] = 1 - x_1 - \cdots - x_i = y_i.$$

Thus the expected cost from renting is

$$\mathbf{E}[\text{cost renting}] = \sum_{i=1}^{\ell} \mathbf{P}[\text{rent on day } i] = y_1 + \cdots + y_\ell.$$

Thus the total expected cost is

$$\begin{aligned}\mathbf{E}[\text{cost}] &= \mathbf{E}[\text{cost buying}] + \mathbf{E}[\text{cost renting}] \\ &= k(x_1 + \cdots + x_\ell) + y_1 + \cdots + y_\ell,\end{aligned}$$

the same cost as our (impractical) fractional solution! Thus we have the same competitive ratio in expectation:

$$\mathbf{E}[\text{competitive ratio}] = 1 + \delta \xrightarrow{k\to\infty} \frac{e}{e-1}.$$

In conclusion:

Theorem 28.3. *There is randomized algorithm for the ski-rental problem with competitive ratio converging to $\frac{e}{e-1}$ for large k.*

28.3 Additional notes and materials

The analysis of deterministic NRU/LRU is from [ST85a]. The randomized NRU algorithm is from [FKL+91]. Additional notes on caching can be found in [Blu00] and [MR95, Chapter 13]. The buy-or-rent algorithm is from [KMM+94].

Lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:

- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Fall 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

28.4 Exercises

Exercise 28.1. Prove that LFU has unbounded competitive ratio. (That is, for all $L > 0$, give a sequence of requests for which LFU obtains a competitive ratio $\geq L$.)

Exercise 28.2. Design and analyze an online, $(2 - o(1))$-competitive algorithm for the ski-rental problem on page 491.

Exercise 28.3. Prove that $2 - o(1)$ is the optimal competitive ratio for any deterministic algorithm for the ski-rental problem on page 491.

Exercise 28.4. Let $\ell < k$. Prove that an LRU cache of size k is $k/(k + 1 - \ell)$-competitive with any cache of size ℓ.

That means, for example, that an LRU cache of size k is 2-competive with any cache of size $k/2$. Some people find this bound more compelling.

You should be able to prove this by a short modification of the argument of the k-competitive bound. It suffices to point out which part of the argument should change, and how.

Exercise 28.5. Suppose you are going to the graduate student social thingy on the 3rd floor of Lawson.[1] You can get to the third floor by either an elevator or the stairs. The elevator takes 15 seconds (once you get in), while the stairs take 2 minutes. Your goal is to get up to the third floor as fast as possible before the donuts are all taken.

You press the button to go up for the elevator. You don't know how long it will take to come down. Do you wait or take the stairs?

1. Suppose you knew how long the elevator would take to arrive. What is the optimal choice, based on this wait?

2. Now suppose you don't know how long the elevator would take. Design a deterministic algorithm that is (15/8)-competitive with the optimal solution (where you know how long the elevator would take).

3. Suppose you have a quarter in your pocket, which lands heads or tails with equal probability. You can toss the coin once every 15 seconds. Design a randomized algorithm with a (slightly) better competitive ratio than the (best) deterministic one from the previous question.[2]

[1] (It used to be on the third floor.)

[2] I actually don't know what the best competitive ratio would be, and I'm interested to see what everyone comes up with.

Chapter 29

Hash tables and linear probing

29.1 Dictionaries

It is difficult to imagine programming without *dictionaries* and *maps*, which are data structures with the following two operations.

1. `set(`k`,`v`)`: Associate the value v with the key k.
2. `get(`k`)`: Return the value associated with the key k (if any).

These two operations form a dead-simple way to store data that can be used in almost any situation. Inevitably all large software systems, however well-planned and structured and object-oriented initially, end up using and passing around dictionaries to organize most of their data. The embrace of dictionaries is taken to another level in `Python` and `Javascript`. These languages provide dictionaries as a *primitive*, and supply a convenient syntax to make them very easy to use. In fact the class object systems in both of these languages are really just dictionaries initialized by some default keys and values and tagged with some metadata. Screenshots of the online documentation for the `Map` interface in `Java` and for `dict` (short for dictionary) in `Python` are given in Fig. 29.1.

We first point out a special case of the dictionary problem that would be ideal. Suppose that there are n keys, and that they are all integers between 1 and n. Then one can simply allocate an array $A[1..n]$ of size n, to hold the n values. Recall that an array consists of n contiguous slots in memory, and the ith slot, $A[i]$, can be retrieved or rewritten in constant time. There is also a real benefit to the fact that the array physically occupies contiguous spots on the hardware. This physical arrangement implies an extremely compact data structure with fewer cache misses.[1]

While the array is ideal for its particular use case, it is not very flexible either. Adding a new key $k = n + 1$, for example, would require rebuilding a new array of

[1] Sometimes, constant factors matter.

Java Map Interface

compact1, compact2, compact3
java.util

Interface Map<K,V>

Type Parameters:
K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:
Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:
AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

public interface **Map<K,V>**

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

The Map interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw UnsupportedOperationException if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an UnsupportedOperationException if the invocation would have no effect on the map. For example, invoking the putAll(Map) method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically NullPointerException or ClassCastException. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the containsKey(Object key) method says: "returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k))." This specification should *not* be construed to imply that invoking Map.containsKey with a non-null argument key will cause key.equals(k) to be invoked for any key k. Implementations are free to implement optimizations whereby the equals invocation is avoided, for example, by first comparing the hash codes of the two keys. (The Object.hashCode() specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying Object methods wherever the implementor deems it appropriate.

Some map operations which perform recursive traversal of the map may fail with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the clone(), equals(), hashCode() and toString() methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Since:
1.2

See Also:
HashMap, TreeMap, Hashtable, SortedMap, Collection, Set

Nested Class Summary

Nested Classes

Modifier and Type Interface and Description

Python Dictionary

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d)` on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use `sorted(d)` instead). To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Figure 29.1: The `Map` interface in `Java` (left) and the built-in `Dictionary` data structure in `Python` (right).

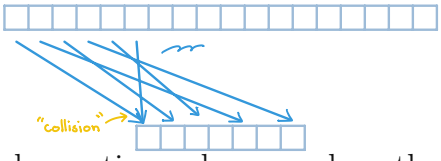
size $n+1$ and copying everything over. It's more problematic when the keys are not a neat contiguous sequence from 1 to n. Perhaps the indices arise implicitly in the bit-string representation of some text, in which case these indices will be spread out over a huge range of possible keys. One cannot allocate an array so big. One could alternatively reindex (i.e., rename) the n arbitrary keys into the slots $1, ..., n$. This works in *static* situations where the keys are presented at the beginning and never change thereafter. But recall that the primary appeal of dictionaries is their flexibility, and their ability to handle all sorts of different keys, without foresight.

A deterministic way to implement dictionaries is via search trees. If the keys are comparable (such as numbers, or strings in alphabetical order), then search trees can organize the data in sorted order in a tree-like data structure. With a well-

designed search tree, searching for a key has roughly the performance of a binary search over a sorted array: $O(\log n)$ time per `get` and `set`. These data structures are often ingenious. Red-black trees use one-bit markers at each node to detect if a subtree has become too "tilted" in one way or another, and rebuilds the tilted portion whenever this occurs. Lazy rebuilding explicitly counts the number of keys in each subtree, and rebuilds an entire subtree when one child subtree becomes much larger than the other (a.k.a. "scapegoat trees" [And89; GR93]). The celebrated *splay tree* data structure by Sleator and Tarjan [ST85b] readjusts itself with every `get` and `set` operation and achieves $O(\log n)$ amortized time (i.e., $O(k \log n)$ time for any sequence of k operations). Another deterministic approach to dictionaries is *tries*, which requires the keys to be (fairly short) bit strings, and uses each successive bit to dictate which direction to go down a binary tree. By compressing long paths in these trees (such as in Patricia tries [Mor68]), these algorithms can be compact and efficient. Now, as clever as these data structures are, they suffer some drawbacks compared to arrays. The $O(\log n)$ query time for search trees is a bit higher then the $O(1)$ time of arrays[2]. They are more complicated to implement, and require a lot of pointer chasing, generating cache misses on the CPU.[3]

We instead consider simpler *randomized* approaches to the dictionary problem; namely, *hash tables*. Hash tables combine the dynamic flexibility of search trees with the raw efficiency of arrays. The only drawback is that the performance guarantees are randomized, which requires a little more sophistication in the analysis. But most people consider the net tradeoff to easily be worth it.

Hash tables are generally based on the following framework. Suppose that there are n keys $k_1, \dots, k_n$ from the set of integers $[U] = \{1, \dots, U\}$, where U is typically incredibly large. One allocates an array $A[1..m]$ of size m (typically $m = O(n)$), and randomly constructs a *hash function* $h : [U] \to [m]$.



Ideally, each key-value pair (k_i, v_i) is stored in the slot $A[h(k_i)]$. The remaining question is what to do when keys *collide*, i.e., when $h(k') = h(k'')$ for two distinct keys k' and k''. There are various ways, to account for collisions, sometimes simple and sometimes clever, such as the following.

1. Make ℓ so large that even a single collision is unlikely. Exercise 27.3 studies how large m needs to be (relative to n) for this to occur.

2. For each slot $j \in [\ell]$ in the hash table, build a linked list of all keys that hash to

[2]Sometimes, log factors matter.
[3]This last point can be helped to some extent by *cache-oblivious* versions.

slot j. We study this first in Section 29.2.

3. For each slot $j \in [\ell]$ in the hash table, build a second hash table (this time following strategy 1) for all keys that hash to slot j. This is the topic of exercise 27.5.

4. Suppose we want to insert a key k. Make *two* hash keys, $h_1(k)$ and $h_2(k)$, and hope that one of these two hash keys is open. More radically, if $h_1(k)$ and $h_2(k)$ are occupied by other keys, see if it is possible to move one of these other keys to its own extra hash key, possibly bumping more keys recursively. This wild approach is called *cuckoo hashing*.

5. Suppose we want to insert a key k and $A[h(k)]$ is occupied. Start scanning the array $A[h(k)+1], A[h(k)+2], \ldots$ until we find the first empty slot, and put k there instead. This approach is called *linear probing*, and will be the topic of the second half of our discussion.

These hash tables have the appeal of potentially running in *constant time*, like an array. Given a key, the hash code $h(k)$ gives a direct index into an array. If the key is there, then we are done. While there may be collisions, we can see in each of the strategies above that $A[h(k)]$ still gets us very "close" to the final location of k. Maybe we have to traverse a short list, hash into a secondary hash table, or continue to scan A until we find our key. For each of these algorithms, some probabilistic analysis is required to understand how much time the "collision-handling" stage will take.

One final remark about the size of hash tables: above, we acted as if we knew *a priori* the number of keys that will be put in the table, and used this to choose the size of the array A. Sometimes, that is the case, but oftentimes it is not, and again the point of dictionary data structures is to not have to plan for these things ahead of time. The easy way to handle an unknown number of keys is by the *doubling trick*. We start with 0 keys and a modestly sized array A; say, of size 64. Whenever the number of keys approaches a constant fraction of the capacity (say, 16), we double the size of the array (to 128). This means we allocate a new array A' with double the capacity, scan the previous array A, and rehash each of the items into A'. A simple amortized analysis shows that the extra effort spent rebuilding is neglible. We note that there are some *distributed* computational settings where one wants to maintain a *distributed dictionary*, and where simply rehashing items becomes expensive and impractical. We refer the reader to a technique called *consistent hashing* that addresses this challenge [KLL+97]. Distributed dictionaries are particularly useful for caching on the web.

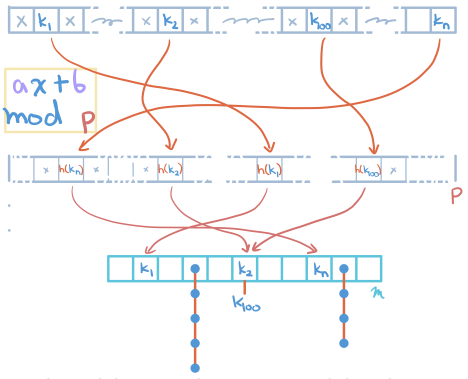


Figure 29.2: Hash tables with universal hashing and chaining.

29.2 Hash tables with chaining

We first consider hash tables that use linked lists to handle collisions. These are maybe the easiest to analyze, and also are most similar in spirit to the `count-min` data structure from Chapter 27.

We recall the basic framework. We have n distinct keys $k_1, \ldots, k_n$, from a universe of integers $\{1, \ldots, U\}$. We allocate an array $A[1..m]$ of size m. (Eventually we will set $m = O(n)$, but for the moment we leave it as a variable to explore the tradeoffs between smaller and larger m.)

We randomly construct a hash function $h : [U] \to [m]$. Here we analyze the setting where h is a universal hash function, but later we will also explore stronger notions of independence. Exercise 27.4 explores the setting where h is an ideal hash function.

We hash the n keys into A. At each slot $A[i]$, we build a linked list over all the keys k_j such that $h(k_j) = i$. To find a key k, we go to the linked list stored at $A[h(k)]$, and scan the linked list looking for key k. A high level diagram of the scheme is given in Fig. 29.2. Clearly, the running time of each `get` and `set` will be proportional to the length of the list at the hashed array index. Thus most of our analysis will focus on the lengths of these lists.

We first recall the definition of a *universal hash function*.

Definition 29.1. *A randomly constructed function $h : [n] \to [m]$ is* universal *if, for any two indices $i_1 \neq i_2$, we have*

$$\mathbf{P}[h(i_1) = h(i_2)] = \frac{1}{m}.$$

We also remind the reader that a universal hash function can be constructed as a random function of the form $h(x) = (ax + b \mod p) \mod m$, where p is a prime number larger than the maximum possible key.

Now we present the expected running time of a hash table with chaining and universal hash functions, as a function of m and n. We encourage the reader to attempt the proof themselves.

Theorem 29.2. *Consider chaining with n keys, an array $A[1, ..., m]$, and a universal hash function $h : [U] \to [m]$. Then each* `get` *and* `set` *takes $O(1 + n/m)$ time in expectation. In particular, for $m = O(n)$, hash tables with chaining takes $O(n)$ total space and $O(1)$ time per operation in expectation.*

Proof. The time to insert a key k is proportional to the number of collisions with k (plus $O(1)$). The expected number of collisions

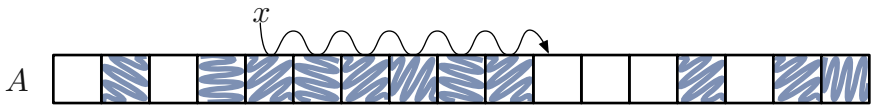
$$\mathbf{E}[|k' : h(k') = h(k)|] \overset{\text{(a)}}{=} \sum_{k' \neq k} \mathbf{P}[h(k') = h(k)] \overset{\text{(b)}}{=} \sum_{k' \neq k} \frac{1}{m} \overset{\text{(c)}}{=} \frac{n-1}{m}$$

Here (a) is by linearity of expectation. (b) is by universality. (c) is because there are $n - 1$ other keys. □

29.3 Linear probing

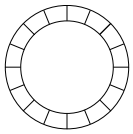
In this section, we explore a different strategy for handling collisions that is arguably more natural: if a key finds its hashed slot already occupied, find the next empty slot in the array and put it there instead.

The hash table, like before, consists of an array $A[1, \ldots, m]$ and a hash function $h : \{1, \ldots, U\} \to \{1, \ldots, m\}$. To insert an item x, we first try to place x at $A[h(x)]$. If $A[h(x)]$ is already occupied, then we instead find the next unoccupied index in the array and place x there instead. (If we reach the end of the array A, then we wrap around to $A[1]$ and continue.)



Since an item x is not necessarily stored at its hashed cell $A[h(x)]$, we carefully use the following terminology. We say that an item *hashes* to a cell $A[i]$ if $h(x) = i$. We say that item x *occupies* a cell $A[i]$ if $A[i] = x$. We stress that an item x hashing into a cell $A[i]$ *does not imply* that x occupies $A[i]$, and that an item x occupying a cell $A[i]$ *does not imply* that x hashes to $A[i]$.

Given two indices $a, b \in [m]$, we define the *interval from a to b*, denoted $[a, b]$, to be the set of indices $\{a, a + 1, \ldots, b \mod m\}$. The "mod m" means that if $b < a$, then we wrap around: $[a, b] = \{a, a + 1, \ldots, m, 1, \ldots, b\}$. One might imagine the array A arranged in a circle rather than a line.



Lemma 29.3. *If an item x occupies cell $\ell \in [m]$, then all of the cells in the interval $[h(x), \ell]$ are occupied.*

Proof. The invariant holds initially with an empty array. We maintain the invariant in the lemma with each insertion, as we insert x in the next unoccupied cell starting from $h(x)$. □

Lemma 29.3 justifies the following lookup procedure. To look up an item x, we first check entry $A[h(x)]$. If item x is not there and the slot is empty, then we conclude the item is not in the array. If the slot $A[h(x)]$ is occupied, but occupied by some item other than x, then we start scanning the array cells to the right of $A[h(x)]$ for either item x or any empty cell. If we find an empty slot before finding x, then by Lemma 29.3, it must be that x is not in the hash table.

To delete an item x, we first find it by the same process as when looking up: starting from $A[h(x)]$, we start scanning the cells until we find x. When we find x at some cell i, we delete x from the cell, but then to restore the invariant in Lemma 29.3, we look for another item to try to fill it. In particular, we start scanning the cells for the first item x_1 with $h(x_1) \leq i$, or else an empty cell. If we find such an item x_1 in a cell i_1, then we put it in the cell i where x was deleted from. We then continue scanning for an item to replace i_1, and so forth.

This hashing scheme is called *linear probing*, and has a special place in the history of computer science. It was analyzed by Donald Knuth in 1962 [Knu63]. Knuth has been called the "father of the analysis of algorithms", and he is credited with formalizing the subject and popularizing O-notation.[4] As Knuth this was the first algorithm he ever formally analyzed[5]. He showed that for ideal hash functions, the expected time of any operation is $O((1/(1-\rho))^2)$ for $\rho = n/m$; in particular, a constant, whenever m exceeds n by a constant factor. This data structure also works very well in practice, even if hash functions in practice are not truly independent. Part of that is owed to the simplicity of the data structure. Scanning an array is extremely fast on hardware, and much faster than chasing pointers along a linked list.

[4] He also invented TeX, solved many problems in compiler design, invented many other important algorithms, wrote *The Art of Computer Programming*, and much more... see for example his wikipedia page.

[5] See for example this interview: https://www.youtube.com/watch?v=Wp7GAKLSGnI.

Post-Knuth, there remained a question of how much independence was required to get a constant running time in expectation. We say that a hash function $h : [U] \to [m]$ is *k-wise independent* for $k \in \mathbb{N}$ if for any k distinct keys $x_1, \ldots, x_k \in [U]$, and any k values $v_1, \ldots, v_k \in [m]$, we have

$$\mathbf{P}[h(x_1) = v_1 \wedge h(x_2) = v_2 \wedge \cdots \wedge h(x_k) = v_k] = \frac{1}{m^k}.$$

That is, the hash values of any fixed set of k (or fewer) keys behaves as if they were produced by an ideal hash function.

(Construct a k-wise independence is more subtle than constructing a universal hash function (like in exercise 27.2), but ultimately can be sampled efficiently in $O(k \operatorname{poly}(\log(n), \log(m)))$ time and represented in $O(k(\log(n) + \log(m)))$ bits. See [Vad12, Chapter 3].)

For what values of k does linear probing, with k-wise independent hash function and $m = O(n)$, run in $O(1)$ expected time? Around 1990, Schmidt and Siegel [SS89; SS90] showed that $O(\log n)$-wise independence sufficed[6]. Then, in 2007, Pagh, Pagh, and Ruzic [PPR09] showed that (just!) 5-wise independence sufficed. This was dramatic progress for arguably the oldest problem in algorithm design. Soon after, [PT16] showed that 4-wise independence was *not* enough. So the answer is 5!

Here we give a simplified analysis of the result of [PPR09] based on ideas in [PT16]. We don't put too much emphasis on the constants, preferring to keep the main ideas as clear as possible. Much better constants can be found in [PPR09] and also the reader is encouraged to refine the analysis themselves. Similar proofs of the constant time bound can be found in [Tho15b; Nel16].

29.3.1 Warmup: Ideal hash functions

Consider linear probing with ideal hash functions, assuming $m \geq 2en$.

Suppose we search (or insert or delete) an item x. Each operation on an item x takes time proportional to the number of consecutive occupied cells starting from $A[h(x)]$. To help analyze this length, we introduce the notion of "runs".



A *run* is defined as a maximal interval of occupied slots. Every occupied cell is contained in a unique run. If an item x is in the hash table, then $A[h(x)]$ is occupied,

[6] Alan Siegel taught *me* algorithms.

and x occupies a cell in the run containing $A[h(x)]$. Therefore each operation with an item x takes time bounded by the length of the run containing x. The key question is: what is the expected size of the run at $h(x)$?

Let R represent the (randomized) run containing $h(x)$ and let $|R|$ denote its length. We have

$$\mathbf{E}\begin{bmatrix}\text{running time}\\ \text{(up to constants)}\end{bmatrix} \leq \mathbf{E}[|R|] = \sum_{\ell=1}^{n} \ell \, \mathbf{P}[|R| = \ell]$$

Fix a particular length ℓ and consider the probability that R has length ℓ. There are ℓ possible intervals at $h(x)$, sliding from the interval $[h(x) + 1 - \ell, h(x)]$ ending at $h(x)$ to the interval $[h(x), h(x) + \ell - 1]$ starting from $h(x)$.

Let I be one of these intervals containing $h(x)$. We have $R = I$ only if exactly ℓ of the n elements are hashed into the ℓ slots in I. (Why?) The latter occurs with probability at most

$$\mathbf{P}[R = I] \leq \binom{n}{\ell}\left(\frac{\ell}{m}\right)^{\ell} \overset{\text{(a)}}{\leq} \left(\frac{ne}{m}\right)^{\ell} \leq 1/2^{\ell},$$

where (a) uses the inequality $\binom{n}{\ell} < (ne/\ell)^{\ell}$.

There are ℓ intervals of length ℓ, and each equals R with probability at most $1/2^{\ell}$. By the union bound,

$$\mathbf{P}[|R| = \ell] \leq \frac{\ell}{2^{\ell}}.$$

Altogether, the expected length of R is

$$\mathbf{E}[|R|] = \sum_{\ell} \frac{\ell^2}{2^{\ell}} = O(1),$$

as desired.

29.3.2 Technical preliminaries: 4-wise independence

Our goal now is to establish the same constant time running time for hash functions with limited independence. This analysis will require a particular *concentration inequality* for 4-wise independence hash functions. Here we only state a definition and the lemma statement; a more detailed discussion and proof is given in Section 29.4.

Definition 29.4. *A collection of n variables $X_1, \ldots, X_n$ is* k-wise independent *if for any k variables $X_{i_1}, \ldots, X_{i_k}$, and values $y_1, y_2, \ldots, y_k$, we have*

$$\mathbf{P}[X_{i_1} = y_1, X_{i_2} = y_2, \cdots, X_{i_k} = y_k] = \mathbf{P}[X_{i_1} = y_1]\,\mathbf{P}[X_{i_2} = y_2] \cdots \mathbf{P}[X_{i_k} = y_k].$$

Thus a k-wise independent hash family is one where the hash values are k-wise independent.

In our applications, we will be interested in 5-wise independent hash functions. In the analysis, we will encounter sums of 4-wise independent random variables. The following lemma will be very important.

Lemma 29.5. *Let $X_1, X_2, \ldots, X_n \in \{0, 1\}$ be 4-wise independent random variables where $\mathbf{P}[X_i = 1] = p$ for each i. Let $\mu = pn$ be the expected sum and assume $\mu \geq 1$. Then for all $\beta > \mu$,*

$$\mathbf{P}\left[\sum_{i=1}^{n} X_i \geq \mu + \beta\right] \leq \frac{4\mu^2}{\beta^4}.$$

To develop some intuition, let us compare the lemma above to Markov's inequality. Let $X_1, \ldots, X_n$ and μ be as in Lemma 29.5. Markov's inequality say that for all $\alpha > 0$,

$$\mathbf{P}[X_1 + \cdots + X_n \geq (1 + \alpha)\mu] \leq \frac{1}{1 + \alpha}. \tag{29.1}$$

Lemma 29.5 says that

$$\mathbf{P}[X_1 + \cdots + X_n \geq (1 + \alpha)\mu] \leq \frac{4}{\alpha^4 \mu^2} \tag{29.2}$$

Compare the RHS of (29.1) with the RHS of (29.2). In (29.2), the upper bound is decreasing in α at a $1/\alpha^4$ rate, compared $1/\alpha$ in (29.1). Moreover, (29.2) is decreasing in the expected value μ at a rate of $1/\mu^2$. That is, the *greater the mean*, the *smaller the probability of deviated from the mean.* This is an example of a *concentration inequality*. We will soon see why this helpful in the analysis of linear probing.

29.3.3 Analysis of linear probing with 5-wise independence.

Theorem 29.6. *Let h be 5-wise independent. For $m \geq 8n$, linear probing takes expected constant time per operation.*

Proof. As discussed in Section 29.3.1, the running time of an operation on an item x is bounded by the length of the run R containing $h(x)$. Let $i = h(x)$, and let R be the run at index i. As before, we have

$$\mathbf{E}\begin{bmatrix}\text{running time}\\ \text{(up to constants)}\end{bmatrix} \leq \mathbf{E}[|R|] = \sum_{\ell=1}^{n} \ell \, \mathbf{P}[|R| = \ell],$$

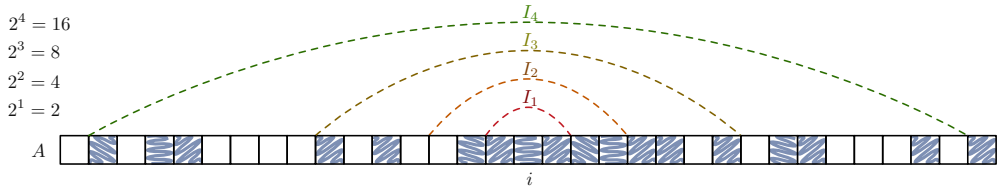
and we want to bound $\mathbf{P}[|R| = \ell]$ for each ℓ. Here we apply a "doubling trick" and group the run lengths by powers of 2:

$$\mathbf{E}\begin{bmatrix}\text{running time}\\ \text{(up to constants)}\end{bmatrix} \leq \sum_{\ell=1}^{n} \ell \, \mathbf{P}[|R| = \ell] \leq \sum_{k=1}^{\lceil \log n \rceil} 2^k \, \mathbf{P}\left[2^{k-1} < |R| \leq 2^k\right]. \tag{29.3}$$

We fix $k \in \mathbb{N}$ and analyze the event that $2^{k-1} < |R| \leq 2^k$. Let

$$I_k = [i - (2^k - 1), i + 2^k - 1]$$

be the interval of length $2^{k+1} + 1$ centered at i.



Observe that all the items occupying R were also hashed into R. Moreover, if R has length $|R| < 2^k$ and contains i, then R is contained in I_k. Thus, for each k, we have

$$\mathbf{P}\left[2^{k-1} < |R| \leq 2^k\right] \leq \mathbf{P}\begin{bmatrix}\text{at least } 2^{k-1} \text{ other}\\ \text{items hash into } I_k\end{bmatrix}.$$

Let

$$\mu \stackrel{\text{def}}{=} \mathbf{E}\begin{bmatrix}\#\text{ other items}\\ \text{hashing into } I_k\end{bmatrix}.$$

Since h is 5-wise independent, conditional on $h(x) = i$, the remaining hash values are 4-wise independent. Each lands in I_k with probability $p = |I_k|/m$. We have

$$\mu = \frac{|I_k| n}{m} \stackrel{\text{(a)}}{\leq} 2^{k-2},$$

where (a) is because $m \geq 8n$. We have

$$\mathbf{P}\left[\begin{pmatrix} \text{\# other items} \\ \text{hashing into } I_k \end{pmatrix} > 2^{k-1}\right] \overset{\text{(b)}}{\leq} \frac{4\mu^2}{(2^{k-1}-\mu)^4} \leq \frac{4\left(2^{k-2}\right)^2}{\left(2^{k-2}\right)^4} \leq \frac{1}{2^{2k-6}}.$$

Here (b) is by Lemma 29.8. Plugging back into RHS(29.3) above, we have

$$\mathbf{E}\left[\begin{matrix}\text{running time} \\ \text{(up to constants)}\end{matrix}\right] \leq \text{RHS(29.3)} \leq \sum_{k=1}^{\lceil \log n \rceil} 2^k \cdot \frac{1}{2^{2k-6}} = 2^6 \sum_{k=1}^{\lceil \log n \rceil} \frac{1}{2^k} \leq 2^6.$$

A constant! □

29.4 4-wise independence

We close the chapter with some probabilistic analysis of k-wise independent random variables. In particular we prove Lemma 29.5, which played a key role in the analysis of linear probing.

29.4.1 Expectations of products of k-wise independent families

Recall the definition of k-wise independent random variables. The following lemma observes that the expected value of a product of (at most) k, k-wise independent random variables is the product of the values.

Lemma 29.7. *Let $X_1, \ldots, X_k$ be k-wise independent random variables. Then*

$$\mathbf{E}[X_1 X_2 \cdots X_k] = \mathbf{E}[X_1]\,\mathbf{E}[X_2] \cdots \mathbf{E}[X_k].$$

Before proving Lemma 29.7, let us give a simple example where k-wise independence matters. Let $X_1, \cdots, X_k \in \{0, 1\}$ where each X_i denotes the outcome of a fair coin toss - 0 for tails, 1 for heads. Then $X_1 \cdots X_k = 1$ if all of the coin tosses come up heads, and 0 otherwise. Consider the following parallel universes.

1. Suppose each X_i was based on a different, independent coin toss. That is, $X_1, \ldots, X_k$ are mutually independent. The probability that k independent coin tosses all comes up heads is $1/2^k$, so $\mathbf{E}[X_1 \cdots X_k] = 1/2^k$.

2. Suppose each X_i was based on the *same* coin toss. That is, $X_1 = \cdots = X_k$; they are certainly *not* k-wise independent. Then the probability that all $X_1, \ldots, X_k = 1$ is the probability of a single coin coming up heads, $1/2$, and so $\mathbf{E}[X_1 \cdots X_k] = 1/2$.

Here there is an exponential gap between independent and non-independent coin tosses.

Proof of Lemma 29.7. We have

$$\begin{aligned}
&\mathbf{E}[X_1 X_2 \cdots X_k] \\
&\overset{(a)}{=} \sum_{y_1, y_2, \ldots, y_k} y_1 y_2 \cdots y_k \, \mathbf{P}[X_1 = y_1,\, X_2 = y_2,\, \ldots, X_k = y_k] \\
&\overset{(b)}{=} \sum_{y_1, y_2, \ldots, y_k} y_1 y_2 \cdots y_k \, \mathbf{P}[X_1 = y_1] \, \mathbf{P}[X_2 = y_2] \cdots \mathbf{P}[X_k = y_k] \\
&= \left(\sum_{y_1} y_1 \, \mathbf{P}[X_1 = y_1] \right) \left(\sum_{y_2} y_2 \, \mathbf{P}[X_2 = y_2] \right) \cdots \left(\sum_{y_k} y_k \, \mathbf{P}[X_k = y_k] \right) \\
&\overset{(c)}{=} \mathbf{E}[X_1] \, \mathbf{E}[X_2] \cdots \mathbf{E}[X_k].
\end{aligned}$$

Here (a) is by definition of expectation[7]. (b) is by k-wise independence. (c) is by definition of expectation, for each X_i. □

29.4.2 A concentration inequality for 4-wise independent sums.

Now we prove Lemma 29.5. Below the claim is stated slightly more generally than in Lemma 29.5.

Lemma 29.8. *Let $X_1, X_2, \ldots, X_n \in \{0, 1\}$ be 4-wise independent variables where for each i, $\mathbf{E}[X_i] = p$. Let $\mu = pn = \mathbf{E}[\sum_{i=1}^n X_i]$. Then for any $\beta > 0$,*

$$\mathbf{P}\left[\sum_{i=1}^n X_i \geq \mu + \beta \right] \leq \frac{\mu + 3\mu^2}{\beta^4}.$$

Proof. We have

$$\mathbf{P}\left[\sum_{i=1}^n X_i \geq \mu + \beta \right] = \mathbf{P}\left[\sum_{i=1}^n X_i - \mu \geq \beta \right] \overset{(a)}{\leq} \mathbf{P}\left[\left(\sum_{i=1}^n X_i - \mu \right)^4 \geq \beta^4 \right] \overset{(b)}{\leq} \frac{\mathbf{E}\left[\left(\sum_{i=1}^n X_i - \mu \right)^4 \right]}{\beta^4}.$$

The key step is (a), where we raise both sides to the fourth power. (b) is by Markov's inequality. *We claim that*

$$\mathbf{E}\left[\left(\sum_{i=1}^n X_i - \mu \right)^4 \right] \leq \mu + 3\mu^2,$$

[7] We are summing over all possible outcomes $(y_1, \ldots, y_k)$ of $(X_1, \ldots, X_k)$, multiplying the value, $y_1 \cdots y_k$, with the probability of the outcome, $\mathbf{P}[X_1 = y_1, \ldots, X_k = y_k]$.

which would complete the proof. We first have

$$\mathbf{E}\left[\left(\sum_{i=1}^{n} X_i - \mu\right)^4\right] = \mathbf{E}\left[\left(\sum_{i=1}^{n}(X_i - p)\right)^4\right]$$

because $\mu = pn$. Now, $\left(\sum_{i=1}^{n}(X_i - p)\right)^4$ expands out to the sum

$$\sum_{i=1}^{n}(X_i - p)^4 + \binom{4}{2}\sum_{i<j}(X_i - p)^2(X_j - p)^2 + \binom{\text{monomials w/ some}}{(X_i - p) \text{ w/ degree 1}}. \tag{29.4}$$

Some examples of the third category would be $(X_1 - p)^3(X_2 - p)$, $(X_1 - p)^2(X_2 - p)(X_3 - p)$, and $(X_1 - p)(X_2 - p)(X_3 - p)(X_4 - p)$. Consider the expected value of each of these categories of monomials.

1. For each i, we have
$$\mathbf{E}\Big[(X_i - p)^4\Big] = p(1-p)^4 + (1-p)p^4p(1-p).$$
2. For each $i \neq j$, we have
$$\mathbf{E}\Big[(X_i - p)^2(X_j - p)^2\Big] \overset{\text{(c)}}{=} \mathbf{E}\Big[(X_i - p)^2\Big]\,\mathbf{E}\Big[(X_j - p)^2\Big] \overset{\text{(d)}}{\leq} p^2(1-p)^2.$$
Here (c) is because of pairwise independence. (d) is because
$$\mathbf{E}\Big[(X_i - p)^2\Big] = p(1-p)^2 + (1-p)p^2 \leq p(1-p).$$
3. Each monomial in the third category has expected value 0. This is because we can pull out the degree 1 term by independence, which has expected value 0. For example,
$$\mathbf{E}\Big[(X_1 - p_1)^3(X_2 - p_2)\Big] \overset{\text{(e)}}{=} \mathbf{E}\Big[(X_1 - p_1)^3\Big]\,\mathbf{E}[X_2 - p_2] \overset{\text{(f)}}{=} 0,$$
where (e) is by pairwise independence, and (f) is because $\mathbf{E}[X_2 - p_2] = 0$.

Plugging back in above, we have

$$\mathbf{E}\left[\left(\sum_{i=1}^{n} X_i - \mu\right)^4\right] = np(1-p) + \binom{n}{2}\binom{4}{2}\Big(p^2(1-p)^2\Big) \leq np + 3(np)^2,$$

as desired. This completes the proof. □

Remark 29.9. The claim would hold even for X_i not identically distributed (as long as they are 4-wise independent and are each in $[0, 1]$). The restrictive assumptions here simplify the exposition and suffice for our applications.

29.5 Takeaways

- Dictionary data structures provide everyday motivation for studying randomization, where hash tables offer simpler and better performance (in expectation) than search trees.
- There are different ways to implement hash tables and they mostly differ in how they handle collisions.
- *Chaining* uses linked lists to handle collisions. It has reasonable performance *in expectation* for universal hash functions, and stronger guarantees when the hash function is more independent.
- *Linear probing* is perhaps the easiest hash table to implement, and scanning an array is hardware-friendly. It had been observed to perform well in practice long before it had been properly analyzed.
- The analysis of linear probing cleverly uses *canonical intervals* (doubling in size) to limit the number of "bad events" we have to avoid, to roughly $\log n$ (per key).
- It turns out that *5-wise independence* is sufficient for linear probing to have $O(1)$ running time in expectation. Interestingly, 4-wise independence is not enough.

29.6 Additional notes and materials

Thorup [Tho15a] describes several families of hash functions with both theoretical and practical considerations. See [Eri17] for additional notes on hashing.

Lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Fall 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

29.7 Exercises

Chapter 30

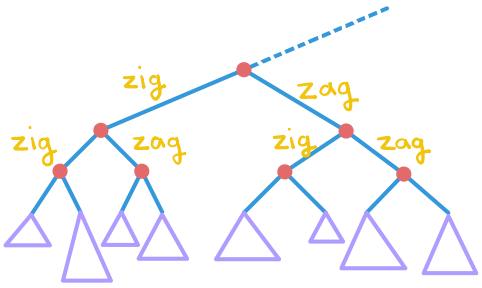
Fun with trees: splay trees, Euler trees, and dynamic connectivity

30.1 Splay trees

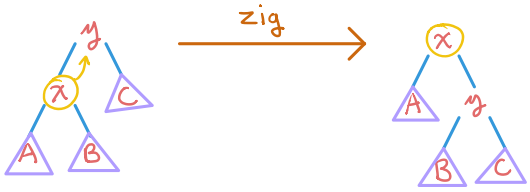
Splay trees are binary search trees that constantly update themselves with every operation. Whenever we search for and find a node, we adjust the tree so that the node we retrieved becomes the root. This way, when we query the same key repeatedly (as happens naturally), although the first operation might be slow, all the subsequent queries will find the key at the root, and will be very fast.

30.1.1 Rotations

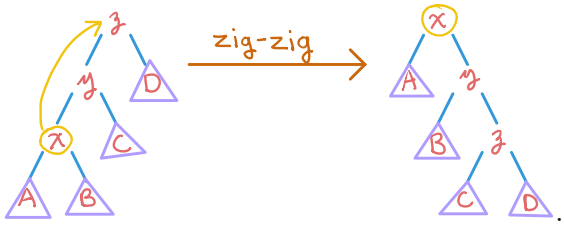
The tree is adjusted via local *tree rotations*. Here it is convenient to label the children and grandchildren by the following "zig-zag" naming scheme.



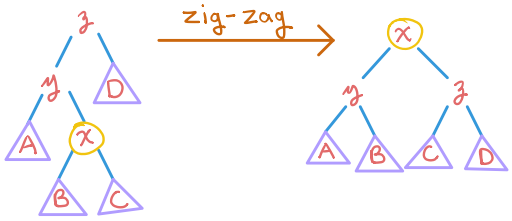
For example, suppose we wanted to rotate the left child to be the root. We call this a "zig". There is really only one natural way to make the left child the root while (a) maintaining the binary search tree invariant and (b) leaving intact all the subtrees that don't contain x, as follows.



As a second example, suppose we want to rotate the left child of the left child to be the root. We call this a "zig-zig". If we put this child at the root, and then arrange the rest to satisfy (a) and (b) above, then we naturally arrive at the following.



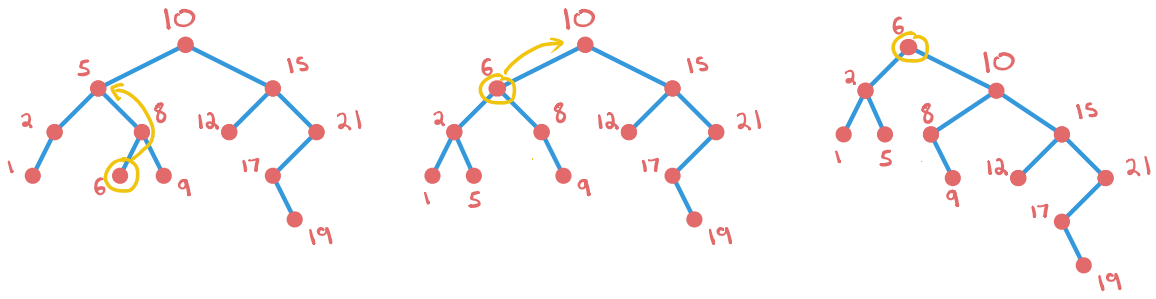
The third example rotates the right child of the left child of the root. This is called a "zig-zag".



Similarly, we have "zag", "zag-zig", and "zag-zag" operations. All $6 = 2+2^2$ operations are given in Fig. 30.1.

30.1.2 Introducing splay

The main subroutine of the splay tree is the `splay` operation. `splay` searches for a node and then makes local rotations until it becomes the root.



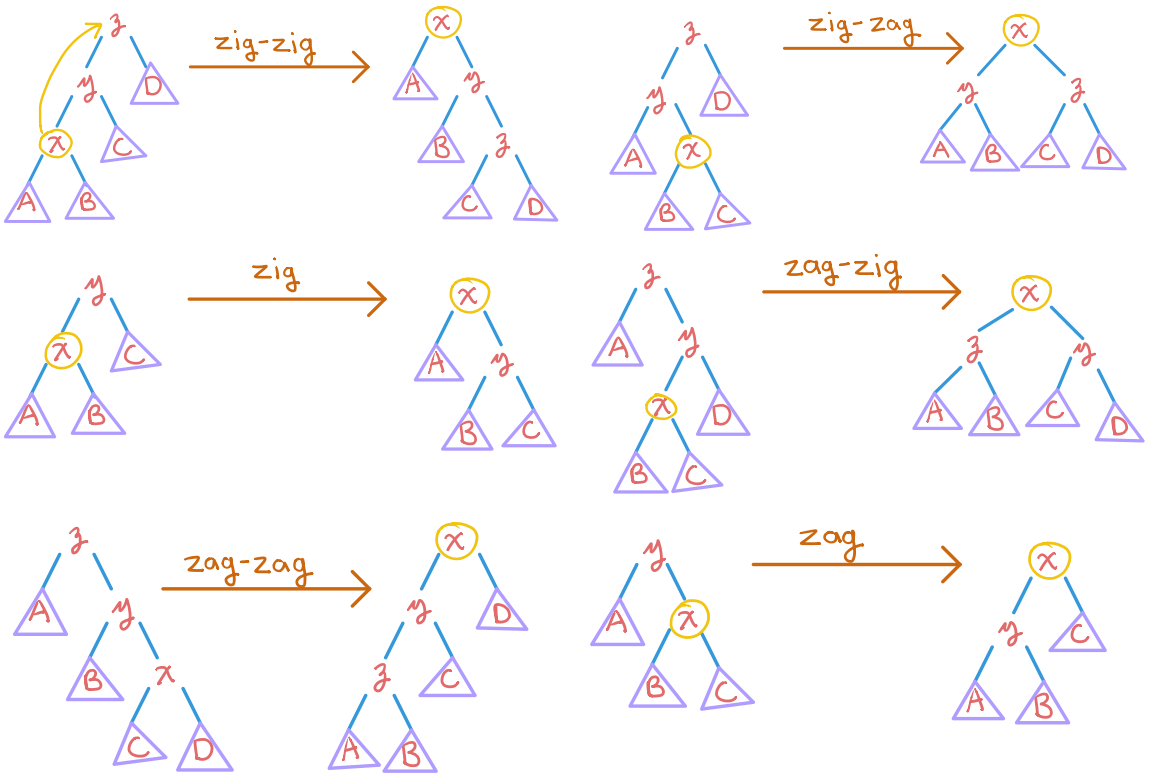


Figure 30.1: All 6 tree rotations.

Above is an example where the node with key 6 is splayed to the top. The first is a zag-zig at the subtree rooted at 5. The second is a zig from the root of the entire tree. Curiously, the tree becomes (subjectively) *less balanced* from splaying 6.

Theorem 30.1. `splay`(x) *takes* $O(\log(n))$ *amortized time.*

30.1.3 Analyzing `splay`: proof of Theorem 30.1

The high level approach is as follows. First we define a potential function. Second we analyze each of the rotation operations w/r/t this potential. Third, we show that the per-rotation bounds add up nicely over an entire call to `access`. Here we will only state the amortized bounds for each rotation, and do the first and third steps of the proof in detail. Afterwards we analyze each rotation separately in subsequent sections, which completes the overall proof.

For each node x, we define We first define potential Φ_x by

$$\Phi_x \stackrel{\text{def}}{=} \log(\#\text{ of nodes in the subtree rooted at } x)$$

We define the total potential $\mathbf{\Phi}$ as the sum of potential of all the nodes,

$$\mathbf{\Phi} \stackrel{\text{def}}{=} \sum_{x \in T} \Phi_x.$$

The local rotations have the following guarantees with respect to the potential functions defined above. Here we only state the bounds and will prove them separately in subsequent sections.

Lemma 30.2. *Consider a single zig-zag (or zag-zig) making a node x the root of a subtree. Then*

$$2 + \Delta\mathbf{\Phi} \leq 2\Delta\Phi_x.$$

Lemma 30.3. *Consider any zig-zig or zag-zag rotation making a node x the root of a subtree. Then*

$$2 + \Delta\mathbf{\Phi} \leq 3\Delta\Phi_x.$$

Lemma 30.4. *Consider any zig or zag rotation making a node x the root of a subtree. Then $\Delta\mathbf{\Phi} \leq \Delta\Phi_x$.*

Suppose that all three lemma's hold holds, and we splay x to the top. We have

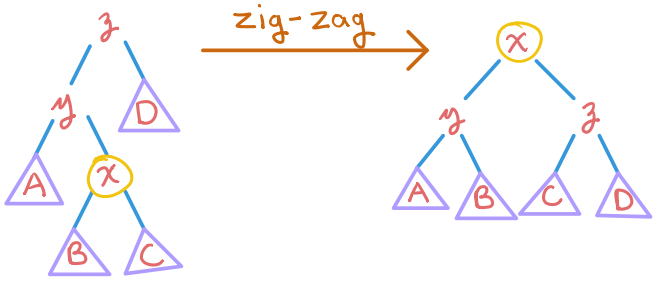
$$2(\text{total } \#\text{ rotations}) + \Delta\mathbf{\Phi} \stackrel{\text{(a)}}{\leq} 1 + 3\Delta\Phi_x \stackrel{\text{(b)}}{\leq} O(\log n)$$

as desired. (a) applies Lemma 30.2 or Lemma 30.3 (where we point out that the inequality $2 + \Delta\mathbf{\Phi} \leq 3\Delta\Phi_x$ is valid for any double rotation), and Lemma 30.4 to the last rotation if necessary. (b) observes that $\Delta\Phi_x \leq \Phi_x \leq O(\log n)$.

It remains to prove Lemma 30.2, Lemma 30.3, and Lemma 30.4. Below we present the proof of Lemma 30.2, for the zig-zag and zag-zig operations. The proofs of the remaining splay tree operations are similar, so their proofs are deferred to Section 30.A at the end of the chapter.

30.1.4 Zig-Zag: Proof of Lemma 30.2

Let us label the relevant subtrees and nodes as follows.



We let $A, \dots, D$ also denote the size of the respective subtrees. We have

$$\Delta\mathbf{\Phi} = \log(1 + A + B) + \log(1 + C + D) - \log(2 + A + B + C) - \log(1 + B + C).$$

We also have

$$\Delta\Phi_x = \log(3 + A + B + C + D) - \log(1 + B + C).$$

Now,

$$\begin{aligned}\log(1 + A + B) + \log(1 + C + D) &\overset{(a)}{\leq} 2\log\left(\frac{2 + A + B + C + D}{2}\right) \\ &= 2\log(2 + A + B + C + D) - 2\end{aligned}$$

by (a) concavity. Thus

$$\begin{aligned}&\Delta\mathbf{\Phi} + 2 \\ &\leq 2\log(2 + A + B + C + D) - \log(2 + A + B + C) - \log(1 + B + C) \\ &\leq 2\Delta\Phi_x,\end{aligned}$$

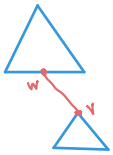
as desired. This completes the proof of Lemma 30.2.

30.2 Dyamic trees, subtree queries, and Euler trees

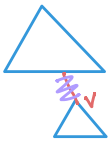
30.2.1 The dynamic tree model and subtree queries

In this section we shift into a different kind of "tree data structure" from binary search trees, called "dynamic trees". Here we model *rooted trees* updated by the user by adding and removing parent edges.[1]

[1]To be clear, in dynamic trees, there are no "keys" that have to be maintained in a particular order, and vertices can have any number of children. We are modeling a rooted tree in the graph algorithms sense, and not a binary search tree in the classical data structure sense.



Link. Given two vertices w and v, where v is the root of a tree, `link(`w`,`v`)` make v a child of w.



Cut. Given a vertex v that is not a root, `cut(`v`)` removes v's parent edge, making v the root of its own tree.

Besides adding and deleting edges, one can also change the root.

Reroot. Given a vertex v, reorient the edges in v's rooted tree to make v the root.

The next query helps vertices identify which tree they are a part of.

Root. Given a vertex v, return the root r of the tree containing v.

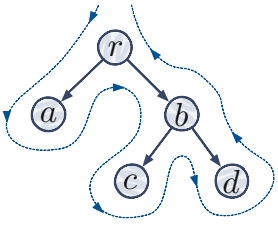
Now, let us suppose that the vertices are associated with auxiliary data such as numerical weights on each vertex, or a list of elements stored at each vertex. As the structure of the tree is updated via `link`/`cut`, we typically want to support aggregate queries over these values. In this section we focus on aggregates over *subtrees*.

Subtree aggregates. Given a vertex v, aggregate a class of values over all the vertices in the subtree rooted at v.

1. Suppose each vertex is associated with a weight. Return the sum of weights of all vertices in the subtree rooted at v.
2. Suppose each vertex is associated with a (possibly empty) list of elements. Return a list concatenating all the lists in the subtree rooted at v.

We mention that there is another query of interest, called *path queries*, which aggregates over all vertices along the path from a vertex v to the root of its tree. This query is addressed in by a data structure called "link cut trees"; see Chapter 31.

30.2.2 Euler-tour trees.



Given an undirected tree T, an *Euler tour* is a walk that traverses the entire tree in each direction exactly once. Every tree has an Euler tour and it is easy to construct one. Root the tree arbitrarily. Starting from the root, pick any child of the root. Go down that edge, and inductively take an Euler tour of that subtree, and come back up to the root. Clearly we use the edge to the subtree once

in each direction. We repeat this for the other children of the root, which together gives an Euler tour of the entire tree.

Euler tours inspired a data structure called *Euler-tour trees* [HK99] that supports subtree aggregation in the dynamic tree model.

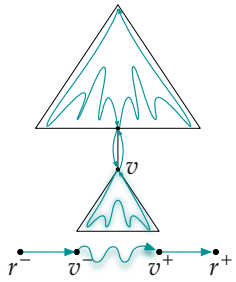
The data structure maintains a fixed Euler tour over the rooted tree. As we follow the Euler tour, write down each vertex every time we visit it. This produces a sequence of vertex labels starting and ending with the root. For example, in the Euler tour above, we have the sequence

$$r,\ a,\ r,\ b,\ c,\ b,\ d,\ b,\ r$$

The first appearance of a vertex v corresponds to the point where we enter the subtree rooted at v. The last appearance of a vertex v corresponds to the point where we leave the subtree rooted at v. Vertices of nested subtrees are nested in the Euler tour as well. For a vertex v, we let v^+ denote the first appearance of v in the Euler tour and let v^- denote the last appearance.

We build and maintain a splay tree[2] over the Euler tour (based on the order of the Euler tour). For each vertex v, we explicitly maintain pointers to the first instance v^+ and the last instance v^- in the splay tree.

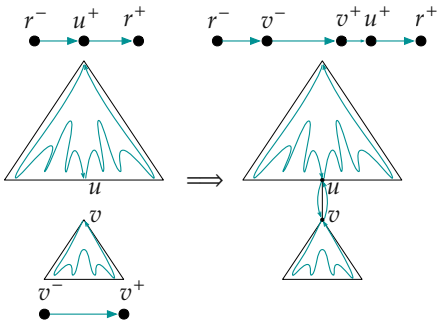
30.2.3 Aggregating over subtrees.



Suppose we want to aggregate (e.g., sum all values) in the subtree rooted at a node v. Euler trees are designed specifically for this application. Aggregating over a subtree rooted at v corresponds to a range query between v^+ and v^- in the corresponding splay tree. As discussed previously, this takes $O(\log n)$ amortized time.

Lemma 30.5. *An Euler tour tree reduces aggregation over subtrees to range queries in a search tree. In particular, with splay trees, most operations take $O(\log n)$ amortized time.*

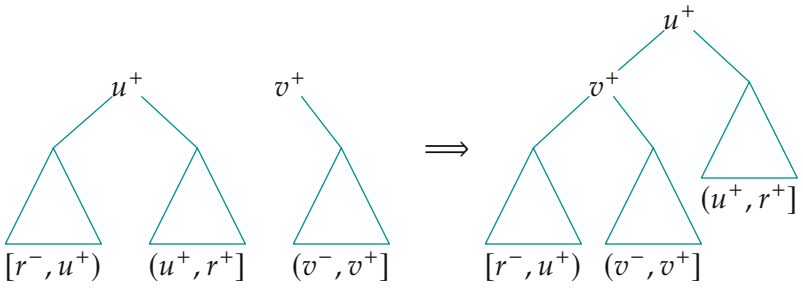
30.2.4 Link.



Suppose we want to link v as a child of the node u, where we assume v is the root of its tree. In terms of the Euler tour, we want to insert the entire Euler tour rooted at v – from v^+ to v^- – just before the

[2]Other balanced binary search trees would also work.

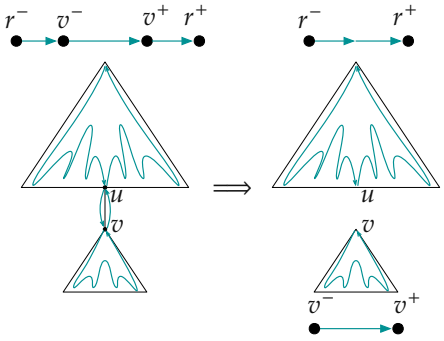
last time we visit u: u^+. To do so, we first splay u^+ and v^+ in their respective trees. Note that v^+ has no left subtree because v is the root. We remove the left subtree of u^+ and make it the left subtree of v^+, and make the tree rooted at v^+ the left subtree of u^+. See the diagram below.



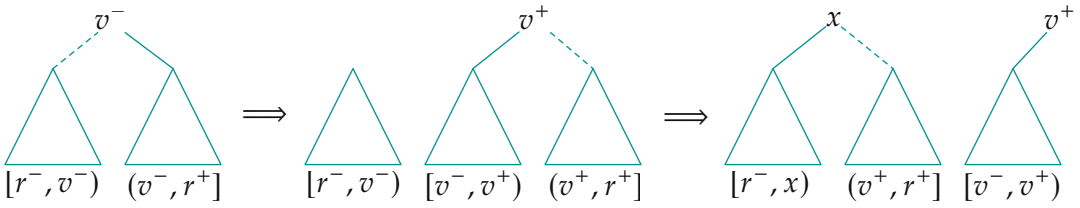
Overall, we made two calls to `splay`, and did some constant work to attach v^+ to u^+. That already gives $O(\log n)$ amortized time. Additionally, combining the subtrees at the end increases u's potential, Φ_u (where we recall the potential functions for splay trees). However, $\Phi_u \leq \log(n)$ no mater what, so this adds at most $O(\log n)$ to the amortized cost. Overall we obtain the following.

Lemma 30.6. *Euler tour trees support* `link` *in* $O(\log n)$ *amortized time.*

30.2.5 Cut.



Suppose we want to cut the parent edge of a vertex v. In terms of the Euler tour, we remove the subsequence between the first and last visit of v from the Euler tour, gluing together the part before v and the part after v. This is easy to do in a splay tree. We splay v^+, which puts the interval $[v^-, v^+)$ in the left subtree of v^+. We disconnect this left subtree. Then we splay v^-, which puts the interval $(v^-, r^+]$ in the right subtree of v. We disconnect that right subtree as well. This gives us three splay trees with the intervals $[v^-, v^+)$, $[v^+, v^-]$, and $(v^-, r^+]$. We then concatenate the first and third intervals as follows. Let x be the last node in $[v^-, v^+)$. We splay x, upon which it has no right subtree, and then attach the splay tree for $(v^-, r^+]$ as a right subtree. See the diagram below.



The running time is bounded by a constant number of `splay`'s. Thus we have the following.

Lemma 30.7. *Euler tour trees support* `cut` *in* $O(\log n)$ *amortized time.*

30.2.6 Root.

An additional type of query that will be useful in the subsequent application is called `root`. Given a vertex v, `root(`v`)` returns the root of the tree containing v. This is the first vertex r^- in the Euler tour of the tree containing v, hence the leftmost node in the underlying splay tree. Finding this vertex and splaying it takes $O(\log n)$ amortized time.

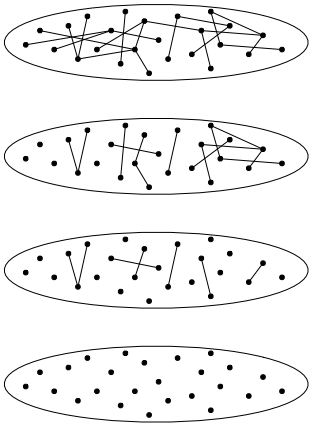
30.3 Dynamic Connectivity

The dynamic graph model and dynamic connectivity Lastly we consider the dynamic graph model. The idea is that we have a graph undergoing small changes such as edge insertions and deletions, and we want to maintain certain information of the graph. Here we will consider the problem of maintaining a spanning forest in an undirected graph.

Let $G = (V, E)$ be an unweighted and undirected graph. Recall that we can build a spanning forest F in G in $O(m + n \log n)$ time. In this section, our goal is to maintain a spanning tree in G as G changes. In each iteration, an edge is either added or deleted from the graph. Computing an entire spanning tree from scratch seems wasteful. Instead we will develop an algorithm

30.3.1 The data structure

Let $L = \lceil \log n \rceil$. We maintain a nested family of subsets of both E and F,



$$E_1 \subseteq E_2 \subseteq E_3 \cdots \subseteq E_L \subseteq E$$

and

$$F_1 \subseteq F_2 \subseteq F_3 \cdots \subseteq F_L \subseteq F,$$

subject to the following invariants.

1. $E_L = E$ and $F_L = F$.
2. For each i, $F_i = F \cap E_i$.
3. For each i, $E_i \subseteq \text{span}(F_i)$.
4. Every component in E_i has size $\leq 2^i$.

We denote the differences between the sets by

$$E'_i \stackrel{\text{def}}{=} E_i \setminus E_{i-1} \text{ and } F'_i = F_i \setminus F_{i-1}$$

for each i. The *level* of an edge e is defined as the lowest index i for which $e \in E_i$; this is also the unique index i such that $e \in E'_i$.

For each level i, and each vertex v, we maintain the list of edges at level i incident to v (i.e., $\delta(v) \cap E'_i$, where $\delta(v)$ denotes the edges incident to v). We create an Euler tree over each component of F_i to support the following queries.

1. *Given the root of a subtree, return any edge $e \in F'_i$ in that subtree, if one exists.*
2. *Given the root of a subtree, return any edge $e \in E'_i$ incident to a vertex in a subtree, if one exists.*

Both of these operations correspond to a "list aggregate", as discussed previously, and take $O(\log n)$ amortized time.

Querying connectivity. The only query is for pairwise connectivity: given u and v, are u and v connected in G? This corresponds to querying for the roots of u and v in their Euler trees, and seeing if they are the same. As discussed in Section 32.2, this takes $O(\log n)$ amortized time.

Insertion an edge. Suppose we want to insert an edge e. If F spans e, then we insert e into the first level i such that F_i spans e. If e is not spanned by F, then we insert e into both F_L and E_L. Note that "inserting" an edge also involves updating the Euler tree at the level the edge is inserted.

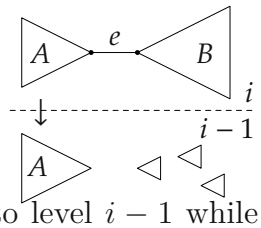
30.3.2 Deleting an edge.

Suppose we want to delete an edge e from G. Suppose this edge is at at level ℓ. We remove e from E_ℓ (and $E_{\ell+1}$, $E_{\ell+1}$, ...) and we remove e from the incidence lists of its endpoints. If e is not in F then we are done.

Now suppose $e \in F$; in particular, $e \in F'_\ell$. We first remove e from F_ℓ (and $F_{\ell+1}, F_{\ell+2}, \dots$). Then we look for an edge to replace it. For each level i from ℓ to L we do the following.

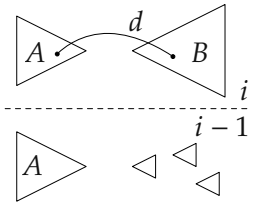
Suppose removing e separated a tree in F_i into two components A and B. We assume that A is the smaller of the two components. Then we have

$$|A| \leq \frac{1}{2}(|A| + |B|) \leq 2^{i-1},$$



so in particular, we can now push all of the edges in A down to level $i-1$ while satisfying invariant 4 above. So that's the first thing we do – we repeatedly query the Euler tree for level i edges in $A \cap E'_i$, pushing each one down to level $i-1$, until there are none left.

At this stage all of A is at level $i-1$ or lower. Now we look at all the level i edges incident to any vertex in A. These edges can be retrieved one at a time from the Euler tree (in $O(\log n)$ amortized time per edge). For each such edge d, we check to see if d is spanned by A. If it is, then we push it down to level $i-1$. If it isn't, then in particular it must connect A to B. That is, d is our replacement edge. We insert d into F_i and all higher forests, and exit the entire procedure.



If we end up processing all of the level i edges without finding a replacement, then we go up to level $i+1$, and repeat. If we end up going through all the levels without a replacement, then we conclude that there is no replacement for e, and that the current forest is a spanning forest.

30.3.3 Analysis

Above we already observed that querying connectivity takes $O(\log n)$ amortized time via the Euler tree. It remains to analyze `insert` and `delete`. Note that `delete` can

`insert(`e`)`

/ Below, when we "insert an edge" at a level i, this also means we update all the relevant data structures. */*

1. If F spans e
   A. Insert e into the first level i such that F_i spans e
2. Otherwise insert e into F_L and E_L.

`delete(`$e = \{a, b\}$`)`

1. remove e from the incident lists of a and b
2. if $e \in F$
   A. let ℓ be the level of e
   // remove e from each forest that contains it
   B. call `cut(`u, v`)` on $F_\ell, \dots, F_L$
   C. for $i = \ell, \ell + 1, \dots, L$
      1. Let A and B be the trees of F_i containing a and b, respectively.
      // We assume $|A| \leq |B|$ (otherwise switch a and b)
      // $|A| \leq \frac{1}{2}(|A| + |B|) \leq 2^{i-1}$, and we can push all of A to F_{i-1}
      2. For each level i edge $f = \{x, y\} \in A \cap F'_i$
         a. Call `link(`x`,`y`)` on F_{i-1} and add f to E_{i-1}
      3. For each level i edge $f = \{x, y\} \in E'_i$ with $x \in A$
         // To facilitate this loop, we maintain
         a. if $y \in B$ *// Replace e with f*
            1. call `link(`x`,`y`)` on $F_i, F_{i+1}, \dots, F_L$ and exit
         b. else *// f is spanned by A and can be pushed to level $i - 1$*
            1. add f to E_{i-1}

have high running times in the worst case if it has to search through many candidate edges for a replacement. So we will use amortized analysis instead.

For each edge e, we define a potential function Φ_e by

$$\Phi_e = O(\log n) \cdot (\text{level of edge } e).$$

Here an edge that is deleted is understood to have potential 0. We define an overall potential by

$$\Phi = \sum_{e \in E} \Phi_e = O(\log n) \sum_{e \in E} (\text{level of edge } e).$$

Insertion. Inserting an edge e requires updating an Euler tree which takes at most $O(\log n)$ amortized time (per Section 32.2). Meanwhile the potential increases by $O\Big(\log^2 n\Big)$ since the maximum level is $O(\log n)$. Thus `insert` takes $O\Big(\log^2 n\Big)$ amortized time.

Deletion. Deleting an edge e first requires updating up to $O(\log n)$ Euler trees, each of which takes $O(\log n)$ amortized time. Removing e only decreases the potential.

It remains to account for the work from trying to replace e. We first observe that for each level i we pay $O(\log n)$ time to query for at least one edge in $A \cap F_i'$ and for at least one edge in E_i'. These initial queries for each level, over at most $O(\log(n))$ levels, contribute $O\Big(\log^2(n)\Big)$ amortized time.

Beyond these initial queries, each query can be charged to an edge d that is either (a) pushed one level down, or (b) selected as the replacement. Scenario (b) happens only once and is negligible. But event (a) may occur repeatedly. In event (a), the potential Φ_d of d decreases by $O(\log n)$ when d is pushed to the next level. This pays for the $O(\log n)$ amortized time that the query for d required. Thus these queries are free.

All put together, even though we may have to go through many edges to replace the deleted edge, the amortized running time for a deletion comes out to $O\Big(\log^2(n)\Big)$.

Theorem 30.8 ([HLT01])**.** *In a dynamically changing graph, in $O\Big(\log^2 n\Big)$ amortized time per edge insertion and edge deletion, we can support connectivity queries in $O(\log n)$ amortized time.*

30.4 Additional notes and references

Splay trees were introduced by Sleator and Tarjan [ST85b]. Euler trees are by Henzinger and King [HK99] and dynamic connectivity is by Holm, Lichtenberg, and Thorup [HLT01]. For more on splay trees, see [Eri13b]. For more on dynamic connectivity, see also the lecture notes and video by Demaine [Dem12a].

Lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

30.5 Exercises

Exercise 30.1. In previous discussions about link-cut trees, we focused on aggregate operations on node-to-v paths. Show how to extend link-cut trees to support the following operations in polylogarithmic amortized time.

1. Given two vertices u and v, where u is an ancestor of v, aggregate (e.g., sum values) over the u-to-v path in the tree.
2. Given two vertices u and v, find the *least common ancestor* of u and v.
3. Given two vertices u and v, aggregate (e.g., sum all values) over the u-to-v path in the tree.

30.A Amortized analysis of the remaining splay tree operations

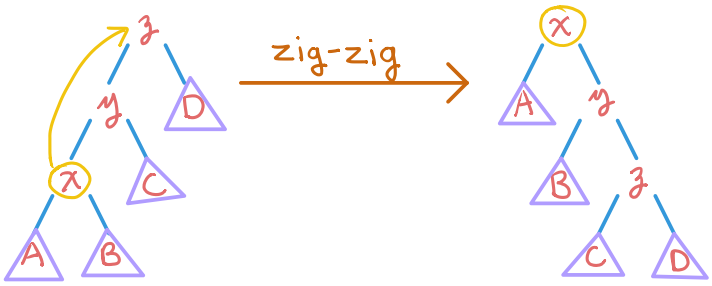
Here we present the remaining analysis of the splay tree rotations zig-zag, zag-zig, zig, and zag.

Zig-zig (and zag-zag)

Lemma 30.3. *Consider any zig-zig or zag-zag rotation making a node x the root of a subtree. Then*

$$2 + \Delta\mathbf{\Phi} \leq 3\Delta\Phi_x.$$

Proof. Let us label the relevant subtrees and nodes as follows.



We let $A, \dots, D$ also denote the size of the respective subtrees. We have

$$\begin{aligned}&\Delta\mathbf{\Phi}\\ &= \log(2+B+C+D) + \log(1+C+D) - \log(2+A+B+C) - \log(2+A+B).\end{aligned}$$

We also have

$$\Delta\Phi_x = \log(3+A+B+C+D) - \log(1+A+B).$$

Observe that

$$\log(2+B+C+D) - \log(2+A+B+C) \le \Delta\Phi_x$$

by monotonicity. This leaves the terms $\log(1+C+D) - \log(2+A+B)$. We also have

$$\begin{aligned}\log(1+C+D) + \log(1+A+B) &\le 2\log\Big(\frac{2+A+B+C+D}{2}\Big)\\ &\le 2\log(2+A+B+C+D) - 2.\end{aligned}$$

Thus

$$\begin{aligned}&2 + \log(1+C+D) - \log(2+A+B)\\ &\le 2\log(2+A+B+C+D) - \log(1+A+B) - \log(2+A+B)\\ &\le 2\Delta\Phi_x.\end{aligned}$$

This gives the overall claim that

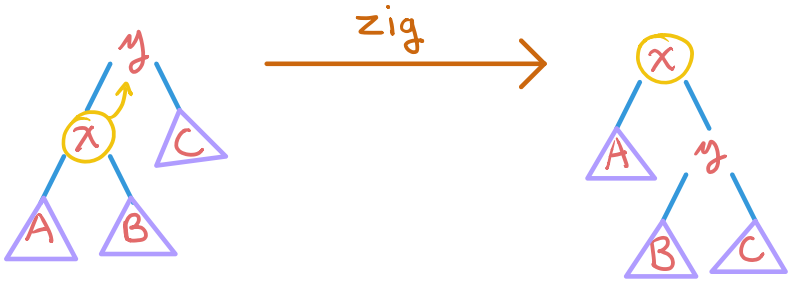
$$2 + \Delta\mathbf{\Phi} \le 3\Delta\Phi_x.$$

□

30.A.1 Zig and zag

Lemma 30.4. *Consider any zig or zag rotation making a node x the root of a subtree. Then $\Delta\mathbf{\Phi} \le \Delta\Phi_x$.*

Proof. By symmetry it suffices to analyze the `zig` operation. We refer to the relevant quantities per the following diagram.



We have

$$\Delta\mathbf{\Phi} = \log(1 + B + C) - \log(1 + A + B).$$

We also have

$$\Delta\Phi_x = \log(1 + A + B + C) - \log(1 + A + B).$$

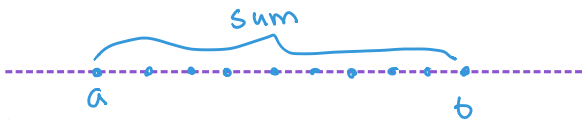
The claim follows immediately. □

This gives the last missing proof from Theorem 30.1, establishing the theorem.

Chapter 31

Range Queries and Dynamic Trees

31.1 Range queries



Suppose we have a family of n comparable keys $k_1 < \cdots < k_n$ where each key is associated with some data (e.g., a numerical value). A *range query* is specified by an interval[1] and some method of aggregation, and the output is the appropriately aggregation of values over all keys that lie in the interval. For example, if there are numeric values associated with each key, we might ask for

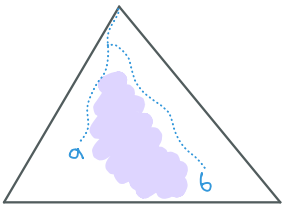
> *The sum of all values associated with all keys in* $[a, b]$.

Here a key k is in $[a, b]$ if $a \leq k \leq b$. Similarly for (a, b), $(a, b]$, and $[a, b)$. Alternatively we can ask for the *max* of all values (with keys) in $[a, b]$. A slightly different kind of query occurs in a situation where each key may be associated with a list of values of some kind. We might ask for

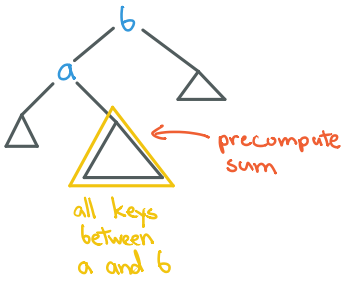
> *The combined lists of all values associated with all keys in* $[a, b]$*, in sorted order w/r/t their keys.*

Not only do we want to serve these queries efficiently, but we want to keep serving them efficiently as the keys and values are updated dynamically.

[1] open, closed, or half-open. We remind the reader that $[a, b] = \{x : a \leq x \leq b\}$, $(a, b) = \{x : a < x < b\}$, $(a, b] = \{x : a < x \leq b\}$, and $[a, b) = \{x : a \leq x < b\}$.



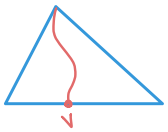
One solution to the problem is with binary search trees. We insert all the keys in a binary search tree tree where each node also auxiliary pointers to the data associated with the key. Now, suppose we want to be able to compute the sum over an interval $[a, b]$. For a static tree, we would search for a and b and then sum over all the nodes "inside" the search paths. To avoid visiting all of these nodes, we can precompute the sum over all subtrees. Then we only have to visit the root of each subtree inside the search paths. Still the total time will be proportional to the path lengths, which may be very long.



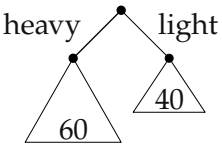
Instead we can use balanced binary search trees. Here we will focus on splay trees, continuing our discussion from last time. Again we maintain the sum of all subtrees, and point out that it is easy to update these sums with each local rotation. Now, suppose we want to query the sum over $[a, b]$. We assume a and b are both keys; otherwise replace a with the first key $> a$ and b with the first key $< b$. If we splay a and then b, then b will be the root, a will be the left child of b, and all keys between a and b will be in a's right subtree. Thus we sum the values of a, of b, and the precomputed sum of the a's right subtree.

Clearly this approach extends more generally than sums – as long as the aggregation can be computed pairwise in any order (i.e., it is defined by an *associative* binary operator) in constant time, then we can maintain the aggregates at each subtree, and follow the above approach. This is more flexible then might appear. Take for example the second type of query where we want to list of all the values in $[a, b]$. We maintain, at each node, the total number of values in the subtree rooted at that node. We can use these quantities to search and splay for the key k that contains the ith value in $[a, b]$, for any fixed i, in $O(\log n)$ time per key. (There are slightly more involved approaches that can reduce the time to $O(\log n)$ plus $O(1)$ per key.)

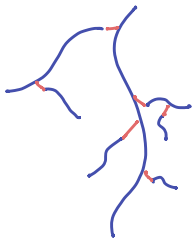
31.2 Range queries on paths in a tree



One way to generalize range queries on a line is in trees. The model is as follows. Let T be a rooted tree, whose nodes are associated with some values. We now consider aggregations over all the nodes on the path from a given vertex to a root. For example, given a node v, we may want to compute the sum of all values on the path from v to the root.



Let v be a node in a rooted tree, and w a child of v. We call w a *heavy child* of v if it contains at least half the nodes in the tree rooted at v. Any node has at most one heavy child. We call the edge from a parent to a heavy child a *heavy edge*. The paths of heavy edges partitions the nodes of a tree into paths, called the heavy path decomposition.

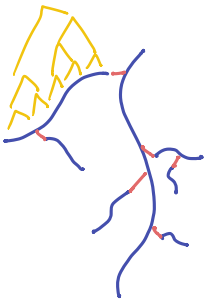


Edges that are not heavy are called *light*. For a vertex v, the *light depth* of a vertex v is defined as the number of light edges on the v to root path. Observe that

$$(\text{light depth } v) \leq \log_2(n)$$

because each light edge divides the number of remaining nodes in half.

Theorem 31.1. *One can support path queries in a rooted tree in $O\left(\log^2(n)\right)$ amortized time per operation.*

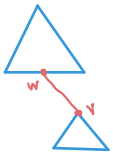


Proof. We compute a heavy path decomposition over the tree. For each heavy path, we build a splay tree on that path keyed by depth, that supports range queries along that path. Now, when querying a node v, the root to v path breaks up into segments of heavy paths split up by light edges. There are at most $O(\log n)$ light edges, so each there are $O(\log n)$ heavy segments. We sum over each segment by a range query to the corresponding splay tree. Then we sum all the values. There are $O(\log n)$ queries to splay trees and each query takes $O(\log n)$ amortized time. □

31.3 Dynamic trees

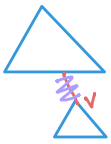


We extend the model from the previous section further by allowing the rooted tree T to change. More generally, we will design a data structure that maintains a forest of rooted trees. The rooted trees are updated by the following operations.



Link. Given two vertices w and v, where v is the root of a tree, `link(`w`,`v`)` make v a child of w.

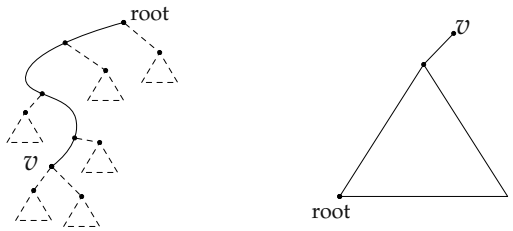
Cut. Given a vertex v that is not a root, `cut(`v`)` removes v's parent edge, making v the root of its own tree.



In the static setting, the heavy path decomposition reduces path aggregates to $O(\log n)$ range queries. But here the heavy paths can frequently change as edges are added and deleted, which would require us to start to rebuild many splay trees.

31.3.1 Access.

The `link-cut-tree` data structure maintains a *dynamic* path decomposition of each tree that rapidly responds to each operation. These changes are made through an operation called `access`. `Access` takes as input a node v, and updates the path decomposition so that the v's path is one of the paths in the path decomposition. `Access` also update the underlying splay trees on the fly to ensure that each path in the decomposition is still represented by a splay tree. In particular, after `access(`v`)`, there is a splay tree over the root-to-v, of which v is the root. As the last vertex in the path, v has no left child in the auxiliary tree.



Represented tree v's auxiliary tree

The pictures above describe the arrangement after `access(`v`)`. The represented tree is on the left, and the root to v path is emphasized by solid lines. The dashed edges hanging off the root to v path are not in any path of the path decomposition. To be explicit, let us state the guarantees of `access(`v`)` that we invoke.

1. The root-to-v path is a path in the path decomposition.
2. v is the root of the splay tree representing its path.

We will eventually show that `access(`v`)` takes $O\left(\log^2(n)\right)$ amortized time.[2] Given `access`, we can implement `link`, `cut`, and path queries as follows.

[2]One can obtain an improved bound of $O(\log n)$, but the analysis is a little too complicated to fit in this lecture. We refer the reader to [ST85b] or [Dem12b] for that analysis.

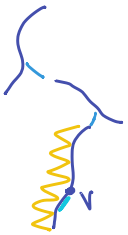
Cut. Recall that `cut` removes the parent edge of a node v. To implement this, we first call `access(`v`)`, which makes the root path to v part of the path decomposition with v the root of the auxiliary splay tree. In the splay tree, v has no right child, and the left child of v contains the rest of the path. We remove the edge from v to its left subtree.

Link. Recall that `link(`v`,`w`)` takes a node v, which is the root of some tree, and makes it a child of a node w, which is the child of another tree. To implement `link(`v`,`w`)`, we first access w. This gives us a splay tree over the root path to w. We also access v. Since v was already the root, v will be only vertex in its auxiliary tree. We set w to be v's parent, and we make w's auxiliary tree the left child of the (singleton) splay tree at v.

Find-root. Recall that `find-root(`v`)` returns the root of the tree containing v. To find the root, we access v, which gives us a splay tree containing the root-to-v path. The root has the smallest key in this tree. We `splay` the root to be the root of the auxiliary tree, and return it.

Aggregating over a path. Suppose the nodes were associated with numerical values, and we want to sum up the values along the root to v path for a node query. We extend the data structure so that all the auxiliary splay trees act as a range tree over its nodes. Assuming this is in place, we implement `sum-path(`v`)` by calling `access(`v`)`, and then return the sum in v's auxiliary tree. Of course, summation can be replaced with other aggregate operations as discussed in Section 32.1.

31.3.2 Analyzing `access`



It remains to define and analyze `access`. Before defining `access`, we first introduce a helpful auxiliary function called `clip`. `clip` takes as input a vertex v. That vertex v is part of some auxiliary path in the path decomposition. `clip` adjusts the path decomposition so that v is the last vertex in its auxiliary path. Psuedocode for `clip` is as follows.

`clip`(v)

// *Makes v the last vertex in its auxiliary path in the path decomposition. Makes v at the root of its auxiliary splay tree.*

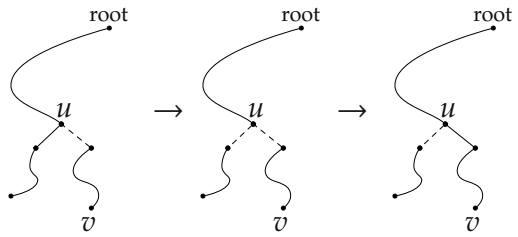
1. Splay v within its auxiliary tree.

// *The right subtree of v in the auxliary tree represents the part of the path below v.*

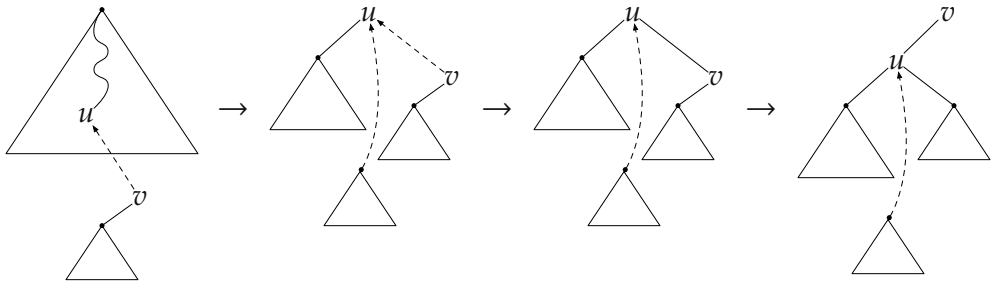
2. In the auxiliary tree, split off the right subtree of v as its own path, with v as the parent of the path.

`clip` consists of one splay operation and a constant number of pointer changes, hence takes $O(\log n)$ amortized time.

Now we describe `access`. `Access` takes as input a vertex v, and the goal is to adjust the path decomposition so that the root-to-v path is one of the paths. The first step is to `clip` v. Then, until v's auxiliary path is the entire root-to-v path, we extend v's auxiliary path as follows. We first look at the parent u of the auxiliary path. We `clip` u, make u the bottom of it's auxiliary path. We then attach v's auxiliary path to u's auxiliary path, making one long auxiliary path. These steps, in terms of their effect on the path decomposition, are illustrated below.



Above, the solid edges represent paths in the path decomposition, and dashed edges represent edges excluded from the path decomposition. In terms of the underlying splay trees, `clip`(u) makes u the root of its splay tree with no right subtree. We already have v at the root of its splay tree. We then make v the right child of u. Then we splay v to the top. The steps on the splay trees are diagrammed below.



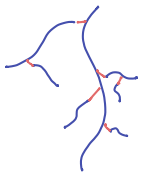
Above, the solid triangles represent splay trees. The dashed edges represent parent pointers in the represented tree. Pseudocode for `access` is as follows.

`access(`v`)`

1. clip v
2. until the path for v is the entire v to root path
   A. let u be the parent vertex of v's auxiliary path.
   B. clip u
   C. attach the auxiliary tree for v as the right subtree of u in the auxiliary tree for u
   D. splay v

Lemma 31.2. *Suppose a call to* `access(`v`)` *requires C changes in the path decomposition. Then* `access(`v`)` *takes $O((C+1)\log(n))$ amortized time.*

Proof. We first pay $\log(n)$ amortized time to call `clip(`v`)`. For each iteration of the loop, we pay $O(\log n)$ amortized time for a constant number of `splay` and `clip` operations for each iteration of the loop. Each iteration corresponds to a change in the path decomposition. □



It remains to bound C, the total number of changes to the path decomposition. We do this by comparison to the heavy path decomposition. Recall that a vertex is heavy if its subtree represents at least half of the subtree of its parents (Section 31.2). Vertices that aren't heavy are called light. An edge is called heavy if the child is heavy and an edge is called light if the child is light.

We classify each edge in the represented tree as follows. First, an edge is either in one of the paths in the path decomposition, or not. Second, each edge is either heavy, or not. The combinations creates four disjoint categories of edges.

Let us say that an edge is *active* when it is becomes part of a path in the path decomposition. We say that an edge is *inactive* when it is not in the path decomposition. The cross product of

$$\begin{Bmatrix}\text{heavy}\\ \text{light}\end{Bmatrix} \times \begin{Bmatrix}\text{active}\\ \text{inactive}\end{Bmatrix}$$

gives four classes of edges at any point. Every time we change the path decomposition, we are either

1. Creating an active light edge.

2. Destroying a light active edge.
3. Destroying a heavy active edges.
4. Creating a heavy active edge.

Every time we change the path decomposition, we are demoting one edge and promoting another edge, and these two edges are siblings. In particular, at most one of them is heavy. More precisely, let us define

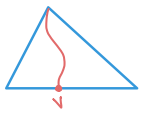
$$\begin{aligned} C &= \#\text{ changes to the path decomposition,} \\ L &= \#\text{ active light edges created,} \\ L' &= \#\text{ light active edges destroyed,} \\ H &= \#\text{ active heavy edges destroyed,} \\ H' &= \#\text{ heavy active edges created.} \end{aligned}$$

Then

$$C \leq L + H + L' + H' \overset{\text{(a)}}{\leq} 2L + H + H' \overset{\text{(b)}}{<} 2(L+H) + n.$$

Here (a) is because every light edge we destroyed was previously created. (b) is because the number of heavy active edges we create, but do not later destroy, is less than n as there are only $n-1$ active edges total. It remains to upper bound L and H for `access`, `link`, and `cut`.

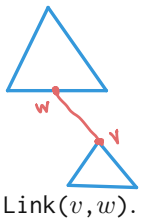
1. `Access(`v`)`. We have



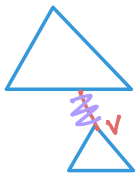
$$L + H \overset{\text{(c)}}{\leq} 2L + 1 \overset{\text{(d)}}{\leq} O(\log n),$$

   for the following reasons. (c) is because every time a heavy edges is demoted, unless it was edge with parent v, a light edge is promoted instead. (d) is because there are at most $O(\log n)$ light edges along the root to v path.

2. When we link v to w, besides the work charged to calls to `access`, some light edges along w's path may become heavy, and some heavy edges hanging off of w's path may become light. However, the edges that become heavy are already promoted, and the edges that became light are already demoted, by the preceding called to `access`. Thus, excluding the parts charged to `access`, we have $L + H = 0$.



3. When we cut v from its parent, some of the nodes on the root-to-v's path may become light. But there are at most $O(\log n)$ light edges from the root to v path. The cut may also destroy a heavy edge if v's parent heaby was heavy. Thus, excluding changes due to `access`, we have $O(\log n)$ with respect to the potential.



`Cut(`v`)`.

Thus we have given an amortized analysis that shows that each `access`, `link`, and `cut` introduces $O(\log n)$ amortized changes to the path decomposition. Meanwhile the amortized running time, by Lemma 31.2, is an additional $O(\log n)$ factor on top of that, in addition to a $O(n \log n)$ overhead from the n extra changes. Thus we have shown the following.

Lemma 31.3. *With $O(n \log n)$ additional overhead,* `access`*,* `link`*, and* `cut` *each take $O\big(\log^2(n)\big)$ amortized time.*

31.4 Roots, re-rooting, and an application to dynamic disjoint union

Recall the disjoint union problem which was important for MST. We have a set of n elements each initially in their own singleton sets. For any two elements x and y in different sets[3], we can call `union` to combine their sets. Meanwhile for any two elements we can query if they are in the same set.

This version of the disjoint union data structure is said to be *partially dynamic* – they allow us to take `union`'s of two sets, but we cannot "delete" a previous `union`. We want to extend this to a *fully dynamic* disjoint union data structure in the following sense. Suppose we call `union(`x`,`y`)` on two elements x and y. Later on, we want to be able to call "`split(`x`,`y`)`" to undo this union, while retaining all other `union`'s that might have occurred since.

An equivalent formulation is as follows. Let $F = (V, E)$ be a forest. Suppose edges are inserted into F (without introducing cycles) and deleted from F. The goal is to be able to query, for any two vertices u and v, whether u and v are connected in F. Edge insertions correspond to `union`'s, and edge deletions correspond to `split`'s.

This dynamic variant of disjoint-union can be resolved by link-cut trees with two new operations.

1. `root(`v`)` returns the root of the tree containing v.

2. `re-root(`v`)` reroot's v's tree to have root v.

[3]Here we restrict ourselves to unions over elements in different sets. This distinction was not necessary before.

`Union(`u`,`v`)` corresponds `reroot`-ing v and then `link`-ing the nodes. `Split(`u`,`v`)` corresponds to a `cut`. To query if two elements are in the same set, we call `root` on both and see if they are in the same set.

Implementing `root(`v`)` is immediate. We `access(`v`)`, and then splay the leftmost node in the auxiliary tree to find the root. To implement `re-root`, we first observe that this corresponds to reversing all the edges on the v-to-root path. To this end, we maintain, for each node, a bit flag that indicates whether or not the subtree is given in order or in reverse order. Then to reverse all the edges along a path in some splay tree, we flip this bit. Thus to `re-root(`v`)`, we `access(`v`)`, and flip the bit in the auxiliary tree.

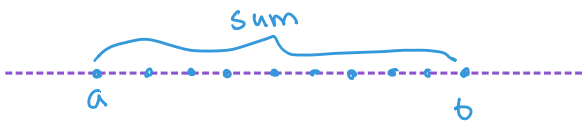
As currently planned, the next lecture will discuss how to remove the restriction that F is a forest. That is, we want to able to query connectivity in a dynamically updated graph.

Chapter 32

Dynamic Connectivity and MST's

This topic, while advanced, was chosen because it sits at the intersection of data structures and graph algorithms.

32.1 Range queries



Suppose we have a family of n comparable keys $k_1 < \cdots < k_n$ where each key is associated with some data (e.g., a numerical value). A *range query* is specified by an interval[1] and some method of aggregation, and the output is the appropriately aggregation of values over all keys that lie in the interval. For example, if there are numeric values associated with each key, we might ask for

> *The sum of all values associated with all keys in* $[a, b]$.

Here a key k is in $[a, b]$ if $a \leq k \leq b$. Similarly for (a, b), $(a, b]$, and $[a, b)$. Alternatively we can ask for the *max* of all values (with keys) in $[a, b]$. A slightly different kind of query occurs in a situation where each key may be associated with a (possibly empty) list of values of some kind. We might ask for

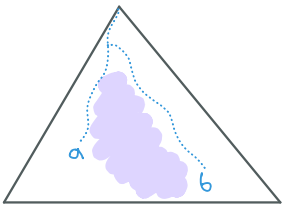
> *The combined lists of all values associated with all keys in* $[a, b]$*, in sorted order w/r/t their keys.*

This is sometimes called a *list query*. We point out list queries are bound to be slow if the output list is very long. For this, we often seek running times proportional to the

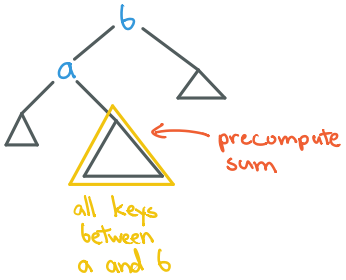
[1] open, closed, or half-open. We remind the reader that $[a, b] = \{x : a \leq x \leq b\}$, $(a, b) = \{x : a < x < b\}$, $(a, b] = \{x : a < x \leq b\}$, and $[a, b) = \{x : a \leq x < b\}$.

total number of elements in the output. An example might be $O(\log(n) + k)$ where k is the number of values listed in the output.

Now, not only do we want to serve these queries efficiently, but we want to keep serving them efficiently as the keys and values are updated dynamically.

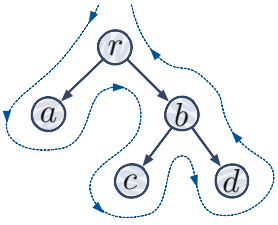


Consider sums for simplicity. One solution to the problem is with binary search trees. We insert all the keys in a binary search tree tree where each node also auxiliary pointers to the data associated with the key. Now, suppose we want to be able to compute the sum over an interval $[a, b]$. For a static tree, we would search for a and b and then sum over all the nodes "inside" the search paths. To avoid visiting all of these nodes, we can precompute the sum over all subtrees. Then we only have to visit the root of each subtree inside the search paths. Still the total time will be proportional to the path lengths, which may be very long.



We can control the path lengths by using balanced binary search trees. Recall that with lazy rebuilding, we can maintain a binary search tree of height $O(\log n)$ in $O(\log n)$ amortized time per insertion and deletion. One can update the sums of each subtree as elements are inserted and marked for deletion, and recompute the sums when rebuilding, with negligible overhead.

32.2 Euler trees



Given an undirected tree T, an *Euler tour* is a walk that traverses the entire tree in each direction exactly once. Every tree has an Euler tour and it is easy to construct one. Root the tree arbitrarily. Starting from the root, pick any child of the root. Go down that edge, and inductively take an Euler tour of that subtree, and come back up to the root. Clearly we use the edge to the subtree once in each direction. We repeat this for the other children of the root, which together gives an Euler tour of the entire tree.

Euler tours inspired a data structure called *Euler-tour trees* [HK99] that, like link-cut trees, support aggregation in rooted trees that are dynamically updated. Compared to link-cut trees, they aggregate over *subtrees*, rather than paths. They are also simpler than link-cut trees.

We maintain a fixed Euler tour over the rooted tree. As we follow the Euler tour, we write down each vertex the first and last time we visit it (that is, when we enter

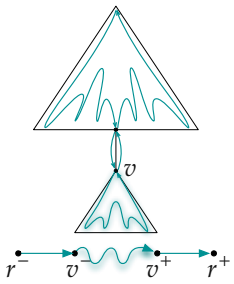
and leave the subtree rooted at that vertex). This produces a sequence of vertex labels starting and ending with the root. For example, in the Euler tour above, we have the sequence

$$r,\ a,\ r,\ b,\ c,\ b,\ d,\ b,\ r$$

The first appearance of a vertex v corresponds to the point where we enter the subtree rooted at v. The last appearance of a vertex v corresponds to the point where we leave the subtree rooted at v. Vertices of nested subtrees are nested in the Euler tour as well. For a vertex v, we let v^+ denote the first appearance of v in the Euler tour and let v^- denote the last appearance.

We build and maintain a splay tree[2] over the Euler tour (based on the order of the Euler tour). For each vertex v, we explicitly maintain pointers to the first instance v^+ and last instance v^- in the splay tree.

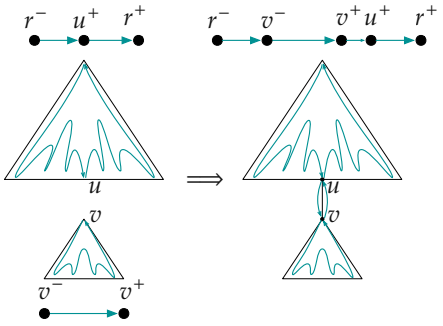
32.2.1 Aggregating over subtrees.



Suppose we want to aggregate (e.g., sum all values) in the subtree rooted at a node v. Euler trees are designed specifically for this application. Aggregating over a subtree rooted at v corresponds to a range query between v^+ and v^- in the corresponding splay tree. As discussed previously, this takes $O(\log n)$ amortized time.

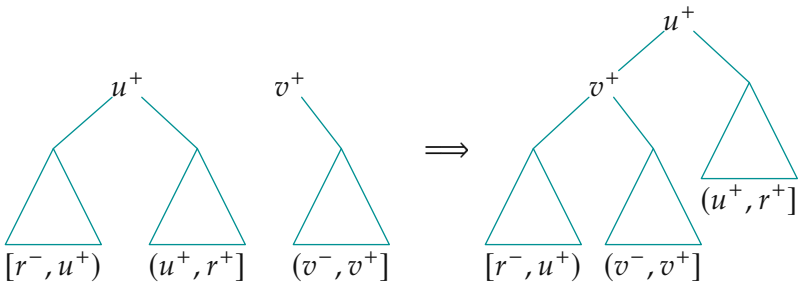
Lemma 32.1. *An Euler tour tree reduces aggregation over subtrees to range queries in a search tree. In particular, with splay trees, most operations take $O(\log n)$ amortized time.*

32.2.2 Link.



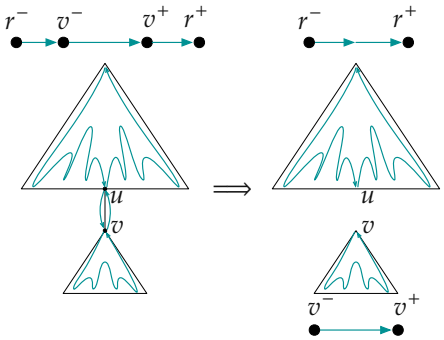
Suppose we want to link v as a child of the node u, where we assume v is the root of its tree. In terms of the Euler tour, we want to insert the entire Euler tour rooted at v – from v^+ to v^- – just before the last time we visit u: u^+. We splay u^+ and v^+ in their respective trees. Note that v^+ has no left subtree. We remove the left subtree of u^+ and make it the left subtree of v^+, and make the tree rooted at v^+ the left subtree of u^+. See the diagram below.

[2]Other balanced binary search trees would also work.

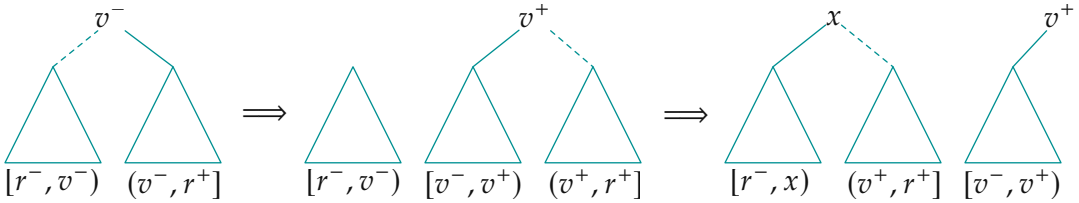


Lemma 32.2. *Euler tour trees support* `link` *in* $O(\log n)$ *amortized time.*

32.2.3 Cut.



Suppose we want to cut the parent edge of a vertex v. In terms of the Euler tour, we remove the subsequence between the first and last visit of v from the Euler tour, gluing together the part before v and the part after v. This is easy to do in a splay tree. We splay v^+, which puts the interval $[v^-, v^+)$ in the left subtree of v^+. We disconnect this left subtree. Then we splay v^-, which puts the interval $(v^-, r^+]$ in the right subtree of v. We disconnect that right subtree as well. This gives us three splay trees with the intervals $[v^-, v^+)$, $[v^+, v^-]$, and $(v^-, r^+]$. We then concatenate the first and third intervals as follows. Let x be the last node in $[v^-, v^+)$. We splay x, upon which it has no right subtree, and then attach the splay tree for $(v^-, r^+]$ as a right subtree. See the diagram below.



The running time is bounded by a constant number of `splay`'s. Thus we have the following.

Lemma 32.3. *Euler tour trees support* `cut` *in* $O(\log n)$ *amortized time.*

32.2.4 Root.

An additional type of query that will be useful in the sequel is called `root`. Given a vertex v, `root(`v`)` returns the root of the tree containing v. This corresponds to

splaying the smallest vertex in the tree containing v, which takes $O(\log n)$ amortized time.

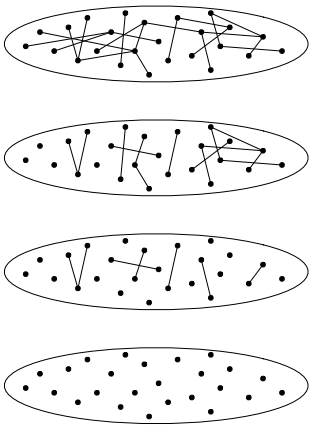
32.2.5 Re-root.

(Todo... rotate...)

32.3 Dynamic Connectivity

Let $G = (V, E)$ be an unweighted and undirected graph. Recall that we can build a spanning forest F in G in $O(m + n \log n)$. Our goal is to *maintain* a spanning tree in G as G *changes*. In each iteration, an edge is either added or deleted from the graph. Here computing an entire spanning tree from scratch seems wasteful.

Let $L = \lceil \log n \rceil$. We maintain a nested family of subsets of both E and F,



$$E_1 \subseteq E_2 \subseteq E_3 \cdots \subseteq E_L \subseteq E$$

and

$$F_1 \subseteq F_2 \subseteq F_3 \cdots \subseteq F_L \subseteq F,$$

subject to the following invariants.

1. $E_L = E$ and $F_L = F$.
2. For each i, $F_i = F \cap E_i$.
3. For each i, $E_i \subseteq \text{span}(F_i)$.
4. Every component in E_i has size $\leq 2^i$.

We denote the differences between the sets by

$$E'_i \stackrel{\text{def}}{=} E_i \setminus E_{i-1} \text{ and } F'_i = F_i \setminus F_{i-1}$$

for each i. The *level* of an edge e is defined as the lowest index i for which $e \in E_i$; this is also the unique index i such that $e \in E'_i$.

For each level i, and each vertex v, we maintain the list of edges at level i incident to v (i.e., $\delta(v) \cap E'_i$, where $\delta(v)$ denotes the edges incident to v). We create an Euler tree over each component of F_i to support the following queries.

1. *Given the root of a subtree, return any edge $e \in F'_i$ in that subtree, if one exists.*
2. *Given the root of a subtree, return any edge $e \in E'_i$ incident to a vertex in a subtree, if one exists.*

Both of these operations correspond to a "list aggregate", as discussed previously, and take $O(\log n)$ amortized time.

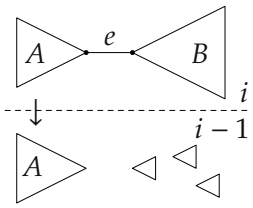
Querying connectivity. The only query is for pairwise connectivity: given u and v, are u and v connected in G? This corresponds to querying for the roots of u and v in their Euler trees, and seeing if they are the same. As discussed in Section 32.2, this takes $O(\log n)$ amortized time.

Insertion Suppose we want to insert an edge e. If F spans e, then we insert e into the first level i such that F_i spans e. If e is not spanned by F, then we insert e into both F_L and E_L. Note that "inserting" an edge also involves updating the Euler tree at the level the edge is inserted.

Deletion. Suppose we want to delete an edge e at level ℓ. Then we remove e from E_ℓ (and $E_{\ell+1}$, $E_{\ell+1}$, ...) and we remove e from the incidence lists of its endpoints. If e is not in F then we are done.

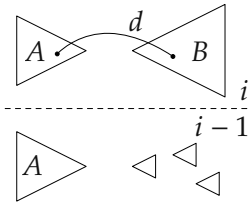
Now suppose $e \in F$; in particular, $e \in F'_\ell$. We first remove e from F_ℓ (and $F_{\ell+1}, F_{\ell+2}, \dots$). Then we look for an edge to replace it. For each level i from ℓ to L we do the following.

Suppose removing e separated a tree in F_i into two components A and B. We assume that A is the smaller of the two components. Then we have



$$|A| \leq \frac{1}{2}(|A| + |B|) \leq 2^{i-1},$$

so in particular, we can now push all of the edges in A down to level $i-1$ while satisfying invariant 4 above. So that's the first thing we do – we repeatedly query the Euler tree for level i edges in $A \cap E'_i$, pushing each one down to level $i-1$, until there are none left.



At this stage all of A is at level $i-1$ or lower. Now we look at all the level i edges incident to any vertex in A. These edges can be retrieved one at a time from the Euler tree (in $O(\log n)$ amortized time per edge). For each such edge d, we check to see if d is spanned by A. If it is, then we push it down to level $i-1$. If it isn't, then in particular it must connect A to B. That is, d is our replacement edge. We insert d into F_i and all higher forests, and exit the entire procedure.

If we end up processing all of the level i edges without finding a replacement, then we go up to level $i+1$, and repeat. If we end up going through all the levels without a replacement, then we conclude that there is no replacement for e, and that the current forest is a spanning forest.

32.3.1 Analysis

Above we already observed that querying connectivity takes $O(\log n)$ amortized time via the Euler tree. It remains to analyze `insert` and `delete`. Note that `delete` can have high running times in the worst case if it has to search through many candidate edges for a replacement. So we will use amortized analysis instead.

For each edge e, we define a potential function Φ_e by

$$\Phi_e = O(\log n) \cdot (\text{level of edge } e).$$

Here an edge that is deleted is understood to have potential 0. We define an overall potential by

$$\Phi = \sum_{e \in E} \Phi_e = O(\log n) \sum_{e \in E} (\text{level of edge } e).$$

Insertion. Inserting an edge e requires updating an Euler tree which takes at most $O(\log n)$ amortized time (per Section 32.2). Meanwhile the potential increases by $O\Big(\log^2 n\Big)$ since the maximum level is $O(\log n)$. Thus `insert` takes $O\Big(\log^2 n\Big)$ amortized time.

Deletion. Deleting an edge e first requires updating up to $O(\log n)$ Euler trees, each of which takes $O(\log n)$ amortized time. Removing e only decreases the potential.

It remains to account for the work from trying to replace e. We first observe that for each level i we pay $O(\log n)$ time to query for at least one edge in $A \cap F_i'$ and for at least one edge in E_i'. These initial queries for each level, over at most $O(\log(n))$ levels, contribute $O\Big(\log^2(n)\Big)$ amortized time.

`insert(`e`)`

/ Below, when we "insert an edge" at a level i, this also means we update all the relevant data structures. */*

1. If F spans e
   A. Insert e into the first level i such that F_i spans e
2. Otherwise insert e into F_L and E_L.

`delete(`$e = \{a, b\}$`)`

1. remove e from the incident lists of a and b
2. if $e \in F$
   A. let ℓ be the level of e
   // remove e from each forest that contains it
   B. call `cut(`u, v`)` on $F_\ell, \dots, F_L$
   C. for $i = \ell, \ell + 1, \dots, L$
      1. Let A and B be the trees of F_i containing a and b, respectively.
      // We assume $|A| \leq |B|$ (otherwise switch a and b)
      // $|A| \leq \frac{1}{2}(|A| + |B|) \leq 2^{i-1}$, and we can push all of A to F_{i-1}
      2. For each level i edge $f = \{x, y\} \in A \cap F'_i$
         a. Call `link(`x`,`y`)` on F_{i-1} and add f to E_{i-1}
      3. For each level i edge $f = \{x, y\} \in E'_i$ with $x \in A$
         // To facilitate this loop, we maintain
         a. if $y \in B$ *// Replace e with f*
            1. call `link(`x`,`y`)` on $F_i, F_{i+1}, \dots, F_L$ and exit
         b. else *// f is spanned by A and can be pushed to level $i - 1$*
            1. add f to E_{i-1}

Beyond these initial queries, each query can be charged to an edge d that is either (a) pushed one level down, or (b) selected as the replacement. Scenario (b) happens only once and is negligible. But event (a) may occur repeatedly. In event (a), the potential Φd of d decreases by $O(\log n)$ when d is pushed to the next level. This pays for the $O(\log n)$ amortized time that the query for d required. Thus these queries are free.

All put together, even though we may have to go through many edges to replace the deleted edge, the amortized running time for a deletion comes out to $O\big(\log^2(n)\big)$.

Theorem 32.4 ([HLT01])**.** *In a dynamically changing graph, in $O\big(\log^2 n\big)$ amortized time per edge insertion and edge deletion, we can support connectivity queries in $O(\log n)$ amortized time.*

32.4 Dynamic MST

As a natural generalization of the previous section, we can consider the problem of maintaining the *minimum spanning tree* as edges are inserted and deleted from the graph. Here will focus on the partially dynamic, *decremental* setting where edges are only deleted. One can also handle insertions but it is a bit more complicated; see [HLT01].

We take the dynamic connectivity and make one obvious modification: the forest F is required to be the *minimum weight* spanning forest of the graph. This implies that each F_i is the minimum spanning forest of E_i.

In addition to the Euler trees from before, we have a link-cut tree for F, that supports the following queries in $O(\log n)$ amortized time[3]

1. *Given two vertices u and v in the same component of F_i, find the maximum weight edge in F_i' on the path from u to v.*
2. *Given two vertices u and v in the same component of F_i, find the minimum weight edge in E_i' incident to any vertex on the u to v path.*

These u-to-v path queries are a generalization of root-to-v path queries previously discussed. Link-cut trees can be extended to support these queries as well, which we leave as exercise 32.1.

[3]In class, we proved $O\big(\log^2(n)\big)$ running times. But a more advanced analysis gives $O(\log n)$.

Insertion. Suppose we insert an edge e. If e is not spanned by F, then we insert e in F at level L.

Now suppose e is spanned by F. Using the link-cut tree, we identify the minimum weight edge f on the unique path in F between the endpoints of e. If f has weight greater than e, then we replace e with f at the same level as f. Otherwise we insert e at the first level ℓ where F_ℓ spans e.

Deletion. We following the same process as in connectivity, with one adjustment. When querying for a possible replacement edge incident to the current component, we query in particular for the minimum weight edge. We can adjust our Euler tree to support this with no additional overhead.

Analysis. In terms of running time, the only real modification is that we are also using link-cut trees. But this only generates $O(\log n)$ additive overhead and does not affect the asymptotic running time.

Theorem 32.5 ([HLT01])**.** *Let $G = (V, E)$ be a weighted undirected graph dynamically updated by deletions. Then we can maintain the minimum weight spanning tree in $O\left(\log^2 n\right)$ amortized time per edge deletion.*

32.5 Exercises

Exercise 32.1. In previous discussions about link-cut trees, we focused on aggregate operations on node-to-v paths. Show how to extend link-cut trees to support the following operations in polylogarithmic amortized time.

1. Given two vertices u and v, where u is an ancestor of v, aggregate (e.g., sum values) over the u-to-v path in the tree.
2. Given two vertices u and v, find the *least common ancestor* of u and v.
3. Given two vertices u and v, aggregate (e.g., sum all values) over the u-to-v path in the tree.

32.6 Additional notes and references

Euler trees are from [HK99], who gave a *randomized* algorithms for dynamic connectivity and $(1+\epsilon)$-approximate MST with polylogarithmic update time. The (deterministic) dynamic connectivity and MST presented here are from [HLT01]. See also the lecture notes and video by Demaine [Dem12a].

Chapter 33

Convex minimization

This section is about the general, *continuous* optimization problem

$$\text{minimize } f(x) \text{ over } x \in \mathbb{R}^n,$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a real-valued function. We want to design algorithms as general as possible - consequently, we will try to identify the minimum structure necessary to make the above problem tractable (step 1: convexity), and only query f in a black-box manner.

All of the algorithms presented here have been extended to more substantially more sophisticated settings (e.g., constrained optimization) and also improved (e.g., accelerated methods). Our primary goal to expose the reader to some of the principles and approaches of continuous optimization, while introducing a relatively small number of parameters and definitions. We refer the reader to [Nes18] for an advanced and comprehensive text on continuous optimization.

Convex minimization is a basic and practical algorithmic problem. However our discussion be in a different language from previous discussions, due to the *continuous* nature of the analysis. It may take some getting use to. There will be some basic mathematical objects (e.g., vector spaces, linear maps, and the Hessian) that not all CS students have background in. Here we can only give a brief introduction. We recommend [Axl15] for an introduction to linear algebra and [BL00] for an introduction to convex analysis. In the long run, you will want to develop a balanced skill set and appreciation of both the continuous and discrete perspectives.

33.1 Preliminaries

Let $\mathbb{R}^n$ be the set of real-valued, n-dimensional *vectors* of the form

$$x = (x_1, \ldots, x_n)$$

where $x_1, \dots, x_n \in \mathbb{R}$. Two vectors $x, y \in \mathbb{R}^n$ sum to make a vector $x + y \in \mathbb{R}^n$, defined by

$$x + y = (x_1 + y_1, \dots, x_n + y_n).$$

For $a \in \mathbb{R}$ and $x \in \mathbb{R}^n$, we can *scale* x by a to obtain the vector

$$ax = (ax_1, \dots, ax_n).$$

Note that $a(x + y) = ax + ay$ for $a \in \mathbb{R}$ and $x, y \in \mathbb{R}^n$.

The *inner product* of two vectors $x, y \in \mathbb{R}^n$, denoted $\langle x, y \rangle$, is the quantity $\langle x, y \rangle = \sum_{i=1}^n x_i y_i$. The *Euclidean norm* of a vector x, denoted $\|x\|$, is defined by

$$\|x\| = \sqrt{\langle x, x \rangle} = \sqrt{\sum_{i=1}^n x_i^2}.$$

Two helpful inequalities to keep in mind are as follows. The *triangle inequality* states that for all $x, y \in \mathbb{R}^n$,

$$\|x - y\| \leq \|x\| + \|y\|.$$

The Cauchy-Schwartz inequality states that for all $x, y \in \mathbb{R}^n$,

$$\langle x, y \rangle \leq \|x\| \|y\|.$$

We note that most of our discussion generalizes to more general normed vector spaces, but we will stick to Euclidean space for simplicity.

33.2 Convex functions

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a real-valued function. f is *convex* if all $x, y \in \mathbb{R}^n$, and all $\lambda \in [0, 1]$, we have

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda) f(y).$$

Some simple, one-dimensional examples of convex functions include:

- $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x^2$.
- $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = |x|$.

Some higher dimensional examples include $f : \mathbb{R}^n \to \mathbb{R}$ defined by $f(x) = \langle x, x \rangle$, or more generally, $f(x) = \langle Ax, Ax \rangle$, where $A \in \mathbb{R}^{m \times n}$ is a matrix.

Certain combinations of convex functions are also convex. If $f : \mathbb{R}^n \to \mathbb{R}$ is convex and $\alpha > 0$, then αf is convex. If $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}$ are both convex, then $f + g$ is convex. If $f : \mathbb{R}^n \to \mathbb{R}$ is convex, and $A : \mathbb{R}^m \to \mathbb{R}^n$ is a linear map, then the function $g(y) = f(Ay)$ is convex.

Derivatives. Our definition above did not require f to be continuously differentiable. In fact, the example $f(x) = |x|$ is not differentiable at $x = 0$. Henceforth, we assume that all our functions f are continuously differentiable. Algorithmically, we will also assume that we are able to compute the derivative of f.

Recall that the *derivative* of f is given by a vector

$$f'(x) \in \mathbb{R}^n, \text{ where } f'_i(x) = \frac{d}{dx_i} f(x) \text{ for each } i.$$

One way to interpret $f'(x)$ is as follows. Fix an input point $x \in \mathbb{R}^n$, as well as a direction $u \in \mathbb{R}^n$. Consider the one-variable function $g(t)$ defined by

$$g(t) = f(x + tu).$$

Then by the chain rule, we have

$$g'(t) = \langle f'(x + tu), u \rangle.$$

In particular, at $t = 0$,

$$g'(0) = \langle f'(x), u \rangle$$

is the (infinitesimal) change in f from moving in the direction u.

Another interpretation, building on the last one, is as follows. Let $x_0, x_1 \in \mathbb{R}^n$ be two input points. For $t \in \mathbb{R}$, let $x_t = tx_1 + (1 - t)x_0$. Consider the function $g(t)$ defined by

$$g(t) \stackrel{\text{def}}{=} f(x_t) = f(tx_1 + (1 - t)x_0).$$

In particular, from $t = 0$ to 1, $g(t)$ traces the value $f(x_t)$ along the line segment from x_0 to x_1. We have

$$f(x_1) - f(x_0) = g(1) - g(0) = \int_0^1 g'(t)\, dt = \int_0^1 \langle f'(x_t), x_1 - x_0 \rangle\, dt.$$

The point is that the rate of change $\langle f'(x_t), x_1 - x_0 \rangle$ along the line segment from x_0 to x_1 adds up to the total difference $f(x_1) - f(x_0)$.

Derivatives of convex functions. We now examine the case where $f : \mathbb{R}^n \to \mathbb{R}$ is continuously differentiable *and* convex.

Lemma 33.1. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be differentiable. Then f is convex iff for all $x, y \in \mathbb{R}^m$,*

$$f(y) \geq f(x) + \langle f'(x), y - x \rangle. \tag{33.1}$$

Proof. Suppose f is convex. We have

$$\langle f'(x), y - x\rangle \overset{\text{(a)}}{=} \frac{f(x + \lambda(y-x)) - f(x)}{\lambda} = \frac{f((1-\lambda)x + \lambda y) - f(x)}{\lambda}$$
$$\overset{\text{(b)}}{\leq} \frac{(1-\lambda)f(x) + \lambda f(y) - f(x)}{\lambda} \overset{\text{(c)}}{=} f(y) - f(x),$$

which gives the desired inequality up to rearrangement. Here (a) is the chain rule. (b) is by definition of the derivative. *(c) is by convexity.*

Conversely, suppose f satisfies the inequality (33.1). Let $x_0, x_1 \in \mathbb{R}^n$. Let $x_t = tx_1 + (1-t)x_0$. We want to show that $f(x_t) \leq tf(x_1) + (1-t)f(x_0)$. We have

$$f(x_1) \geq f(x_t) + \langle f'(x_t), x_1 - x_t\rangle = f(x_t) + \langle f'(x_t), (1-t)(x_1 - x_0)\rangle,$$

and

$$f(x_0) \geq f(x_t) + \langle f'(x_t), x_0 - x_t\rangle = f(x_t) + \langle f'(x_t), t(x_0 - x_t)\rangle,$$

by assumption. By adding t times the first inequality and $(1-t)$ times the second inequality, we have

$$tf(x_1) + (1-t)f(x_0) \geq f(x_t),$$

as desired. □

First-order conditions of optimality. We say that a point x is a *global minimum* of f if $f(x) \leq f(y)$ for all y. We point out that, because the input space is continuous, it is impossible to tell if x is the global minimum by querying values $f(x)$ alone. (What if we nudge x a little bit this way? Or that way?) The following theorem is thus extremely important: it gives us a *certificate* of optimality.

Theorem 33.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable and convex. A point x is a global minimum of f iff $f'(x) = \mathbb{0}$.*

Proof. Suppose $f'(x) = \mathbb{0}$. Then for all y, we have

$$f(y) \overset{\text{(a)}}{\geq} f(x) + \langle f'(x), y - x\rangle \overset{\text{(b)}}{=} f(x).$$

Here (a) is convexity (i.e., Lemma 33.1). (b) is because $f'(x) = \mathbb{0}$.

Conversely, suppose x is the global minimum. For all u, we have

$$\langle f'(x), u\rangle = \lim_{t \to 0} \frac{f(x + tu) - f(x)}{t} \overset{\text{(c)}}{\geq} 0.$$

Here (c) is because $f(x + tu) \geq f(x)$. In particular, for $u = -f'(x)$, we have

$$-\|f'(x)\|^2 \geq 0,$$

hence $\|f'(x)\| \leq 0$. But this implies that $f'(x) = \mathbb{0}$. □

33.3 Gradient descent for smooth functions

Lipschitz-continuity and smoothness. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function. For a fixed parameter $L > 0$, f is *L-Lipschitz continuous* if we have

$$|f(x_0) - f(y)| \leq L\|x_0 - y\|$$

for all x_0, y. The *derivative of f* is *L-Lipschitz* continuous if we have

$$\|f'(x_0) - f'(x_1)\| \leq L\|x - y\|.$$

In the convex optimization literature, f is called *L-smooth* if its derivative is L-Lipschitz.

Lemma 33.3. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be L-smooth, and let $x_1, x_0 \in \mathbb{R}^n$. Then*

$$f(x_1) \leq f(x_0) + \langle f'(x_0), x_1 - x_0\rangle + \frac{L}{2}\|x_1 - x_0\|^2.$$

Proof. For $t \in [0, 1]$, let $x_t = tx_1 + (1 - t)x_0$. We have

$$\begin{aligned}
f(x_1) - f(x_0) - \langle f'(x_0), x_1 - x_0\rangle &\overset{\text{(a)}}{=} \int_0^1 \frac{d}{dt} f(x_t)\, dt - \langle f'(x_0), x_1 - x_0\rangle \\
&\overset{\text{(b)}}{=} \int_0^1 \langle f'(x_t) - f'(x_0), x_1 - x_0\rangle, dt \\
&\overset{\text{(c)}}{\leq} \int_0^1 \|f'(x_t) - f'(x_0)\|\|x_1 - x_0\|\, dt \\
&\overset{\text{(d)}}{\leq} \int_0^1 \|x_t - x_0\|\|x_1 - x_0\|\, dt \\
&\overset{\text{(e)}}{=} \|x_1 - x_0\|^2 \int_0^1 t\, dt = \frac{1}{2}\|x_1 - x_0\|^2.
\end{aligned}$$

Here (a) is the fundamental theorem of calculus. (b) is the chain rule. (c) is by the Cauchy-Schwartz inequality. *(d) is because f is L-smooth.* (e) observes that $x_t - x_0 = t(x_1 - x_0)$. □

33.3.1 Gradient step for smooth functions

Let $f : \mathbb{R}^n \to \mathbb{R}$ be an L-smooth function. Then, as observed above, the function

$$g(y) = f(x) + \langle f'(x), y - x\rangle + \frac{L}{2}\|y - x\|^2$$

gives an upper bound for $f(y)$ for all y. Moreover, $g(y)$ is *not* a black box, and rather simple. Thus a natural *algorithmic idea* to improve from a point x is to identify the point y minimizing $g(y)$. Let us denote

$$x^+ = \arg\min_y f(x) + \langle f'(x), y - x\rangle + \frac{L}{2}\|x - y\|^2. \tag{33.2}$$

We can interpret x^+ as a *local search* heuristic (pending rigorous analysis). Given a current point x, we use some local information (namely $f'(x)$) to find a new point x^+ that we hope is an improvement on x.

Lemma 33.4. *Let f be L-smooth and $x \in \mathbb{R}^n$. Let x^+ be defined by (33.2). Then*

$$x^+ = x - \frac{f'(x)}{L}.$$

Moreover,

$$f(x^+) \le f(x) - \frac{1}{2L}\|f'(x)\|^2.$$

Proof. The function $g(y)$ defined above is convex in y, and the first-order conditions say it is minimized when

$$g'(y) = f'(x) + L(y - x) = 0 \iff y = x - \frac{1}{L}f'(x).$$

The RHS gives x^+.

Moreover, we have

$$\begin{aligned} f(x^+) &\overset{(a)}{\le} f(x) + \langle f'(x), x^+ - x\rangle + \frac{L}{2}\|x^+ - x\|^2 \\ &\overset{(b)}{=} f(x) - \frac{1}{L}\|f'(x)\|^2 + \frac{1}{2L}\|f'(x)\|^2 \\ &= f(x) - \frac{1}{2L}\|f'(x)\|^2, \end{aligned}$$

as desired. Here (a) is because L is smooth. (b) substitutes $x^+ = x - f'(x)/L$. □

Minimizing the gradient. A point x is a *critical point* of f is $f'(x) = \mathbb{0}$. Above we observed that critical points of a convex function are a global minimizer. The following theorem gives an algorithm for minimizing the gradient $\|f'(x)\|$. Note that the following does not require f to be convex.

Theorem 33.5. *Let $f : \mathbb{R}^n \to \mathbb{R}$ be an L-smooth function. Let $\epsilon > 0$ and x_0 be given. Let $x^\star$ be a global minimizer for f. Let $x_0, x_1, \cdots \in \mathbb{R}^n$ be defined by $x_{t+1} = x_t^+$ (per equation (33.2) above). Then for all $T \in \mathbb{N}$,*

$$\min_{t=1,\dots,T} \|f'(x_t)\|^2 \leq \frac{2L(f(x_0) - f(x^\star))}{T}.$$

Proof. We have

$$\begin{aligned} f(x_0) - f(x^\star) \geq f(x_0) - f(x_T) &\overset{(a)}{=} \sum_{t=1}^{T} f(x_{t-1}) - f(x_t) \\ &\overset{(b)}{\geq} \sum_{t=1}^{T} \frac{\|f'(x_t)\|^2}{2L} \geq \frac{T}{2L} \min_{t=1,\dots,T} \|f'(x_t)\|^2, \end{aligned}$$

as desired. Here (a) is by telescoping series. (b) is by Lemma 33.4. □

33.4 Gradient descent for smooth and convex functions

In this section, we continue to let $f : \mathbb{R}^n \to \mathbb{R}$ be an L-smooth for a known parameter $L > 0$. For a point $x \in \mathbb{R}^n$, let x^+ be the gradient step as defined in equation (33.2) above. In addition to being smooth, we assume f is *convex*. Let $x^\star$ be a global minimizer of f.

We want to argue that x^+ makes progress towards optimality. We can measure this in two ways. The first is the difference in objective value between the current point and optimum, $f(x) - f(x^\star)$. The second is in the Euclidean distance in the input space, $\|x - x^\star\|$. For both we will show that x^+ improves x.

The first lemma shows that x^+ is closer to $x^\star$ than x.

Lemma 33.6. *For all x,*

$$\|x^+ - x^\star\|^2 \leq \|x - x^\star\|^2.$$

Proof. First we have

$$\frac{1}{2L}\|f'(x)\|^2 \leq f(x) - f(x^+) \overset{(a)}{\leq} f(x) - f(x^\star) \overset{(b)}{\leq} \langle f'(x), x - x^\star \rangle \tag{33.3}$$

Here (a) is by Lemma 33.4. (b) is because $x^\star$ is the global minimum. (c) is by convexity of f. Now, we have

$$\begin{aligned} \|x^+ - x^\star\|^2 &= \|x - f'(x)/L - x^\star\|^2 \\ &= \|x - x^\star\|^2 - \frac{2}{L}\langle f'(x), x - x^\star \rangle + \frac{1}{L^2}\|f'(x)\|^2 \\ &\overset{(c)}{\leq} \|x - x^\star\|^2. \end{aligned}$$

Here (d) is by the inequality obtained in (33.3). □

The next lemma is about the decrease in objective value. Previously we showed that the decrease in objective is proportional to $\|f'(x)\|^2$. The following lemma goes one step further, showing that the decrease in $f(x)$ is proportional to the current gap from the optimum squared, $(f(x) - f(x^\star))^2$.

Lemma 33.7. *For all x,*

$$f(x) - f(x^+) \geq \frac{1}{2L}\left(\frac{f(x) - f(x^\star)}{\|x - x^\star\|}\right)^2.$$

Proof. We have

$$f(x) - f(x^\star) \overset{(a)}{\leq} \langle f'(x), x - x^\star \rangle \overset{(b)}{\leq} \|f'(x)\|\|x - x^\star\| \overset{(c)}{\leq} \|x - x^\star\|\sqrt{2L(f(x) - f(x^+))}.$$

Here (a) is by convexity. (b) is by Cauchy-Schwartz. (c) is by Lemma 33.4. Squaring both sides (where we note that both sides are nonnegative) and rearranging gives the desired inequality. □

33.4.1 Gradient descent

We now extend our analysis to an overall algorithm for minimizing a convex function. In this section, Let $f : \mathbb{R}^n \to \mathbb{R}$ be a convex and L-smooth function. Let $x^\star$ be a global minimizer for f.

Consider the following algorithm that takes as input an initial point $x_0 \in \mathbb{R}^n$. We compute a sequence of points $x_0, x_1, x_2, \ldots$, where x_{t+1} takes a gradient step from x_t; i.e.,

$$\begin{aligned} x_{t+1} = x_t^+ &= \arg\min_y f(x_t) + \langle f'(x_t), y - x_t \rangle + \frac{L}{2}\|y - x\|^2 \\ &= x_t - \frac{1}{L} f'(x_t). \end{aligned}$$

Then $f(x_t)$ converges to $f(x_0)$ at the following rate.

Theorem 33.8. *For $T = O\Big(L\|x_0 - x^\star\|^2/\epsilon\Big)$, we have*

$$f(x_T) - f(x^\star) \leq \epsilon.$$

Proof. We have

$$f(x_t) - f(x_{t+1}) \overset{\text{(a)}}{\geq} \frac{\left(f(x_t) - f(x^\star)\right)^2}{2L\|x_t - x^\star\|^2} \overset{\text{(b)}}{\geq} \frac{\left(f(x_t) - f(x^\star)\right)^2}{2L\|x_0 - x^\star\|^2}.$$

Here (a) is by Lemma 33.7 and (b) is by Lemma 33.6.

Now, fix $\delta > 0$. Suppose the error is bounded above by $f(x_t) - f(x^\star) \leq 2\delta$, and consider the number of iterations until $f(x_t) - f(x^\star) \leq \delta$. As long as $f(x_t) - f(x^\star) \geq \delta$, the decrease in error is at least

$$f(x_t) - f(x_{t+1}) \geq \frac{\delta^2}{2L\|x_0 - x^\star\|^2}.$$

Thus there can be at most $2L\|x_0 - x^\star\|^2/\delta$ iterations before has decreased from at most 2δ to below δ.

Our goal is to understand the number of iterations to reduce the error to $\epsilon > 0$. For each i, let $\epsilon_i = 2^i\epsilon$. Then for each i, we have at most $O\Big(L\|x_0 - x^\star\|^2/\epsilon_i\Big)$ iterations where the error, $f(x_t) - f(x^\star)$, is between ϵ_i and ϵ_{i-1}. Thus the total number of iterations to reach error $\epsilon = \epsilon_0$ is at most

$$\sum_{i=0}^{\infty} O\left(\frac{L\|x_0 - x^\star\|^2}{\epsilon_i}\right) = O\left(\frac{L\|x_0 - x^\star\|^2}{\epsilon}\right) \sum_{i=0}^{\infty} \frac{1}{2^i} \leq O\left(\frac{L\|x_0 - x^\star\|^2}{\epsilon}\right),$$

as desired. □

33.5 Exercises

Exercise 33.1. Prove that for all $x, y \in \mathbb{R}^n$,

$$\|x + y\|^2 = \|x\|^2 + \|y\|^2 + 2\langle x, y\rangle.$$

Exercise 33.2. Prove that, for all $a, b \in \mathbb{R}$,

$$2ab \leq a^2 + b^2.$$

Exercise 33.3. Prove the AM-GM inequality (for two variables): For two variables $a, b > 0$,

$$\sqrt{ab} \leq \frac{a+b}{2}.$$

Exercise 33.4. Prove the Cauchy-Schwartz inequality.

Exercise 33.5. Prove the triangle inequality.

Exercise 33.6. Prove the following inequality.

$$\|x\| - \|y\| \leq \|x - y\|$$

for all x, y.

Exercise 33.7. Let $f : \mathbb{R} \to \mathbb{R}$ be a convex and continuously differentiable function over $\mathbb{R}$. You may assume that querying a value $f(x)$, or a gradient $f'(x)$, each take $O(1)$ time.

Consider the problem of minimizing f. Suppose we are promised that there is a unique global minimizer $x^\star$ that lies in the open interval $(0, 1)$. Design and analyze an algorithm that, given an additional parameter $\epsilon > 0$, finds a point such that $x^\star - x \leq \epsilon$. (The running time should depend on ϵ (or rather $1/\epsilon$). As usual, the better the dependency on ϵ, the better.)

Exercise 33.8. Suppose we want to minimize a smooth convex function f, but did not actually know the value of the parameter L. Instead suppose you knew that L was a value between $1/\operatorname{poly}(n)$ and $\operatorname{poly}(n)$. Design and analyze a variation of the gradient descent algorithm in Section 33.3 that does not explicitly use L, but still obtains the same iteration count depending on (the unknown value) L. In particular, one needs to figure out how to choose an appropriate step size since the algorithm we discussed choose the step size as a function of L.

Exercise 33.9. Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be an invertible map such that $\|f(x)\| \geq \mu\|x\|$ for all x. Prove that

$$\left\|f^{-1}(y)\right\| \leq \frac{1}{\mu}\|y\|$$

for all y.

Chapter 34

Greedy Approximations and LP Duality

34.1 Maximum coverage

In the maximum coverage problem, we have m elements $[m] = \{1, \ldots, m\}$, and n sets $A_1, \ldots, A_n \subseteq [m]$. We also have a cardinality constraint $k \in \mathbb{N}$. The goal is to select (at most) k sets $A_{e_1}, A_{e_2}, \ldots, A_{e_k}$ that maximizes the size of the union

$$|A_{e_1} \cup A_{e_2} \cup \cdots \cup A_{e_k}|.$$

This problem is NP-Hard. We will describe the proof next class, when we discuss a closely related problem, set cover.

We will analyze the following *greedy algorithm* for maximum coverage.

> *Starting with an empty collection of sets. For k iterations, select the set that covers the most elements left uncovered by the current collection, and that set to the collection.*

Notation. To analyze the greedy algorithm, it is convenient to adopt the following notation. We identify a set A_e by its index e. We identify a collections of indices $S \subseteq [n]$ with the corresponding collection of sets $\{A_e : e \in S\}$. We define a function $f : 2^{[n]} \to \mathbb{R}_{\geq 0}$ by

$$f(S) \stackrel{\text{def}}{=} (\#\text{ points covered by sets (w/ index) in } S) = \left|\bigcup_{e \in S} A_e\right|.$$

Now, for $i = 1, \ldots, k$, let $e_i \in [n]$ be the index of the ith set selected by the greedy algorithm. Let $S_i = \{e_1, \ldots, e_i\}$ be the first i selected indices. Thus $S_0 = \emptyset$, and S_k is the output of the greedy algorithm.

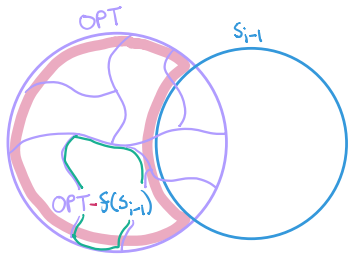
The key lemma. The key lemma analyzes the improvement we make in each iteration. We can interpret Lemma 34.1 below as saying that the greedy algorithm maintains the inequality

$$(\text{improvement}) \geq \frac{1}{k}(\text{room for improvement}),$$

on each iteration.

Lemma 34.1. *For $i = 1, \dots, k$, $f(S_i) - f(S_{i-1}) \geq \frac{1}{k}(\mathrm{OPT} - f(S_{i-1}))$.*

Proof of Lemma 34.1. The proof is illustrated by the following diagram.



There are at least $\mathrm{OPT} - f(S_{i-1})$ elements covered by the optimal solution, but not covered by S_{i-1}. Meanwhile there are k sets in the optimal solution. In particular there is some set in the optimal solution that covers at least $(1/k)$th of the elements. Thus whatever set the greedy algorithm selects adds at least $(1/k)(\mathrm{OPT} - f(S_{i-1}))$ to the total coverage. □

Overall analysis.

Theorem 34.2. $f(S_k) \geq (1 - e^{-1})\,\mathrm{OPT}$.

Proof. We have shown that

$$(i\text{th improvement}) = f(S_i) - f(S_{i-1}) \geq \frac{1}{k}(\mathrm{OPT} - f(S_{i-1})) = \begin{pmatrix} \text{room for improvement} \\ \text{after } i-1 \text{ iterations} \end{pmatrix}.$$

Rearranging, we get

$$\mathrm{OPT} - f(S_i) \leq \left(1 - \frac{1}{k}\right)(\mathrm{OPT} - f(S_{i-1})).$$

That is,

$$\begin{pmatrix} \text{room for improvement} \\ \text{after } i \text{ iterations} \end{pmatrix} \leq \left(1 - \frac{1}{k}\right)\begin{pmatrix} \text{room for improvement} \\ \text{after } i-1 \text{ iterations} \end{pmatrix}.$$

We see that "room for improvement" decays exponentially. Unrolling the above, we have

$$\begin{aligned}
\mathrm{OPT} - f(S_0) &= \mathrm{OPT} \\
\mathrm{OPT} - f(S_1) &\leq \left(1 - \frac{1}{k}\right) \mathrm{OPT} \\
\mathrm{OPT} - f(S_2) &\leq \left(1 - \frac{1}{k}\right)^2 \mathrm{OPT} \\
\mathrm{OPT} - f(S_3) &\leq \left(1 - \frac{1}{k}\right)^3 \mathrm{OPT} \\
&\vdots \\
\mathrm{OPT} - f(S_k) &\leq \left(1 - \frac{1}{k}\right)^k \mathrm{OPT} \\
&\overset{\text{(a)}}{\leq} e^{-1} \mathrm{OPT}\,.
\end{aligned}$$

The last step (a) comes from the inequality $1 + x \leq e^x$ for all x. The final inequality is the claim, up to rearrangement. □

34.2 Linear programs and LP duality

Preliminaries. A matrix $A \in \mathbb{R}^{m \times n}$ is defined by coordinates $A_{i,j} \in \mathbb{R}$ for $i \in [m]$ and $j \in [n]$. A matrix A can be understood as a linear map $A : \mathbb{R}^n \to \mathbb{R}^m$ defined by

$$(Ax)_i = \sum_{j=1}^{n} A_{ij} x_j$$

for all $i \in [m]$. Conversely every linear map from $\mathbb{R}^n$ to $\mathbb{R}^m$ is defined by a matrix.

The *transpose* of a matrix $A \in \mathbb{R}^{m \times n}$ is the matrix $A^T \in \mathbb{R}^{n \times m}$ uniquely defined by[1]

$$\langle y, Ax \rangle = \left\langle A^T y, x \right\rangle$$

for all $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$. The coordinates of the transpose $A^T \in \mathbb{R}^{n \times m}$ are given by

$$A^T_{ij} = A_{ji}$$

for all $i \in [n]$ and $j \in [m]$.

[1] Recall that the inner product $\langle \cdot, \cdot \rangle$ in $\mathbb{R}^n$ is defined by $\langle x, y \rangle = \sum_{i=1}^n x_i y_i$.

Linear programs. Linear programs are *constrained* continuous optimization problems where the goal is to (a) select a vector $x \in \mathbb{R}^n$ that (b) optimizes a linear objective subject to (c) linear equality and inequality constraints. That is, an optimization problem of the form

$$\min/\max \langle b, x \rangle = \sum_{j=1}^{n} b_j x_j \text{ over } x \in \mathbb{R}^n$$
$$\text{s.t. } A_1 x \leq c_1,\ A_2 x = c_2, \text{ and } A_3 x \geq c_3.$$

where A_1, A_2, A_3 are matrices and b, c_1, c_2, c_3 are vectors. Linear programs are useful for modeling real problems where we seek continuous solutions. Next time we will also discuss their use in *relaxations* of integer problems, and how to turn the continuous solutions produced by a linear program into integral solutions, often for NP-Hard problems.

Packing LPs. A *packing LP* is a linear program of the form

$$\max \langle b, x \rangle \text{ over } x \in \mathbb{R}^n_{\geq 0} \text{ s.t.} Ax \leq c. \tag{P}$$

where $A \in \mathbb{R}^{m \times n}_{\geq 0}$, $b \in \mathbb{R}^n_{>0}$, and $c \in \mathbb{R}^m_{>0}$ all have nonnegative coefficients. We let $\textsc{Opt}(\mathsf{P})$ denote the optimum value of the LP $\textsc{Opt}(\mathsf{P})$.

Covering LPs. A *covering LP* is a linear program of the form

$$\min \langle c, y \rangle \text{ over } y \in \mathbb{R}^m_{\geq 0} \text{ s.t. } A^T y \geq b, \tag{C}$$

where[2] $A \in \mathbb{R}^{m \times n}_{\geq 0}$, $b \in \mathbb{R}^n_{>0}$, and $c \in \mathbb{R}^m_{>0}$. Abusing notation, we also let $\textsc{Opt}(\mathsf{C})$ denote the optimum value of the LP $\textsc{Opt}(\mathsf{C})$.

LP duality. LP duality is about the relationship between the linear programs $\textsc{Opt}(\mathsf{P})$ and $\textsc{Opt}(\mathsf{C})$, particular when the matrices and vectors A, b, c are the same for both problems.

Let $x \in \mathbb{R}^n_{\geq 0}$ be any feasible solution to (P) and let $y \in \mathbb{R}^n_{\geq 0}$ be any feasible solution to (C). We have

$$\langle b, x \rangle \overset{\text{(a)}}{\leq} \left\langle A^T y, x \right\rangle \overset{\text{(b)}}{=} \langle y, Ax \rangle \overset{\text{(c)}}{\leq} \langle y, c \rangle.$$

[2]Of course, in (C), we could have written A instead of its transpose A^T, and swapped b and c, which would more closely resemble (P). It is convenient for the subsequent discussion on LP duality for A, b and c to have the same dimensions in (P) and (C).

Here (a) is because $x \geq \mathbb{0}$ and $A^T y \geq b$. (b) is by definition of transpose. (c) is because $y \geq \mathbb{0}$ and $Ax \leq c$. Thus, for a packing problem (P) and a covering problem (C) linked by duality, we have

$$\textsc{Opt}(\mathsf{P}) \leq \textsc{Opt}(\mathsf{C}).$$

We have seen this argument before in several concrete instances: between (s,t)-flow and cuts, and between matchings and vertex covers. The max-flow min-cut theorem states that the maximum (s,t)-flow equals the minimum (s,t)-cut. On the other hand, the maximum cardinality matching does not generally equal the minimum vertex cover (e.g., in a triangle). We ask the same question for packing and covering LP's. When, if ever, is $\textsc{Opt}(\mathsf{P}) = \textsc{Opt}(\mathsf{C})$? The *LP duality theorem* (for packing and covering problems) states that in fact they are always equal.

Theorem 34.3 (LP Duality for packing and covering)**.** $\textsc{Opt}(\mathsf{P}) = \textsc{Opt}(\mathsf{C})$.

Our primary goal for the rest of the discussion is to prove Theorem 34.3. Along the way, we will also develop polynomial time approximation algorithms for (P).

34.3 An algorithmic proof of LP duality

34.3.1 Preliminaries

The partition function. We will use a convex potential function $\pi : \mathbb{R}^m \to \mathbb{R}$, alternatively called the *partition* or *soft-max* function, defined by

$$\pi(x) = \log\left(\sum_i e^{x_i}\right).$$

$\pi(x)$ has several nice properties which we now go over. The first property explains the name "soft-max".

Lemma 34.4. *For all $x \in \mathbb{R}^m$,*

$$\max_i x_i \leq \pi(x_i) \leq \max_i x_i + \log(m).$$

Proof. Suppose (without loss of generality) that $\max_i x_i = x_1$. Then

$$x_1 = \log(e^{x_1}) \leq \log\left(\sum_i e^{x_i}\right) \leq \log(me^{x_1}) = x_1 + \log(m),$$

as desired. □

The second property, below, is about the derivative of $\pi(x)$ and how it is particularly easy to interpret.

Lemma 34.5. *For all x, $\pi'(x) \in \mathbb{R}^m$ is positive and its coordinates sum to 1. That is, $\pi'(x)$ describes a probability distribution over $[m]$.*

Proof. For each $i \in m$, we have

$$\pi'_i(x) = \frac{e^{x_i}}{\sum_{j=1}^m e^{x_j}}.$$

Note that $\pi'_i(x)$ is strictly positive, and the sum over i is 1. □

The third and final property we discuss is similar to the "smoothness" property discussed last lecture. In a *multiplicative* sense, the following lemma says that a small change in x can only induce a small change in $\pi'(x)$.

Lemma 34.6. *Let $x, y \in \mathbb{R}^n$ and $\epsilon > 0$, and suppose $x_i \leq y_i \leq x_i + \epsilon$ for all coordinates i. Then for all i,*

$$e^{-\epsilon}\pi'_i(x) \leq \pi'_i(y) \leq e^{\epsilon}\pi'_i(x).$$

Proof. Recall that

$$\pi'_i(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}.$$

If $x_i \leq y_i \leq x_i + \epsilon$, then

$$e^{x_i} \leq e^{y_i} \leq e^{\epsilon} e^{x_i}$$

for all i. The claim follows. □

One can also show that $\pi(x)$ is convex in x, but we will not actually leverage convexity in our analysis.

Normalization. To simplify notation, we rescale (P) as follows. We first scale b to $\mathbb{1}$ (replacing each A_{ij} with A_{ij}/b_j). We also scale c to $\mathbb{1}$ (replacing each A_{ij} with A_{ij}/c_i). We are thus left with the following normalized packing problem,

$$\text{maximize } \langle \mathbb{1}, x \rangle \text{ over } x \in \mathbb{R}^n_{\geq 0} \text{ s.t. } Ax \leq \mathbb{1} \tag{$\mathsf{P}_{\mathbb{1}}$}$$

and the normalized covering problem,

$$\text{minimize } \langle \mathbb{1}, y \rangle \text{ over } y \in \mathbb{R}^m_{\geq 0} \text{ s.t. } A^T y \geq \mathbb{1}. \tag{$\mathsf{C}_{\mathbb{1}}$}$$

Note that the rescaling did not affect the optimum value of either LP: $\textsc{Opt}(\mathsf{P}) = \textsc{Opt}(\mathsf{P}_{\mathbb{1}})$, and $\textsc{Opt}(\mathsf{C}) = \textsc{Opt}((\mathsf{C}_{\mathbb{1}}))$. For ease of notation we will use $\textsc{Opt}(\mathsf{P})$ to refer to the optimum value of ($\mathsf{P}_{\mathbb{1}}$) and $\textsc{Opt}(\mathsf{C})$ to refer to the optimum value of ($\mathsf{C}_{\mathbb{1}}$).

34.3.2 Lagrangian relaxations

We let Δ^m denote the set of probability vectors over $[m]$; i.e.,

$$\Delta^m = \Big\{ p \in \mathbb{R}^m_{\geq 0} : p_1 + \cdots + p_m = 1 \Big\}.$$

For a probability vector $p \in \Delta^m$, consider the packing problem obtained by using p to collapse the packing constraints $Ax \leq \mathbb{1}$ into a single packing constraint $\langle p, Ax \rangle \leq 1$.

$$\text{maximize } \langle \mathbb{1}, x \rangle \text{ over } x \in \mathbb{R}^n_{\geq 0} \text{ s.t. } \langle p, Ax \rangle \leq 1. \tag{L}$$

For fixed p, we let $\textsc{Opt}(\mathsf{L})$ denote the objective value to (L).

Observe that $\langle p, Ax \rangle$ is a convex combination of packing constraints ($(Ax)_i \leq 1$ for all i) of ($\mathsf{P}_{\mathbb{1}}$). As such, (L) is a *relaxation* of ($\mathsf{P}_{\mathbb{1}}$) – every feasible solution to ($\mathsf{P}_{\mathbb{1}}$) is a feasible solution to (L). It is sometimes called a *Lagrangian relaxation* of ($\mathsf{P}_{\mathbb{1}}$). As a relaxation of ($\mathsf{P}_{\mathbb{1}}$), we automatically have

$$\textsc{Opt}(\mathsf{L}) \geq \textsc{Opt}(\mathsf{P}).$$

Interestingly, we can also relate $\textsc{Opt}(\mathsf{L})$ to $\textsc{Opt}(\mathsf{C})$, as follows.

Lemma 34.7. *For all $p \in \Delta^m$, $\textsc{Opt}(\mathsf{L}) \geq \textsc{Opt}(\mathsf{C})$.*

Proof. Let $\lambda > \textsc{Opt}(\mathsf{L})$. We claim that λp is a feasible solution to ($\mathsf{C}_{\mathbb{1}}$). This claim implies that $\lambda \geq \textsc{Opt}(\mathsf{C})$ for all $\lambda > \textsc{Opt}(\mathsf{L})$. Taking the limit as $\lambda \downarrow \textsc{Opt}(\mathsf{L})$, we obtain $\textsc{Opt}(\mathsf{L}) \geq \textsc{Opt}(\mathsf{C})$, desired.

Now, fix $i \in [m]$. Let $x = \lambda e_i$, where e_i is the vector that is 0 everywhere except the ith coordinate is 1. Since $\langle \mathbb{1}, x \rangle = \lambda > \textsc{Opt}(\mathsf{L})$, we must have $\langle p, Ax \rangle > 1$. Expanding out $\langle A, p \rangle x$, we have

$$\langle p, Ax \rangle = \lambda (A^T p)_i = \Big(A^T (\lambda p) \Big)_i > 1.$$

Thus $\Big(A^T (\lambda p) \Big)_i > 1$ for all i. That is, λp is a feasible solution to ($\mathsf{C}_{\mathbb{1}}$), hence $\langle \mathbb{1}, \lambda p \rangle = \lambda > \textsc{Opt}(\mathsf{C})$. □

34.3.3 Finally, the algorithm

We design an algorithm for ($\mathsf{P}_{\mathbb{1}}$) based on the following potential function that uses π. Let $\epsilon > 0$ be given, and let $\eta = \log(m)/\epsilon$. Consider the function $f : \mathbb{R}^n_{\geq 0} \to \mathbb{R}$ defined by

$$f(x) \stackrel{\text{def}}{=} \frac{1}{\eta} \pi(\eta A x). \tag{34.1}$$

We think of f is a continuous approximation to $\max_i (Ax)_i$, as follows.

Lemma 34.8. *For all x,*

$$\max_i (Ax)_i \leq f(x) \leq \max_i (Ax)_i + \epsilon.$$

Proof. By Lemma 34.4, we have

$$\max_i \eta(Ax)_i \leq \eta f(x) \leq \max_i \eta(Ax)_i + \log(m).$$

Dividing by η gives the claimed inequality. □

Note that the derivative of f,

$$f'(x) = A^T \pi'(\eta Ax) \in \mathbb{R}^n_{\geq 0},$$

can be interpreted as a convex combination of the original packing constraints.

Theorem 34.9. *Let $\epsilon > 0$ and consider an LP of the form ($\mathsf{P}_{\mathbb{1}}$). Suppose that, given any distribution $p \in \Delta_m$, one can compute a $(1-\epsilon)$-approximation to the Lagrangian relaxation*

$$\textit{maximimize } \langle \mathbb{1}, x \rangle \textit{ s.t. } \langle p, Ax \rangle \leq 1. \tag{L}$$

Then in $O(m \log(n)/\epsilon^2)$ iterations, one can compute a point $x \in \mathbb{R}^n_{\geq 0}$ such that

$$\langle \mathbb{1}, x \rangle \geq (1-\epsilon)\textsc{Opt}(\mathsf{C}) \text{ and } Ax \leq \mathbb{1}.$$

Therefore, $\textsc{Opt}(\mathsf{P}) \geq (1-\epsilon)\textsc{Opt}(\mathsf{C})$ for all $\epsilon > 0$ sufficiently small, hence $\textsc{Opt}(\mathsf{P}) = \textsc{Opt}(\mathsf{C})$.

Proof. For simplicity, we will directly show how to obtain a $(1-c\epsilon)$-approximation for some constant $c > 0$. A $(1-\epsilon)$-approximation is obtained by decreasing ϵ appropriately.

We will analyze a discrete algorithm that iteratively solves different Lagrangian relaxations of ($\mathsf{P}_{\mathbb{1}}$). It is convenient for the analysis to also take a continuous point of view. Let $t \in [0,1]$ be a variable that represents time; the algorithm starts at $t = 0$ and terminates at $t = 1$. Each iteration k will increase t by a discrete amount of time $\delta_k > 0$. For $k = 1, 2, \ldots$, we let $t_k = \sum_{\ell < k} \delta_k$ denote the time t at the beginning of the kth iteration.

We build a solution x incrementally over time. To this end, we let $x(t)$ denote the value of x at time t. Thus $x(t_k)$ denotes the value of x at the beginning of the kth iteration. We start from an empty solution, $x(t_1) = x(0) = \mathbb{0}$. Each iteration k, we will define a fixed rate of change $x'(t)$ for all $t \in [t_k, t_{k+1})$. Thus we have $x'(t) = \int_0^t x'(s)\, ds$ for all t.

Fix an iteration k. Consider the optimization problem,

$$\text{maximize } \langle \mathbb{1}, z \rangle \text{ s.t. } \langle f'(x(t_k)), z \rangle \leq 1.$$

Expanding out $f'(x(t_k)) = A^T \pi'(\eta A x(t_k))$ and transposing, *the above problem is the same as the following Lagrangian relaxation of* $(\mathsf{P}_{\mathbb{1}})$*:*

$$\text{maximize } \langle \mathbb{1}, z \rangle \text{ s.t. } \langle \pi'(\eta A x(t_k)), A x(t_k) \rangle \leq 1.$$

Here we recall that $\pi'(\eta A x(t_k)) \in \Delta^m$, by Lemma 34.5. Now, by Lemma 34.7, this relaxation has optimum value at least $\textsc{Opt}(\mathsf{C})$.

We approximately solve the Lagrangian relaxation and obtain a point z_k such that

$$\langle \mathbb{1}, z_k \rangle \geq (1-\epsilon)\textsc{Opt}(\mathsf{C}) \text{ and } \langle \pi'(\eta A x(t_k)), A z_k \rangle \leq 1.$$

Let δ_k large as possible such that $t_k + \delta_k \leq 1$ and

$$\delta_k \eta A z_k \leq \epsilon \mathbb{1}.$$

We set $t_{k+1} = t_k + \delta_k$. For all $t \in [t_k, t_k)$, we update $x(t)$ at a rate of

$$x'(t) = z_k.$$

Claim 1. For all $t \in [0,1]$, $\dfrac{d}{dt}\langle \mathbb{1}, x(t) \rangle \geq (1-\epsilon)\textsc{Opt}(\mathsf{C})$.

At any time t, we have $x'(t) = z_k$ for some k, hence $\frac{d}{dt}\langle \mathbb{1}, x(t) \rangle = \langle \mathbb{1}, x'(t) \rangle \geq (1-\epsilon)\textsc{Opt}(\mathsf{C})$.

Claim 2. For all $t \in [0,1]$*, we have*

$$\frac{d}{dt} f(x(t)) \leq e^{\epsilon}.$$

Fix t. Then $t \in [t_k, t_{k+1})$ for some iteration k. We have

$$\eta A x(t_k) \leq \eta A x(t) \leq \eta A x(t_{k+1}) = \eta A x(t_k) + \delta_k \eta A z_k \overset{\text{(a)}}{\leq} \eta A x(t_k) + \epsilon \mathbb{1}$$

by (a) choice of δ_k. By Lemma 34.6, the inequalities above imply that $f'(x(t)) \leq e^{\epsilon} f'(x(t_k))$. Thus

$$\frac{d}{dt} f(x(t)) = \langle f'(x(t)), z_k \rangle \leq e^{\epsilon} \langle f'(x(t_k)), z_k \rangle \leq e^{\epsilon},$$

as desired.

We have now analyze the rate of change of both the objective, $\langle \mathbb{1}, x(t) \rangle$, and the surrogate function for the constraints, $f(x(t))$, over time t. These translate to the following bounds for $x(1)$ by elementary calculus.

Claim 3. $\langle \mathbb{1}, x(1) \rangle \geq \textsc{Opt}(\mathsf{C})$.

We have

$$\langle \mathbb{1}, x(1) \rangle = \langle \mathbb{1}, x(1) - x(0) \rangle \overset{\text{(b)}}{=} \int_0^1 \frac{d}{dt} \langle \mathbb{1}, x(t) \rangle \, dt \overset{\text{(c)}}{\geq} \int_0^1 (1-\epsilon) \textsc{Opt}(\mathsf{C}) \, dt = (1-\epsilon)\textsc{Opt}(\mathsf{C}).$$

Here (b) is the fundamental theorem of calculus. (c) is by claim 1.

Claim 4. $A(x(1)) \leq (e^\epsilon + \epsilon)\mathbb{1}$.

We have

$$\begin{aligned} \max_i A(x(1)) - \epsilon &\overset{\text{(d)}}{\leq} f(x(1)) - f(x(0)) = \int_0^1 \frac{d}{dt} f(x(t)) \, dt \\ &= \int_0^1 \langle f'(x(t)), x(t) \rangle \, dt \leq \int_0^1 e^\epsilon \, dt = e^\epsilon. \end{aligned}$$

(d) is by Lemma 34.8. (e) is the fundamental theorem of calculus. (f) is the chan rule. (g) is by claim 2. Rearranging, we have

$$\max_i (A(x(1)))_i \leq e^\epsilon + \epsilon.$$

We also $e^\epsilon \leq 1 + 2\epsilon$ for $\epsilon > 0$ sufficiently small, which gives the claim.

Thus at time $t = 1$, $x(1)$ satisfies $\langle \mathbb{1}, x \rangle \geq (1-\epsilon)\textsc{Opt}(\mathsf{C})$ and $Ax \leq (1+3\epsilon)\mathbb{1}$. This means that

$$\left(\frac{1}{1+3\epsilon}\right) x(\mathbb{1})$$

is a feasible solution to ($\mathsf{P}_{\mathbb{1}}$), with objective value

$$\left\langle \mathbb{1}, \left(\frac{1}{1+3\epsilon}\right) x(\mathbb{1}) \right\rangle \geq \frac{1-\epsilon}{1+3\epsilon} \textsc{Opt}(\mathsf{C}).$$

Finally, for $\epsilon > 0$ sufficiently small, we have

$$\frac{1-\epsilon}{1+3\epsilon} \geq 1 - 5\epsilon.$$

We have now show that the algorithm produces an $(1-\epsilon)$-approximation (up to scaling) to ($\mathsf{P}_{\mathbb{1}}$). It remains to show that the number of iterations is at most $O(m \log(m)/\epsilon^2)$. Recall that each iteration k, we choose a step size δ_k such that

$$\eta(Az_k)_i = \epsilon$$

for some coordinate $i \in [m]$. That is, each iteration k, we increase $(Ax(t_k))_i$ by ϵ/η for some coordinate i. At the end, we have $(Ax(1))_i \leq 1 + 3\epsilon$ for all i. It follows that we can have at most $O(m/(\epsilon/\eta)) = O(m \log(m)/\epsilon^2)$ iterations. □

Chapter 35

Random Sums and Graphs

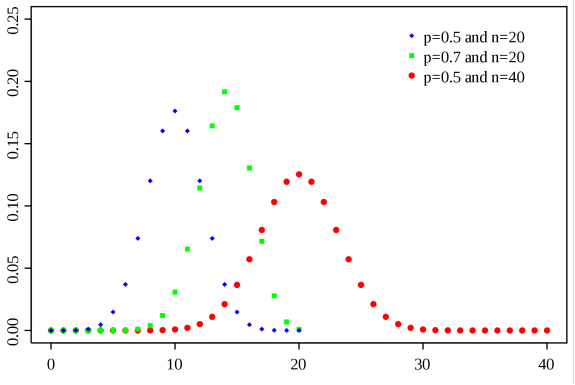
35.1 Random sums

If, out of 100 coin tosses, you were told that 50 of them were heads, would you be surprised? Actually, you should be a little surprised. The odds of getting exactly 50 heads is about 8%. But if you were told that the number was in the range 45 to 55, you probably wouldn't think much of it.

If you were told that all 100 coin tosses came up heads, you wouldn't believe it. The odds of that, we know, is $1/2^{100}$. If you bet money and lost on this event, you would be outraged (and, at even odds, certainly broke for the rest of eternity).

Suppose you were told that at most 25 coin tosses came up heads. Should you be surprised? On one hand, 25 is half of the expected amount. On the other hand, the claim is not that there was exactly 25 heads, but *at most* 25 heads. There could be 25, 24, 23, etc., down to 0. Although probability of getting any one of these counts — being far from average — should be low, there are also 26 of these events. Do the probabilities add up to very much? It turns out that the probability of getting 25 or fewer heads is tiny: about 2.818×10^{-7}.

The *scale* is very important in this discussion. If, out of 10 coin tosses, you got 4 or fewer heads, you shouldn't be too surprised. There is (roughly) a 37.7% chance of getting at most 4 heads. But if at most 40% of 1000 coin tosses came up heads, you should be *very* surprised. The odds of this occurring is occurring is roughly 1.364×10^{-10}.



The point generalizes to coins with any fixed probability of heads, $p \in [0,1]$. The *binomial distribution*, denoted $\mathcal{B}(n,p)$, is the distribution of the number of heads over n independent coin tosses that each flip heads with probability p. The probabilities of different binomial distributions is plotted above. (See also [Wik].)

We write $B \sim \mathcal{B}(n,p)$ to denote a random variable $B \in \{0,\dots,n\}$ drawn from the binomial distribution $\mathcal{B}(n,p)$. The expected value of B is $\mathbf{E}[B] = pn$. The following lemma bounds the probability of B being a multiplicative factor smaller than its mean, pn. Note that the probability decays *exponentially fast in the mean.*

Lemma 35.1. *Let $B \sim \mathcal{B}(n,p)$ and $\epsilon \in (0,1)$. Then*

$$\mathbf{P}[B \le (1-\epsilon)pn] \le e^{-\epsilon^2 pn/2}.$$

Proof. We have

$$\mathbf{P}[B \le (1-\epsilon)pn] = \mathbf{P}\Big[e^{-\epsilon B} \ge e^{-\epsilon(1-\epsilon)pn}\Big] \overset{(a)}{\le} \frac{\mathbf{E}\Big[e^{-\epsilon B}\Big]}{e^{-\epsilon(1-\epsilon)pn}} = e^{\epsilon(1-\epsilon)pn}\,\mathbf{E}\Big[e^{-\epsilon B}\Big] \tag{35.1}$$

by (a) Markov's inequality. It remains to analyze $\mathbf{E}\Big[e^{\epsilon B}\Big]$. Write B as the sum $B = X_1 + \cdots + X_n$, where each X_i is an independent $\{0,1\}$-random variable with $\mathbf{P}[X_i = 1] = p$. Now we have

$$\mathbf{E}\Big[e^{-\epsilon B}\Big] = \mathbf{E}\Big[e^{-\epsilon X_1 - \epsilon X_2 - \cdots - \epsilon X_n}\Big] \overset{(b)}{=} \mathbf{E}\Big[e^{-\epsilon X_1}\Big]\,\mathbf{E}\Big[e^{-\epsilon X_2}\Big]\cdots\mathbf{E}\Big[e^{-\epsilon X_n}\Big],$$

where (b) is because the X_i's are independent. For each X_i, we have

$$\begin{aligned}\mathbf{E}\Big[e^{-\epsilon X_i}\Big] &= pe^{-\epsilon} + (1-p) = 1 + p(e^{-\epsilon} - 1)\\ &\overset{(c)}{\le} p\Big(1 - \epsilon + \epsilon^2/2\Big) + (1-p) = 1 - (\epsilon - \epsilon^2/2)p\\ &\overset{(d)}{\le} e^{-(\epsilon - \epsilon^2/2)p}.\end{aligned}$$

Here (c) uses the inequality $e^{-x} \leq 1 - x + x^2/2$ for all $x > 0$[1]. (d) is by the inequality $1 + x \leq e^x$ for all x. Thus,

$$\mathbf{E}\left[e^{-\epsilon B}\right] = \mathbf{E}\left[e^{-\epsilon X_1}\right] \mathbf{E}\left[e^{-\epsilon X_2}\right] \cdots \mathbf{E}\left[e^{-\epsilon X_n}\right] \leq e^{-(\epsilon - \epsilon^2/2)pn}, . \tag{35.2}$$

Putting everything together, we have

$$\mathbf{P}[B \leq (1-\epsilon)pn] \overset{\text{(e)}}{\leq} e^{\epsilon(1-\epsilon)pn} \mathbf{E}\left[e^{-\epsilon B}\right] \overset{\text{(f)}}{\leq} e^{\epsilon(1-\epsilon)pn - (\epsilon - \epsilon^2/2)pn} = e^{-\epsilon^2 pn/2},$$

by (e) inequality (35.1) and (f) inequality (35.2), as desired. □

One can prove a similar inequality bounding the probability that B exceeds its mean by a multiplicative factor. The proof is similar to Lemma 35.1 and left as exercise 35.2.

Lemma 35.2. *Let $B \sim \mathcal{B}(n,p)$ and $\epsilon \in (0,1)$. Then for all $\mu \geq pn$*

$$\mathbf{P}[B \geq (1+\epsilon)\mu] \leq e^{-\epsilon^2 \mu/3}.$$

(The most common setting of Lemma 35.2 is when $\mu = pn$. But allowing $\mu \geq pn$ can sometimes be convenient.)

35.2 Random graphs

Paul Erdös, inspired by Ramsey [Ram30] before him, had a series of works analyzing *random graphs*. The results can mostly be grouped into two broad categories. First, he designed elaborate randomized constructions of graphs and showed that with nonzero probability, they can possess certain counterintuitive, seemingly impossible properties. This general approach is now called *Ramsey theory*. Second, he showed that for natural random graph models, these graphs – however random – tend to be extremely consistent about certain other properties.

This discussion falls in the latter category. We will study the $\mathcal{G}(n,p)$ random graph, sometimes called Erdös-Rényi graphs based on work by Erdös and Rényi [ER59; ER60]. A random graph from $\mathcal{G}(n,p)$ is an undirected graph over n vertices, where every edge is sampled independently with probability p. By now there is a large

[1]To see that

$$1 - x + x^2/2 \geq e^{-x}$$

for all $x \geq 0$, observe first that both sides equal 1 at $x = 0$. The derivative of the LHS, $-(1-x)$, is always at least the RHS of the derivative of the RHS, $-e^{-x}$, by the inequality $1 + y \leq e^y$ for all y.

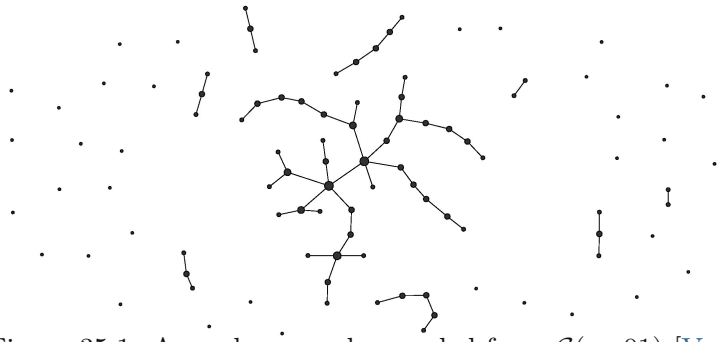


Figure 35.1: A random graph sampled from $\mathcal{G}(n, .01)$ [Von].

catalog of nontrivial and useful properties that, depending on p, are almost certain to appear or not appear in such a graph (for sufficiently large n). Moreover, Erdös and Rényi showed that these properties can vary dramatically with very small changes in p. Consider the following theorem.

Theorem 35.3. *Consider a random graph $G \sim \mathcal{G}(n, p)$ for $p = c/n$, where c is a constant.*

1. *If $c > 1$, then with high probability, there is exactly one connected component of G with $\Omega(n)$ vertices, and all other components have size $\leq O(\log n)$.*
2. *For $c < 1$, then with high probability, all connected components of G have size $< O(\log n)$.*

The parameter c above models the average degree (in expectation). The drama lies in the fact that a tiny change in the average degree c – from .999 to 1.0001 – flips the qualitative nature of a typical random graph from one of many tiny components to essentially one giant component. This is an example of a *threshold phenomena*; alternatively, a *nonlinear dynamic*. Such phenomena is not rare: it occurs in many situations in physics, as well as in models for epidemiology and social networks. Let us briefly mention — without claiming to be very precise — that the sensitivity to c gives some motivation for controlling the "reproductive number" when analyzing and preventing the spread of infectious diseases. The reproductive number is the expected number of healthy individuals that a sick individual effects.

We note that further research has obtained a much more refined and detailed understanding than stated in Theorem 35.3. We refer the reader to [Bol98, Chapter 7] for further details and other results in this area.

We present the proof for $c > 1$. The second case of $c < 1$ is left to the reader as exercise 35.1.

35.2.1 Overview of the proof for $c > 1$

We will prove part 1 of Theorem 35.3 in roughly three parts.

Part 1: the gap theorem. Observe that in Theorem 35.3 above, regardless of the value of p, there are simply no "medium"-size components, such as a component of size $\sqrt{n}$ or of size $n/\log(n)$. The intermediate sizes are ruled out by the following "gap theorem".

Lemma 35.4. *There is a universal constant $C > 0$, such that for all $\epsilon \in (0, 1)$, and for all $n > 0$ sufficiently large, and $p = (1+\epsilon)/n$, we have the following. For a random graph $\mathcal{G}(n, p)$, with probability of error $\leq 1/n^2$, no component has k vertices for any value k in the interval*

$$\frac{C \log(n)}{\epsilon^2} \leq k \leq \frac{\epsilon n}{C}.$$

We analyze Lemma 35.4 theorem in Section 35.3. The proof makes a surprising connection to our discussion on random sums in Section 35.1.

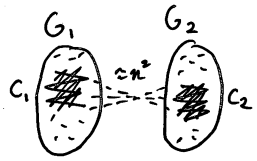
Part 2: existence of a large component. Lemma 35.4 establishes that all components are either very small or very big. However it does not assert that there are any big components. The next theorem, proven in Section 35.4 and based on analyzing a *Galton-Watson branching process*, shows that any single vertex has a reasonable chance of being in a component that is not small.

Lemma 35.5. *Let $p = (1+\epsilon)/n$ for $\epsilon > 0$. Let $v \in V$ be a vertex. For all $3 \leq h \leq \epsilon n$, with probability at least $1/h$, v has at least $1 + h$ vertices in its connected component.*

Lemma 35.5 implies that there is almost certainly at least one giant component as follows. Let $h = c \log(n)/\epsilon^2$ for a sufficiently large constant c, and let $q = 1/h = \Omega(\epsilon^2/\log(n))$. Call a component "small" if it has at most h vertices. We want to argue that, for $p > 1/n$, there is at least one component that is not small. Since the gap theorem (Lemma 35.4) rules out all intermediate sizes, this would imply that there is at least one giant component of size $\Omega(\epsilon^2 n)$.

By Lemma 35.5, any vertex v has at least a probability q of not being in a small component. Now imagine a process where we first randomly select a vertex v and inspect its component. If it is not small, then we have obtained the non-small component we seek. Otherwise, if the component is small, then we throw out v and its component, and randomly select another vertex as v, and repeat. Each vertex we inspect has probability q of not being in a small component. We would have to fail on the order of n/h consecutive samples to conclude there is no small component - which happens with diminishingly small probability. Thus with very high probability, there is at least one component that is not small.

Part 3: uniqueness of the giant component. Can there be two giant components? The answer is no (with high probability) and here is a quick explanation. Instead of sampling from $\mathcal{G}(n,p)$ directly, we can first sample two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ from $G(n/2, p)$. In the second stage we can sample each cross-edge (v_1, v_2), where $v_1 \in V_1$ and $v_2 \in V_2$, independently with probability p. Now, by applying the theory we have already developed to G_1 and G_2, G_1 and G_2 will have some giant components, each of size $\Omega(\epsilon^2 n)$. Note that each graph can only have $O(1/\epsilon^2)$ of them.



Let C_1 be a giant component in G_1 and let C_2 be a giant component in G_2. We can sample up to $|C_1||C_2| \geq \Omega(\epsilon^4 n^2)$ edges between C_1 and C_2. Recalling that p is greater than $1/n$, the odds that all $\Omega(\epsilon^4 n^2)$ edges fail to be sampled is vanishingly small.[2] That is, we almost certainly connect C_1 and C_2. Since there are so few giant components, we will almost certainly connect all of them together. Thus, for $p > (1+\epsilon)/n$ for $\epsilon > 0$, we get a *unique* giant component. This establishes Theorem 35.3 for $c > 1$.

35.3 A gap in component size

In this section we prove Lemma 35.4, which asserts that when $p = (1+\epsilon)/n$ for a constant $\epsilon > 0$, then with high probability, all components are either very small or very large. Our analysis follows an approach due to Karp [Kar90]. His proof is also described in [Bol98]. We first restate Lemma 35.4 for the reader's convenience.

Lemma 35.4. *There is a universal constant $C > 0$, such that for all $\epsilon \in (0,1)$, and for all $n > 0$ sufficiently large, and $p = (1+\epsilon)/n$, we have the following. For a random graph $\mathcal{G}(n,p)$, with probability of error $\leq 1/n^2$, no component has k vertices for any value k in the interval*

$$\frac{C\log(n)}{\epsilon^2} \leq k \leq \frac{\epsilon n}{C}.$$

For a vertex $v \in V$, let $C(v) \subset V$ be the (randomized) component of v. To analyze $C(v)$, we imagine revealing $C(v)$ by a search algorithm. We maintain a collection

[2] We fail with probability $(1-p)^{\Omega(\epsilon^4 n^2)} \leq e^{-\Omega(\epsilon^4 n)}$.

of vertices known to be connected to v; initially just $\{v\}$. Each iteration i, starting from v, select a vertex v_i that is known to be in $C(v)$, but has not been explored. Then "explore" v_i by inspecting all of the edges incident to v_i, possibly adding to the collection of vertices known to be connected to v (but not yet explored).

We annotate this process as follows. For $i \in \mathbb{N}$, let

- v_i be the vertex that is explored in the ith iteration (or `nil` if all of $C(v)$ has already been explored).

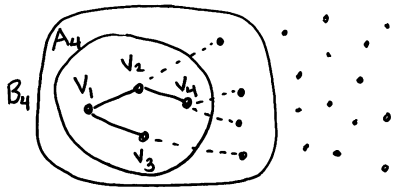
For each $i \in \mathbb{Z}_{\geq 0}$, let

- A_i be the set of vertices known to be in $C(v)$ after i iterations, and let
- B_i be the set of vertices that have been explored.

For the sake of concreteness, one can imagine processing the v_i's in BFS order. Recall that BFS marks each vertex when the vertex is first encountered, and if the vertex was unmarked, it is added to a queue. The next vertex visited is drawn from the queue. In terms of BFS, then, A_i is the set of vertices marked after i iterations, and B_i is the set of vertices that have left the queue and have been fully processed.

Ultimately, B_i, A_i, v_i are built up incrementally as follows.

1. Initially, we have $B_0 = \emptyset$ and $A_0 = \{v\}$.
2. In the first iteration, set $v_1 = v$, set $B_1 = \{v_1\}$, and set $A_1 = A_0 \cup N(v_1)$, where $N(v_1)$ is the (randomized) neighborhood of v_1.
3. In the ith iteration, if $B_{i-1} \neq A_{i-1}$, then select (any) $v_i \in A_{i-1} \setminus B_{i-1}$. Set $B_i = B_{i-1} \cup \{v_i\}$ and $A_i = A_{i-1} \cup N(v_i)$. Otherwise we terminate with $C(v) = B_{i-1} = A_{i-1}$.



The process terminates when $B_i = A_i$. But since $B_i \subseteq A_i$ and $|B_i| = i$, this is precisely when $|A_i| = i$. As long as $|A_i| > i$, A_{i+1} is generated by taking the union of A_i and a random sample of $V - A_i$ where each vertex is included with probability p. Thus we could have generated the sequence of A_i's instead by the following equivalent process, which omits any mention of B_i or v_i.

1. Initially set $A_0 = \{v\}$.
2. For each $i \in \mathbb{N}$, let S sample each vertex in $V \setminus A_{i-1}$ independently with probability p and set $A_i = A_{i-1} \cup S$.
3. Let i be the first index such that $|A_i| = i$, and return $C(v) = A_i$.

Fix an iteration i. The alternative (but equivalent) process described above exposes a simple distribution for A_i. For any vertex $x \neq v$, we have $x \notin A_i$ iff x failed to be added in each of the first i rounds, which occurs with probability exactly $(1-p)^i$. Moreover this event is independent across vertices. Thus $|A_i|$ is distributed exactly as the binomial distribution with $n-1$ coins and probability $1-(1-p)^i$; i.e., $|A_i| \sim \mathcal{B}(n-1, 1-(1-p)^i)$.

We first bound $\mathbf{E}[|A_i|]$.

Lemma 35.6. *Let $i \leq \epsilon n/2(1+\epsilon)$. Then $\mathbf{E}[|A_i|] \geq (1+\epsilon/2)i$.*

Proof. We have

$$(1-p)^i \leq e^{-ip} \leq 1 - ip + \frac{1}{2}(ip)^2 \overset{(a)}{\leq} 1 - ip + \epsilon ip/4 = 1 - (1-\epsilon/4)ip.$$

where (a) is because $ip = (1+\epsilon)i/n \leq \epsilon/2$. Thus

$$\mathbf{E}[|A_i|] = 1 + \Big(1-(1-p)^i\Big)(n-1) \geq (1-\epsilon/4)ipn \overset{(b)}{\geq} (1+\epsilon/2)i.$$

For (b) we observe that $(1-\epsilon/4)(1+\epsilon) \geq (1+\epsilon/2)$ for $\epsilon > 0$ sufficiently small. □

Now we analyze the concentration of $|A_i|$ around its mean.

Lemma 35.7. *Let $i \leq \epsilon n/2(1+\epsilon)$. Then $\mathbf{P}[|A_i| \leq i] \leq e^{-\epsilon^2 i/8}$.*

Proof. We have

$$\mathbf{P}[|A_i| \leq i] \overset{(a)}{\leq} \mathbf{P}[|A_i| \leq (1-\epsilon/2)\,\mathbf{E}[|A_i|]] \overset{(b)}{\leq} e^{-\epsilon^2 i/8}.$$

Here (a) is by Lemma 35.6. (b) is by the tail inequality on binomial distributions, Lemma 35.1. □

To complete the proof of the gap theorem, let

$$I = \Big\{ i \in \mathbb{N} : 32\ln(n)/\epsilon^2 \leq i \leq \epsilon n/2((1+\epsilon)) \Big\}.$$

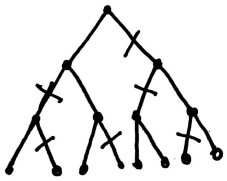


Figure 35.2: A complete binary tree of height 3, where each edge was deleted with probability 1/2.

For all $i \in I$, we have

$$\mathbf{P}[|A_i| \leq i] \leq 1/n^4.$$

By the union bound, we have

$$\mathbf{P}[|A_i| > i \text{ for all } i \in I] \geq 1 - \sum_{i \in I} \mathbf{P}[|A_i| \leq i] \geq 1 - 1/n^3.$$

Thus with probability $\geq 1 - 1/n$, the number of vertices in the connected component of v, $|C(v)|$, does not lie in the range I. Taking the union bound over all $v \in V$ establishes part 1 of Lemma 35.4.

35.4 Galton-Watson process with general branching factors

We now move onto the second part of the analysis. By now we have established that there are (with high probability) no "medium" components – all component sizes have either at most $O(\log(n)/\epsilon^2)$ vertices, or at least $\Omega(\epsilon^2 n)$ vertices. Now we want to prove Lemma 35.5, which we first restate for the reader's convenience.

Lemma 35.5. *Let $p = (1+\epsilon)/n$ for $\epsilon > 0$. Let $v \in V$ be a vertex. For all $3 \leq h \leq \epsilon n$, with probability at least $1/h$, v has at least $1 + h$ vertices in its connected component.*

The proof is by relation to the so-called *Galton-Watson process* that arises in the study of reproducing populations. In the simplest case, imagine a population of size 1. Each generation, each member of the current generation flips k coins, each of which flips heads with probability $1/k$. For each heads, we generate another member of the next generation. The probabilities and number of coins are configured so that each member expects to have one child.

What is the probability that the population survives for h iterations, for a given parameter h? This is answered by the following.

Theorem 35.8. *Let T be a complete k-ary tree of height h, and suppose every edge is deleted independently with probability at most $1 - 1/k$. Then the probability that there is a leaf connected to the root is $\geq 1/h$ for $h \geq 3$, and $\geq (1 - e^{-1})^h$ for $h \leq 2$.*

An example of the case $k = 2$ is drawn in Fig. 35.2.

Proof. For $i \in \mathbb{N}$, let p_i be the probability that a particular node at height i is connected to a subleaf. We have $p_0 = 1$. For a node at height $i + 1$, the probability that there is no path to a leaf via a particular child is

$$1 - \frac{1}{k} + \frac{1}{k}(1 - p_i) = 1 - \frac{p_i}{k}.$$

By independence, we have

$$p_{i+1} = 1 - \left(1 - \frac{p_i}{k}\right)^k.$$

Observe that the RHS is increasing in p_i; thus to lower bound p_{i+1}, we can substitute any lower bound for p_i. We have

$$\begin{aligned}
p_0 &= 1, \\
p_1 &= 1 - (1 - 1/k)^k \geq 1 - e^{-1} \geq .63, \\
p_2 &= 1 - (1 - .63/k)^k \geq 1 - e^{-.63} \geq .467, \\
p_3 &\geq 1 - (1 - .467/k)^k \geq 1 - e^{-.467} \geq .373 \geq 1/3.
\end{aligned}$$

We claim by induction on i that $p_i \geq 1/i$ for all $i \geq 3$. The base case $i = 3$ was just proven. For the general case,

$$p_{i+1} \overset{\text{(a)}}{\geq} 1 - (1 - 1/ik)^k \geq 1 - e^{-1/i} \overset{\text{(b)}}{\geq} \frac{1}{i} - \frac{1}{2i^2} \geq \frac{1}{i+1}$$

Here (a) is by induction. (b) applies the inequality $1 + x \leq e^x$ for all x. (c) applies the inequality $e^x \leq 1 + x + \frac{1}{2}x^2$ for $x \leq 0$. □

35.4.1 Likelihood of small components

We can use the above branching process to analyze the probability that a given vertex v is in a component of size $\geq h$, for any $h \leq \epsilon n/(1 + \epsilon)$.

Fix $h \leq \epsilon n/(1 + \epsilon)$. Let $k = n/(1 + \epsilon)$. Consider a Galton-Watson process where each node produces up to k children, each with probability $1/k$. As proven above, the population lasts h generations with probability at least $1/h$.

Now, consider the following alternative randomized experiment: given the same Galton-Watson process, what is the probability that the populations produces a total of at least h individuals / nodes (including the root)? If the population survives h generations, then there are certainly at least h individuals. So the probability that the population produces at least h individuals is at least h.

We can imagine "revealing" the Galton-Watson outcome as follows. Initially we have the root node, without inspecting which of its k children "survive", and the root node is the only one known to survive. We select that node and reveal which of its k children survive. In general, as long as there is a surviving node where we have not sampled the children, we then sample the k children and reveal which of them survive. This experiment naturally terminates when we have revealed all the children of all the "survivors" and there is nothing left to sample. We also terminate early as soon as we identify at least h survivors.

This version explores the Galton-Watson tree node by node, and does not exploring the branches that die off. As a different pespective on the same phenomena, we have the same conclusion – the experiment produces at least h total (surviving) nodes with probability at least $1/h$.

We will relate this to our random graph as follows. Fix a starting vertex v; we want to analyze the probability that the connected component of v has at least h vertices. Initially, v is (obviously) in the same connected component as v, and we have not yet sampled of the neighbors of v. To explore the (randomized) connected component of v, we choose one vertex that we know is in the same component as v, but have not yet explored, and then reveal/sample all its neighbors. (The first chosen vertex is always v.) The process terminates when we have explored all the vertices known to be connected to w.

Consider the following adjustments to the process which is more conservative. First of all, we terminate immediately once we have identified h vertices connected to v. Second, suppose we have a vertex w that is known to be connected to v, but has not been explored. Since we have identified $< h$ vertices connected to v, there are at least $n - h \geq k$ vertices that are not yet (revealed to be) connected to v. Of these, we select k of them, sample those edges, and add the sampled neighbors to the collection of vertices connected to v.

This adjusted process is conservative relative to the real one – we omit sampling some edges, and we terminate as soon as we find h vertices connected to v. So the probability that the adjusted process finds h vertices connected to v is at most the probability that there are h vertices connected to v in the random graph. Moreover, the adjusted process maps directly to the adjusted k-ary Galton-Watson process mentioned above, where we are only interested in the probability that the population

reaches h total individuals. Here v corresponds to the root, each vertex w connected to v corresponds to a node descending from v in the Galton-Watson tree, and "exploring" from w corresponds to sampling the children of w in the tree.

Putting everything, we conclude that for any $h \leq \epsilon n/(1+\epsilon)$, v is connected to at least h vertices with probability at least $1/h$. This gives us Lemma 35.5.

35.4.2 Directed graphs

One could naturally ask the same questions for directed graphs. Let $D(n,p)$ denote the distribution over *directed graphs* where every directed edge appears independently with probability p. We might similarly ask for the maximum number of vertices reachable from any component, or the size of the maximum strongly component.

It turns out that the analysis of directed graphs can be largely reduced to undirected graphs, as shown by Karp [Kar90] in the following delightfully simple way.

Theorem 35.9. *Let $G \sim G(n,p)$ and $D \sim G(n,p)$, and fix a vertex v. Then the size of the connected component of v in G, and the number of vertices reachable from v in D, are identically distributed.*

Proof. Let us introduce a second distribution of directed random graphs. Let $B(n,p)$ be the distribution of directed graphs where we sample each *undirected* edge $\{u,v\}$ independently with probability p, and for each sampled edge, add both directions (u,v) and (v,u) to the graph. Clearly, for a fixed vertex v, the size of v's (undirected) component in $G(n,p)$ is distributed identically to the number of vertices reachable from v in $B(n,p)$. We claim that the number of vertices reachable from v in $B(n,p)$ is identically distributed as in $D(n,p)$. At this point let us simply quote Karp [Kar90, Lemma 1] (with minor changes in notation) whose proof is very elegant.

> ...To see that the last two random variables are identically distributed, note that the probability spaces $B(n,p)$ and $D(n,p)$ differ in only one respect: a digraph G drawn from $B(n,p)$, are (u,v) is present if and only if arc (v,u) is present, while, in a digraph D drawn from $D(n,p)$, then the event that (v,u) is present is independent of the event that (u,v) is present. Thus no experiment based on checking for the presence or absence of arcs can distinguish between the two probability spaces unless it checks both an arc and its reversal. But any standard sequential algorithm, such as breadth-first search or depth-first search, for building a search tree containing exactly the vertices reachable from vertex 1, checks for the presence of arc (u,v) only if vertex u is in the search tree and v is not;

thus it never checks both an arc and its reversal, and accordingly cannot distinguish $B(n,p)$ from $D(n,p)$.

To summarize the excerpt, standard search algorithms for reachability do not distinguish $B(n,p)$ and $D(n,p)$ anyway, so the number of reachable vertices is identically distributed. □

35.5 Additional notes and materials

Lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2024 lecture notes. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Fall 2022 lecture materials. Click on the links below for the following files:
- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

35.6 Exercises

Exercise 35.1. Prove Item 2 of Theorem 35.3.[3]

Exercise 35.2. Prove Lemma 35.2. (Here the important part is not the constant, $1/3$ – any constant $c > 0$ is already interesting.)

Exercise 35.3. Let $G = (V, E)$ be an undirected graph. Recall that a *proper coloring* is an assignment of colors to V so that the endpoints of any edge are assigned distinct colors. A *k-coloring* is a proper coloring with (at most) k colors. It is NP-Hard to decide if a graph is 3-colorable, while it is very easy to 2-color a graph.

For a fixed constant $\delta \in (0, 1)$, we say that G is *δ-dense* if every vertex has at least $\delta(n-1)$ neighbors. We will design and analyze a randomized algorithm for 3-coloring

[3] It may be helpful to understand the proof of the gap theorem (Lemma 35.4) in Section 35.3.

δ-dense graphs when δ is a fixed constant. (So we are allowed to have running times such as $n^{\text{poly}(1/\delta)}$.)

We will need the following subroutine.

1. (1 point) In the *list k-coloring problem*, each vertex v has a list $L(v) \subset \mathbb{N}$ of at most k colors, and the goal is to decide if there is a proper coloring where each vertex v is assigned a color from its list $L(v)$. The list 2-coloring problem is polynomial time solvable: why?

Henceforth we assume a polynomial time black box for list 2-coloring. (Even if you didn't solve the previous problem.) Let $S \subseteq V$ sample each vertex independently with $p = C\log(n)/\delta n$, for a sufficiently large constant C.

2. (3 points) Prove $|S| \leq O(\log(n)/\delta)$ with high probability.

3. (3 points) Prove that every vertex is either in S or has a neighbor in S with high probability.

4. (3 points) Now, using the previous parts as hints, design and analyze a randomized polynomial time algorithm for 3-coloring a δ-dense graph. (The algorithm should either output a 3-coloring, or declare that the graph is not 3-colorable. If the input graph is 3-colorable, it should compute a 3-coloring with high probability. The exact running time analysis is not important, as long as you can prove it is a polynomial. Don't forget to also prove correctness.)

   The first step of the algorithm samples $S \subseteq V$ as described above. How long does it take to try *every* possible 3-coloring of S? And given a 3-coloring of S, how can you extend it to all of V?

Appendix A

Additional exercises

Appendix B

Basic data structures

We assume the reader has some background in standard data structures; here we briefly review the most basic ones. Additional background can be found in many references including wikipedia.

B.1 Stacks (wiki)

To set an image, imagine a stack of plates. Typically we take plates from the top of the stack, and also put plates on top of the stack. This leads to the property that whenever we take a plate, we always take the plate that was most recently placed on the stack. (This is sometimes refered to as LIFO, an abbreviation for "last-in first-out".)

In algorithms, a *stack* refers to a data structure that models a real-world stack with the following operations.

1. `push(`x`)`: pushes item x on top of the stack.
2. `pop()`: removes and returns the item on top of the stack.

Sometimes stacks provide an `peek`, which returns, but does not remove, the item on top of the stack. Of course `peek` can be implemented as a `pop` followed by a `push`.

Both of the operations can be implemented in $O(1)$ worst-case time. One approach is to use a singly-linked list. In this implementation, we have a *node* for each item in the stack, that represents the place of each. Each node has two pointers:

1. One pointer is for the corresponding item.
2. One pointer is to the next node in the stack (or `null` if it is at the bottom of a stack).

We also maintain a pointer to the node corresponding to the top of the stack. (The pointer is set to `null` if the stack is empty.) A `push(`x`)` creates a new node that points to x and the node at the top of the stack, and points the top of the stack to the new node.A `pop()` look at the node at the top of the stack which has pointers an item x and another node y. Assuming the stack is not empty, `pop()` sets the top of the stack to y and returns x.

Fact B.1. *A stack can be implemented in* $O(1)$ *time per* `push(`x`)` *and* `pop()`.

B.2 Heaps (wiki)

A *heap* implements an order... Here we define a *min-heap* that provides access to the minimum priority pair; a *max-heap* is defined analogously.

1. `insert(`k`,`v`)`: Insert the key k with priority value v.
2. `remove-min`: Return the key-value pair (k, v) with minimum priority v.

Fact B.2. *A min-heap can be implemented in* $O(\log n)$ *time per* `insert` *and* $O(\log n)$ *time per* `remove-min`*, where* n *is the number of items in the heap.*

We should also point out that given n elements, a binary heap over the n elements can be built in $O(n)$ time.

Appendix C

Mathematical Background

C.1 Estimating sums (quickly)

It is not unusual, when analyzing the running time of an algorithm, to encounter a sum (e.g., $\sum_{i=1}^{n} i$), for which we want a quick, *asymptotic* estimate. There is a branch of mathematics that gives sophisticated method to exactly calculate or give very precise estimates for these sums. For us, who are only interested in the $O(\cdots)$, many of these advanced techniques are overkill. Often it is more important to quickly be able to ballpark the sum as this informs our algorithm design. So here we outline a few tricks that suffice for most cases.

C.1.1 The laziest approach.

Take a sum such as $\sum_{i=1}^{n} i$. It is not hard to obtained a closed form solution for this, but it is also a no-brainer to obtain a constant factor estimate. Consider the following simple observations.

1. There are at most n terms, each with value at most n. So $\sum_{i=1}^{n} i \leq n^2$.
2. There are at least $n/2$ terms each with value at least $n/2$. So $\sum_{i=1}^{n} i \geq n^2/4$.

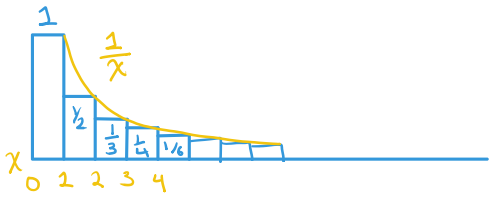
So we have

$$\frac{n^2}{4} \leq \sum_{i=1}^{n} i \leq n^2;$$

that is, $\sum_{i=1}^{n} i = O(n^2)$.

We used this trick when lower-bounding $n!$ in our lower bound for sorting.

C.1.2 The integration trick.



Take a sum such as $\sum_{i=1}^{n} 1/i$. (The "nth harmonic sum".) Observe that for $i \geq 2$, we have

$$\frac{1}{i} \leq \int_{i-1}^{i} \frac{1}{x}\, dx;$$

the RHS represents the uniform average of $1/x$ from $x = i-1$ to i. Consequently

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i} \leq 1 + \sum_{i=2}^{n} \int_{i-1}^{i} \frac{1}{x}\, dx = 1 + \int_{1}^{n} \frac{1}{x}\, dx.$$

Now we know how to integrate the right hand side; recall that $(d/dx)\ln(x) = 1/x$. We have

$$1 + \int_{1}^{n} \frac{1}{x}\, dx = 1 + (\ln(n) - \ln(1)) = 1 + \ln(n).$$

So $\sum_{i=1}^{n} 1/i \leq \ln(n) + 1 = O(\ln(n))$. To see that this is a good estimate, you can similarly apply the integration trick to show that $\sum_{i=1}^{n} 1/i \geq \ln(n)$.

C.1.3 Geometric series.

A *geometric sum* is any sum of the form

$$\sum_{i} \alpha^i$$

for a fixed value α. We almost always have $\alpha > 0$. Now, if $\alpha < 1$, then the terms in the sum are *decreasing*; this is called a *decreasing geometric series*. If $\alpha > 1$, then the terms in the sum are *increasing*; this is called an *increasing geometric series*.

Decreasing geometric series. Suppose $\alpha < 1$. Then even if the sum went from $i = 1$ to $+\infty$ (with infinitely many terms), the sum is still finite. An example of this that you may have seen before is $\alpha = 1/2$: here we are summing

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots$$

Now consider the partial sums of this series. We have

$$\frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \frac{15}{16}, \frac{31}{32}, \ldots$$

Each partial sum halves the gap between the previous partial sum and 1, and the whole sequences converges to 1.

Now suppose in general $\alpha < 1$, and consider the series $\sum_{i=0}^{n} \alpha^i$. Consider $(1 - \alpha) \sum_{i=0}^{n} \alpha^i$. We have We have

$$(1-\alpha)\sum_{i=0}^{n} \alpha^i = \sum_{i=0}^{n} \alpha^i - \alpha^{i+1} \overset{\text{(a)}}{=} 1 - \alpha^{n+1}.$$

Here (a) is because we have a *telescoping series*: the $-\alpha^{i+1}$ term for a fixed i cancels out with the α^i terms for the next value of i. This leaves only the "outer terms" 1 and $-\alpha^{n+1}$. Rearranging we have

$$\sum_{i=0}^{n} \alpha^i = \frac{1 - \alpha^{n+1}}{1 - \alpha} \leq \frac{1}{1-\alpha}.$$

Note that the LHS is increasing in n, but the RHS is an upper bound independent of n. So we can take $n \to \infty$ and obtain

$$\sum_{i=0}^{\infty} \alpha^i \leq \frac{1}{1-\alpha}.$$

Now recall that we are not concerned with exact constants. If α is a fixed constant (like 1/2, 2/3, 3/4, or something else independent of variable parameters such as n), $1/(1-\alpha)$ is also constant, and we simple have

$$\sum_{i=0}^{\infty} \alpha^i \leq \frac{1}{1-\alpha} = O(1).$$

So you can train yourself to automatically recognize that decreasing geometric series are just a constant.

Increasing geometric series. Now suppose $\alpha > 1$; now the terms in the sum $\sum_{i=0}^{n} \alpha^i$ are *increasing*. To build some intuition, suppose $\alpha = 2$. Then the sum has the form

$$1 + 2 + 4 + 8 + \cdots + 2^n.$$

You might have instinctively recognized that the sum is $2^{n+1} - 1$; in particular, largest term represents a large part of the sum.

For the general case we can apply the same telescoping trick. We have

$$(\alpha - 1) \sum_{i=0}^{n} \alpha^i = \sum_{i=0}^{n} \alpha^{i+1} - 1 = \alpha^{n+1} - 1.$$

Thus

$$\sum_{i=0}^{n} \alpha^i = \frac{\alpha^{n+1} - 1}{\alpha - 1}.$$

In particular, when α is a fixed constant (like 2, 3, 4, etc.) the RHS is

$$O(\alpha^n).$$

That is, up to constants, the sum is given by largest term in the series.

Appendix D

Some NP-Complete Problems

Below is a list of some NP-Complete problems that appear in this class. Many of these problems appear in Karp [Kar72].

1. SAT: Given a boolean formula in CNF form, decide if there is a satisfying assignment.
2. 3SAT: Given a boolean formula in CNF form with exactly 3 variables per clause, decide if there is a satisfying assignment.
3. Circuit SAT: Given a boolean circuit, decide if there is a satisfying input.
4. Maximum independent set: Given an undirected graph $G = (V, E)$, find the maximum cardinality set $S \subset V$ such that no two vertices in S share an edge.
5. Clique: Given an undirected graph $G = (V, E)$, find the maximum cardinality set $S \subseteq V$ such that every two vertices in S share an edge.
6. Graph coloring: Given an undirected graph $G = (V, E)$ and an integer k, assign each vertex one of k colors such that no edge has the same color endpoints.
7. Undirected (resp. directed) Hamiltonian cycle: Given an undirected (resp. directed) graph G, decide if there is a cycle containing every vertex exactly once.
8. s-t undirected (resp. directed) Hamiltonian path: Given an undirected (resp. directed) graph G and two vertices s and t, decide if there is a Hamiltonian path (a path visiting each vertex exactly once) from s to t.
9. Traveling salesman: Given a (directed or undirected) graph with positive edge weights, compute a tour (a walk visiting each vertex at least once, starting and ending at the same vertex) of least weight.

10. Subset sum: given a set X of positive integers and an integer T, decide whether X has a subset whose elements sum to T.

11. Planar Circuit SAT: Given a boolean circuit that can be embedded in the plane so that no two wires cross, is there an input that makes the circuit output true?

12. 1-in-3 SAT: Given a 3CNF formula, is there an assignment of values to the variables so that each clause contains *exactly* one True literal?

13. Not-all-equal SAT: Given a 3 CNF formula, is there an assignment of values to the variables so that every clause contains at least true literal and at least one false literal?

14. Planar 3Sat: Given a 3CNF boolean formula, consider a bipartite graph whose vertices are the clauses and variables, where an edge indicates that a variable (or its negation) appears in a clause. If this graph is planar, the 3CNF formula is also called planar. The Planar 3Sat problem asks, given a planar 3CNF formula, whether it has a satisfying assignment.

15. Exact 3 dimensional matching: Let X, Y, Z be three finite collections of elements, and let $T \subseteq X \times Y \times Z$ be a subset of triples (x, y, z) consisting of one element x from X, one element y from Y, and one element z from Z. Find the maximum collection of disjoint triplets. (This problem is similar in spirit to bipartite matching, except the vertex set is divided into 3 parts, and there are "3 vertices in each edge with one from each part," so to speak.)

16. Partition: Given a set of n integers, are there subsets A and B such that $A \cup B = S$, $A \cap B = \emptyset$, and

$$\sum_{a \in A} a = \sum_{b \in B} b?$$

17. 3Partition: Given a set S of $3n$ integers, can it be partition into n disjoint 3-element subsets, such that every subset has exactly the same sum?

18. Set Cover: Given a collection of sets $S = \{S_1, \ldots, S_m\}$, find the smallest sub-collection of S_i's that contains all the elements of $\bigcup_i S_i$.

19. Hitting Set: Given a collection of sets $S = \{S_1, S_2 \ldots, S_m\}$, find the minimum number of elements of $\bigcup_i S_i$ that hit every set in S. This problem is also a generalization of vertex cover.

20. Longest path: given a non-negatively weighted graph G (either directed or undirected) and two vertices u and v, what is the longest path (visiting each vertex at most once) from u to v in the graph?

21. Steiner tree: Given a weighted, undirected graph G with some of the vertices marked, what is the minimum-weight subtree of G that contains every marked vertex?

22. Vertex cover: Let $G = (V, E)$ be an undirected graph. A set of vertices $S \subseteq V$ is a *vertex cover* if every edge $e \in E$ has at least one endpoint in S. Given a graph G and an integer k, is there a vertex cover with at most k vertices?

23. Max 2-Sat: Given a Boolean formula in CNF, with exactly two literals per clause, find a variable assignment that maximizes the number of satisfying clauses.

24. Max cut: Given an undirected graph $G = (V, E)$, find a subset $S \subset V$ that maximizes the number of edges with exactly one endpoint in S.

25. Multicommodity flow: The input consists of either a directed or undirected graph, possibly with integral capacities, and k source-sink pairs $(s_1, t_1), \ldots, (s_k, t_k)$ each with an integral demand $d_i \in \mathbb{N}$. The problem is to pack d_i (s_i, t_i)-paths simultaneously for all i.

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. URL: http://theory.cs.princeton.edu/complexity/.

[ABW15] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. "Tight Hardness Results for LCS and Other Sequence Similarity Measures". In: *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. IEEE Computer Society, 2015, pp. 59–78.

[AH77] K. Appel and W. Haken. "Every planar map is four colorable. Part I: Discharging". eng. In: *Illinois journal of mathematics* 21.3 (1977).

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows - theory, algorithms and applications*. Prentice Hall, 1993.

[And89] Arne Andersson. "Improving Partial Rebuilding by Using Simple Balance Criteria". In: *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*. Vol. 382. Lecture Notes in Computer Science. Springer, 1989, pp. 393–402. URL: https://doi.org/10.1007/3-540-51542-9%5C_33.

[AO89] Ravindra K. Ahuja and James B. Orlin. "A Fast and Simple Algorithm for the Maximum Flow Problem". In: *Oper. Res.* 37.5 (1989), pp. 748–759.

[APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. "A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas". In: *Inf. Process. Lett.* 8.3 (1979), pp. 121–123. URL: https://doi.org/10.1016/0020-0190(79)90002-4.

[AW20] Josh Alman and Virginia Vassilevska Williams. "A Refined Laser Method and Faster Matrix Multiplication". In: *CoRR* abs/2010.05846 (2020). arXiv: 2010.05846. URL: https://arxiv.org/abs/2010.05846.

[Axl15] Sheldon Jay Axler. *Linear Algebra Done Right*. Third. Undergraduate Texts in Mathematics. Springer International Publishing, 2015. URL: https://purdue-primo-prod.hosted.exlibrisgroup.com/permalink/f/1c3q7im/TN_cdi_springer_books_10_1007_978_3_319_11080_6.

[AZ18] Martin Aigner and Günter M. Ziegler. *Proofs from THE BOOK*. eng. 6th ed. 2018. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018.

[Bel58] Richard Bellman. "On a Routing Problem". In: *Quarterly of Applied Mathematics* 16 (1958), pp. 87–90.

[BFP+72] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. "Linear Time Bounds for Median Computations". In: *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*. ACM, 1972, pp. 119–124.

[BG13] Avrim Blum and Anupam Gupta. *Network Flows III: Push-relabel flow algorithms, and Min-Cost Max-Flow*. Lecture 14 of CMU 15-451/651 (Algorithms). Fall 2013. URL: https://www.cs.cmu.edu/~avrim/451f13/lectures/lect1010.pdf.

[BK15] Karl Bringmann and Marvin Künnemann. "Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping". In: *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. IEEE Computer Society, 2015, pp. 79–97. URL: https://arxiv.org/abs/1502.01063.

[BK19] Karl Bringmann and Marvin Künnemann. *Fine-Grained Complexity Theory*. Lecture notes for a course at the Max-Planck-Institut für Informatik. Summer 2019. URL: https://www.mpi-inf.mpg.de/departments/algorithms-complexity/teaching/summer19/fine-complexity/.

[BL00] Jonathan M. Borwein and Adrian S. Lewis. *Convex Analysis and Nonlinear Optimization, Theory and Examples*. 2000.

[Blu00] Avrim Blum. "Paging". Lecture notes from course "Approximation and Online Algorithms". Mar. 2000. URL: http://www.cs.cmu.edu/~avrim/Approx00/lectures/lect0308.

[Blu11a] Avrim Blum. *15-451 Algorithms Lectures 1-10*. 2011. URL: https://www.cs.cmu.edu/~avrim/451f11/lectures/lects1-10.pdf.

[Blu11b] Avrim Blum. "15-451 Algorithms: Lectures 1-10". Course instructed by Avrim Blum and Manuel Blum. Aug. 2011.

[Bol98] Béla Bollobás. *Modern Graph Theory*. 1st ed. Graduate Texts in Mathematics 184. Springer-Verlag New York, 1998.

[Bro41] R. L. Brooks. "On colouring the nodes of a network". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 37.2 (1941), pp. 194–197.

[CG97] Boris V. Cherkassky and Andrew V. Goldberg. "On Implementing the Push-Relabel Method for the Maximum Flow Problem". In: *Algorithmica* 19.4 (1997), pp. 390–410. Preliminary version in IPCO, 1995.

[CGM98] Boris V. Cherkassky, Andrew V. Goldberg, and Paul Martin. "Augment or Push: A Computational Study of Bipartite Matching and Unit-Capacity Flow Algorithms". In: *ACM J. Exp. Algorithmics* 3 (1998), p. 8. Preliminary version in WAE, 1997.

[Cha00] Bernard Chazelle. "A minimum spanning tree algorithm with Inverse-Ackermann type complexity". In: *J. ACM* 47.6 (2000), pp. 1028–1047. Preliminary version in FOCS, 1997.

[Cha13] Timothy M. Chan. "Quake Heaps: A Simple Alternative to Fibonacci Heaps". In: *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*. Vol. 8066. Lecture Notes in Computer Science. Springer, 2013, pp. 27–32. URL: https://doi.org/10.1007/978-3-642-40273-9%5C_3.

[Che14] Chandra Chekuri. "Lecture 6 from CS 598CSC: Algorithms for Big Data". Lecture notes. 2014. URL: https://courses.engr.illinois.edu/cs598csc/fa2014/Lectures/lecture_6.pdf.

[Che15] Chandra Chekuri. *Additional notes on Induction*. 2015. URL: http://chekuri.cs.illinois.edu/teaching/induction-notes.pdf.

[Chu36] Alonzo Church. "An unsolvable problem of elementary number theory". In: *American Journal of Mathematics* 58.2 (Apr. 1936). Available at http://phil415.pbworks.com/f/Church.pdf, pp. 345–363.

[CIP09] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. "The Complexity of Satisfiability of Small Depth Circuits". In: *Parameterized and Exact Computation, 4th International Workshop, IWPEC 2009, Copenhagen, Denmark, September 10-11, 2009, Revised Selected Papers*. Vol. 5917. Lecture Notes in Computer Science. Springer, 2009, pp. 75–85. URL: https://cseweb.ucsd.edu/~paturi/myPapers/pubs/CalabroImpagliazzoPaturi_2009_iwpec.pdf.

[CKL+21] Stephen A. Cook, Richard M. Karp, Leonid A. Levin, Christos H. Papadimitriou, and Avi Wigderson. *50th Anniversary of the Cook-Levin Theorem*. Plenary talk at STOC 2021. 2021. URL: https://www.youtube.com/watch?v=UgGGXXkYqsM.

[CLR+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[CM05] Graham Cormode and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications". In: *J. Algorithms* 55.1 (2005), pp. 58–75. URL: https://doi.org/10.1016/j.jalgor.2003.12.001. Preliminary version in LATIN 2004.

[CM89] Joseph Cheriyan and S. N. Maheshwari. "Analysis of Preflow Push Algorithms for Maximum Network Flow". In: *SIAM J. Comput.* 18.6 (1989), pp. 1057–1086. URL: https://epubs.siam.org/doi/abs/10.1137/0218072?journalCode=smjcat. Preliminary version in FSTTSC, 1988.

[CM99] Joseph Cheriyan and Kurt Mehlhorn. "An Analysis of the Highest-Level Selection Rule in the Preflow-Push Max-Flow". In: *Inf. Process. Lett.* 69.5 (1999), pp. 239–242. URL: https://www.math.uwaterloo.ca/~jcheriya/PDF_files/me3.0.pdf.

[Coo71] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. ACM, 1971, pp. 151–158.

[CQ21] Chandra Chekuri and Kent Quanrud. "Faster Algorithms for Rooted Connectivity in Directed Graphs". In: *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*. Vol. 198. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 49:1–49:16. Full version available at https://arxiv.org/abs/2104.07205.

[CW] Chris 73 and Wikipedia Commons. *WorldWideWeb Around Wikipedia*. URL: https://en.wikipedia.org/wiki/World_Wide_Web#/media/File:WorldWideWebAroundWikipedia.png.

[DD11] Erik Demaine and Srini Devadas. *6.006: Introduction to Algorithms*. Fall 2011. Course website at http://courses.csail.mit.edu/6.006/fall11/staff.shtml. Video lectures posted at https://youtube.com/playlist?list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb.

[DDL15] Erik Demaine, Srini Devadas, and Nancy Lynch. *6.046/18.410: Design and Analysis of Algorithms*. Spring 2015. Course website at https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/. Video lectures posted at https://youtube.com/playlist?list=PLUl4u3cNGP6317WaSNfmCvGym2ucw3oGp.

[Dem12a] Erik Demaine. *Dynamic Graphs 1*. Lecture 20 for MIT 6.851. May 2012. URL: https://courses.csail.mit.edu/6.851/spring12/lectures/L20.html.

[Dem12b] Erik Demaine. *Dynamic Graphs 1*. Lecture 19 for MIT 6.851. May 2012. URL: https://courses.csail.mit.edu/6.851/spring12/lectures/L19.html.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008. Draft version available at https://book.huihoo.com/pdf/algorithms/.

[Edm65] Jack Edmonds. "Paths, Trees, and Flowers". In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467.

[Edm75] Jack Edmonds. "Some Well-Solved Problems in Combinatorial Optimization". In: *Combinatorial Programming: Methods and Applications*. Dordrecht: Springer Netherlands, 1975, pp. 285–301. URL: https://link.springer.com/chapter/10.1007/978-94-011-7557-9_15.

[EIS76] Shimon Even, Alon Itai, and Adi Shamir. "On the Complexity of Timetable and Multicommodity Flow Problems". In: *SIAM J. Comput.* 5.4 (1976), pp. 691–703. Preliminary version in FOCS, 1975.

[EK72] Jack R. Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *J. ACM* 19.2 (1972), pp. 248–264. URL: https://doi.org/10.1145/321694.321699.

[Epp] David Eppstein. *Twenty-one Proofs of Euler's Formula*. URL: https://www.ics.uci.edu/~eppstein/junkyard/euler/.

[ER59] Paul Erdös and Alfred Rényi. "On Random Graphs I". In: *Publicationes Mathematicae Debrecen* 6 (1959), p. 290.

[ER60] Paul Erdös and Alfred Rényi. "On the evolution of random graphs". In: *Publ. Math. Inst. Hungary. Acad. Sci.* 5 (1960), pp. 17–61.

[Eri13a] Jeff Erickson. *Amortized Analysis*. 2013. URL: https://jeffe.cs.illinois.edu/teaching/algorithms/notes/09-amortize.pdf.

[Eri13b] Jeff Erickson. *Scapegoat and Splay Trees*. 2013. URL: https://jeffe.cs.illinois.edu/teaching/algorithms/notes/10-scapegoat-splay.pdf.

[Eri17] Jeff Erickson. *Hashing*. 2017. URL: https://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf.

[Eri19] Jeff Erickson. *Algorithms*. 2019. URL: http://jeffe.cs.illinois.edu/teaching/algorithms/.

[FF56] L. R. Ford and D. R. Fulkerson. "Maximal Flow Through a Network". In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.

[FKL+91] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel Dominic Sleator, and Neal E. Young. "Competitive Paging Algorithms". In: *J. Algorithms* 12.4 (1991), pp. 685–699. URL: https://arxiv.org/abs/cs/0205038.

[Flo62] Robert W. Floyd. "Algorithm 97: Shortest Path". In: *Commun. ACM* 5.6 (June 1962), p. 345.

[For56] L. R. Ford. *Network Flow Theory*. Santa Monica, CA: RAND Corporation, 1956.

[FP] *Functional Pearls*. Wiki article on haskell.org. URL: https://wiki.haskell.org/Research_papers/Functional_pearls.

[Fra11] András Frank. *Connections in combinatorial optimization*. Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, 2011.

[FT87] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *J. ACM* 34.3 (1987), pp. 596–615. URL: https://www.cl.cam.ac.uk/teaching/1112/AlgorithII/1987-FredmanTar-fibonacci.pdf. Preliminary version in FOCS, 1984.

[Gab95] Harold N. Gabow. "A Matroid Approach to Finding Edge Connectivity and Packing Arborescences". In: *J. Comput. Syst. Sci.* 50.2 (1995), pp. 259–273. Preliminary version in STOC 1991.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GJS76] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. "Some Simplified NP-Complete Graph Problems". In: *Theor. Comput. Sci.* 1.3 (1976), pp. 237–267. URL: https://doi.org/10.1016/0304-3975(76)90059-1. Preliminary version in STOC, 1974.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley, 1994. URL: https://www-cs-faculty.stanford.edu/%5C%7Eknuth/gkp.html.

[Gol85] Andrew Vladislav Goldberg. *A New Max-Flow Algorithm*. Tech. rep. MIT/LCS/ TM-291. Cambridge, Massachusetts: Laboratory for Computer Science, Massachusetts Institute of Technology, 1985.

[Gol87] Andrew Vladislav Goldberg. "Efficient graph algorithms for sequential and parallel computers". PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, USA, 1987. URL: http://hdl.handle.net/1721.1/14912.

[GR93] Igal Galperin and Ronald L. Rivest. "Scapegoat Trees". In: *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*. ACM/SIAM, 1993, pp. 165–174. URL: http://dl.acm.org/citation.cfm?id=313559.313676.

[GT88] Andrew V. Goldberg and Robert Endre Tarjan. "A new approach to the maximum-flow problem". In: *J. ACM* 35.4 (1988), pp. 921–940. Preliminary version in STOC, 1986.

[GW] Wereon Gunther and Wikipedia Commons. *Euler's formula*. URL: https://commons.wikimedia.org/wiki/File:Euler%5C%27s_formula.svg.

[Har19] Sariel Har-Peled. *Quick Sort with High Probability*. Lecture notes from Fall 2019 course on randomized algorithms at UIUC. 2019. URL: https://sarielhp.org/teach/19/01_fall/lecs/l19_09_03.pdf.

[HK99] Monika Rauch Henzinger and Valerie King. "Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation". In: *J. ACM* 46.4 (1999), pp. 502–516. URL: https://infoscience.epfl.ch/record/99353/files/HenzingerK99.pdf. Preliminary version in STOC, 1995.

[HLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". In: *J. ACM* 48.4 (2001), pp. 723–760. Preliminary version in STOC 1998.

[HO94] Jianxiu Hao and James B. Orlin. "A Faster Algorithm for Finding the Minimum Cut in a Directed Graph". In: *J. Algorithms* 17.3 (1994), pp. 424–446. Preliminary version in SODA, 1992.

[Ing62] P. Z. Ingerman. "Algorithm 141: Path Matrix". In: *Commun. ACM* 5.11 (Nov. 1962), p. 556.

[IP99] Russell Impagliazzo and Ramamohan Paturi. "Complexity of k-SAT". In: *Proceedings of the 14th Annual IEEE Conference on Computational Complexity, Atlanta, Georgia, USA, May 4-6, 1999*. IEEE Computer Society, 1999, pp. 237–240. URL: http://cseweb.ucsd.edu/~paturi/myPapers/pubs/ImpagliazzoPaturi_2001_jcss.pdf.

[IW] IkamusumeFan and Wikipedia Commons. *A plus bi*. URL: https://commons.wikimedia.org/wiki/File:A_plus_bi.svg.

[Joh77a] Donald B. Johnson. "Efficient Algorithms for Shortest Paths in Sparse Networks". In: *J. ACM* 24.1 (Jan. 1977), pp. 1–13.

[Joh77b] Donald B. Johnson. "Efficient Algorithms for Shortest Paths in Sparse Networks". In: *J. ACM* 24.1 (1977), pp. 1–13. URL: https://doi.org/10.1145/321992.321993.

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103.

[Kar74] Alexander V. Karzanov. "Nakhozhdenie maksimal'nogo potoka v seti metodom predpotokov [Russian; Determining the maximal flow in a network by the method of preflows]". In: *Doklady Akademii Nauk SSSR* 215 (1974), pp. 49–52.

[Kar90] Richard M. Karp. "The Transitive Closure of a Random Digraph". In: *Random Struct. Algorithms* 1.1 (1990), pp. 73–94.

[KKT95] David R. Karger, Philip N. Klein, and Robert Endre Tarjan. "A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees". In: *J. ACM* 42.2 (1995), pp. 321–328.

[KLL+97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*. ACM, 1997, pp. 654–663.

[KMM+94] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan S. Owicki. "Competitive Randomized Algorithms for Nonuniform Problems". In: *Algorithmica* 11.6 (1994), pp. 542–571. URL: https://doi.org/10.1007/BF01189993. Preliminary version in SODA, 1990.

[Knu63] Donald Knuth. *Notes on "open" addressing*. 1963. URL: http://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[KP81] Donald E. Knuth and Michael F. Plass. "Breaking Paragraphs into Lines". In: *Softw. Pract. Exp.* 11.11 (1981), pp. 1119–1184. URL: http://www.eprg.org/G53DOC/pdfs/knuth-plass-breaking.pdf.

[Kro67] M. R. Krom. "The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary". eng. In: *Mathematical logic quarterly* 13.1-2 (1967), pp. 15–20.

[KT06] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.

[Kuh55] Harold W. Kuhn. "The Hungarian Method for the Assignment Problem". In: *Naval Research Logistics Quarterly* 2.1–2 (Mar. 1955), pp. 83–97.

[Lev73] Leonid A. Levin. "Universal Sequential Search Problems". In: *Problems of Information Transmission* 9.3 (1973).

[Mor68] Donald R. Morrison. "PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *J. ACM* 15.4 (Oct. 1968), pp. 514–534.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Nel16] Jelani Nelson. *Load balancing, k-wise independence, chaining, linear probing*. Recorded lecture. 2016. URL: https://www.youtube.com/watch?v=WqBc0ZCU4Uw&list=PL2SOU6wwxB0uP4rJgf5ayhHWgw7akUWSf&index=3.

[Nel20] Jelani Nelson. "Sketching Algorithms". Lecture notes. Dec. 2020. URL: https://www.sketchingbigdata.org/fall20/lec/notes.pdf.

[Nes18] Yurii Nesterov. *Lectures on Convex Optimization*. 2nd ed. Vol. 137. Springer Optimization and Its Applications. Springer International Publishing, 2018. URL: https://purdue-primo-prod.hosted.exlibrisgroup.com/permalink/f/1c3q7im/TN_cdi_springer_books_10_1007_978_3_319_91578_4.

[Neu93] John von Neumann. "First Draft of a Report on the EDVAC". In: *IEEE Ann. Hist. Comput.* 15.4 (Oct. 1993), pp. 27–75. URL: https://en.wikipedia.org/wiki/First_Draft_of_a_Report_on_the_EDVAC.

[NI08] Hiroshi Nagamochi and Toshihide Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Vol. 123. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2008. URL: https://doi.org/10.1017/CBO9780511721649.

[NI92a] Hiroshi Nagamochi and Toshihide Ibaraki. "A linear-time algorithm for finding a sparse k-connected subgraph of a k-connected graph." In: *Algorithmica* 7.1 (596 1992), p. 583.

[NI92b] Hiroshi Nagamochi and Toshihide Ibaraki. "Computing Edge-Connectivity in Multigraphs and Capacitated Graphs". In: *SIAM J. Discret. Math.* 5.1 (1992), pp. 54–66. URL: https://doi.org/10.1137/0405004.

[Oka99] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[Pei13] Chris Peikert. *Cryptanalysis of Knapsack Cryptography*. Lecture 5 from the course "Lattices in Cryptography" at Georgia Tech. 2013. URL: https://web.eecs.umich.edu/~cpeikert/lic13/lec05.pdf.

[PPR09] Anna Pagh, Rasmus Pagh, and Milan Ruzic. "Linear Probing with Constant Independence". In: *SIAM J. Comput.* 39.3 (2009). Preliminary version in STOC, 2007, pp. 1107–1120.

[PT16] Mihai Patrascu and Mikkel Thorup. "On the k-Independence Required by Linear Probing and Minwise Independence". In: *ACM Trans. Algorithms* 12.1 (2016). Preliminary version in ICALP, 2010, 8:1–8:27.

[Qua] Kent Quanrud. *Randomized Algorithms*. URL: http://fundamentalalgorithms.com/randomized.

[Ram30] Frank Plumpton Ramsey. "On a Problem of Formal Logic". In: *Proceedings of the London Mathematical Society* s2-30.1 (Jan. 1930), pp. 264–286.

[Roy59] Bernard Roy. "Transitivité et connexité". In: *C. R. Acad. Sci. Paris (in French)* 249 (1959), pp. 216–218.

[RSL+91] A. H. G. Rinnooy Kan, A. Schrijver, J. K. Lenstra, and International Symposium on Mathematical Programming. *History of mathematical programming : a collection of personal reminiscences / edited by Jan Karel Lenstra, Alexander H.G. Rinnooy Kan, Alexander Schrijver*. English. CWI ; North-Holland ; Distributors for the U.S. and Canada, Elsevier Science Pub. Co Amsterdam : Amsterdam ; New York : New York, N.Y., U.S.A, 1991, ix, 141 p. :

[RW13] Liam Roditty and Virginia Vassilevska Williams. "Fast approximation algorithms for the diameter and radius of sparse graphs". In: *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*. ACM, 2013, pp. 515–524.

[Sch] Alexander Schrijver. "A Course in Combinatorial Optimization". Available at https://homepages.cwi.nl/~lex/files/dict.pdf.

[Sch03] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Vol. 24. Algorithms and Combinatorics. Springer, 2003.

[Shi55] Alfonso Shimbel. "Structure in communication nets". In: *Proceedings of the Symposium on Information Networks (New York, 1954)*. Brooklyn, New York: Polytechnic Press of the Polytechnic Institute of Brooklyn, 1955, pp. 199–203.

[Sip20a] Michael Sipser. *Cook-Levin Theorem*. Lecture 16 of the course MIT 18.404J Theory of Computation, Fall 2020. 2020. URL: https://youtu.be/6Az1gtDRaAU.

[Sip20b] Michael Sipser. *NP-Completeness*. Lecture 15 of the course MIT 18.404J Theory of Computation, Fall 2020. 2020. URL: https://youtu.be/iZPzBHGDsWI.

[Sip97] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997. Accompanying video lectures available at https://youtube.com/playlist?list=PLUl4u3cNGP60_JNv2MmK3wkOt9syvfQWY.

[SS89] Jeanette P. Schmidt and Alan Siegel. "On Aspects of Universality and Performance for Closed Hashing (Extended Abstract)". In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*. ACM, 1989, pp. 355–366.

[SS90] Jeanette P. Schmidt and Alan Siegel. "The Analysis of Closed Hashing under Limited Randomness (Extended Abstract)". In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*. ACM, 1990, pp. 224–234.

[ST85a] Daniel Dominic Sleator and Robert Endre Tarjan. "Amortized Efficiency of List Update and Paging Rules". In: *Commun. ACM* 28.2 (1985), pp. 202–208. URL: https://doi.org/10.1145/2786.2793.

[ST85b] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-Adjusting Binary Search Trees". In: *J. ACM* 32.3 (1985), pp. 652–686. Preliminary version in STOC, 1983.

[Str69] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (1969), pp. 354–356. URL: https://link.springer.com/content/pdf/10.1007%2FBF02165411.pdf.

[Tar13] Robert E. Tarjan. *Analysis of the Highest-Label Push-Relabel Algorithm*. Note for course COS528 at Princeton University. Spring 2013.

[Tar72] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160. Preliminary version in SWAT, 1971.

[Tar75] Robert Endre Tarjan. "Efficiency of a Good But Not Linear Set Union Algorithm". In: *J. ACM* 22.2 (1975), pp. 215–225. URL: https://doi.org/10.1145/321879.321884.

[Tar85] Robert Endre Tarjan. "Amortized Computational Complexity". In: *SIAM Journal on Algebraic Discrete Methods* 6.2 (1985), pp. 306–318. URL: https://courses.cs.duke.edu/fall11/cps234/reading/Tarjan85_AmortizedComplexity.pdf.

[Tho15a] Mikkel Thorup. "High Speed Hashing for Integers and Strings". In: *CoRR* abs/1504.06804 (2015). arXiv: 1504.06804. URL: http://arxiv.org/abs/1504.06804.

[Tho15b] Mikkel Thorup. "Linear Probing with 5-Independent Hashing". In: *CoRR* abs/1509.04549 (2015). arXiv: 1509.04549. URL: http://arxiv.org/abs/1509.04549.

[Tho94] Carsten Thomassen. "Every Planar Graph Is 5-Choosable". In: *J. Comb. Theory, Ser. B* 62.1 (1994), pp. 180–181. URL: https://doi.org/10.1006/jctb.1994.1062.

[Tun94] Levent Tunçel. "On the Complexity of Preflow-Push Algorithms for Maximum-Flow Problems". In: *Algorithmica* 11.4 (1994), pp. 353–359.

[Tur37] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. URL: https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf.

[Vad12] Salil P. Vadhan. "Pseudorandomness". In: *Foundations and Trends in Theoretical Computer Science* 7.1-3 (2012), pp. 1–336. URL: https://people.seas.harvard.edu/~salil/pseudorandomness/pseudorandomness-published-Dec12.pdf.

[Vaz01] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001. URL: http://www.springer.com/computer/theoretical+computer+science/book/978-3-540-65367-7.

[Von] Vonfrisch. *Erdos generated network-p0.01.jpg*. URL: https://commons.wikimedia.org/wiki/File:Erdos_generated_network-p0.01.jpg.

[War62] Stephen Warshall. "A Theorem on Boolean Matrices". In: *J. ACM* 9.1 (Jan. 1962), pp. 11–12.

[Wig19] Avi Wigderson. *Mathematics and Computation: A Theory Revolutionizing Technology and Science*. Princeton University Press, 2019. URL: https://www.math.ias.edu/files/Book-online-Aug0619.pdf#page=1.

[Wik] Wikipedia contributors. *Binomial distribution — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Binomial_distribution.

[Wil05] Ryan Williams. "A new algorithm for optimal 2-constraint satisfaction and its implications". In: *Theor. Comput. Sci.* 348.2-3 (2005), pp. 357–365. URL: https://people.csail.mit.edu/rrw/2-csp-final.pdf. Preliminary version in ICALP, 2004.

[Wil19] David P. Williamson. *Network Flow Algorithms*. Cambridge University Press, 2019. URL: http://www.networkflowalgs.com.

[WS11] David P. Williamson and David B. Shmoys. *The design of approximation algorithms*. Cambridge University Press, 2011. URL: https://www.designofapproxalgs.com.

[WW20] Virginia Vassilevska Williams and Ryan Williams. *Fine-Grained Algorithms and Complexity*. Course 6.S078 at MIT. Fall 2020. URL: https://people.csail.mit.edu/virgi/6.s078/.

Part II

Course materials

Appendix E

Homework assignments



Homework 0

- Due 11:59PM on January 22.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.

1. Exercise 1.3.[1]
2. Exercise 2.1 problems 2–6 and 8.[1]
3. Exercise 2.2 problems 3–7.
4. (Just for fun) Exercises 1.7, 1.8, 2.6 and 2.10.

[1]There is a high bar for "fun and interesting" on the extra credit problems.

Homework 1

- Due 11:59PM on January 29.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.
- Unlike last week, we are also expecting proofs of correctness for these problems (which is the default).
- If you work in a group, please submit it as a group in Gradescope. This cuts down the grading load.
- Also, please follow the formatting instructions on the MCQ, because Gradescope's pattern matching is limited.

1. Exercise 3.3.
2. Exercise 3.7
3. Exercise 4.5.

Homework 2

- Due 11:59PM on February 5.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.
- Exercises 5.1 and 5.2 do not require proofs of correctness (as stated explicitly in their instructions), but the other exercises do (which is generally the default). For these exercises, your proofs should probably include the word "induction".
- If you work in a group, please submit it as a group in Gradescope. This cuts down the grading load.
- Also, please follow the formatting instructions on the MCQ, because Gradescope's pattern matching is limited.

1. Exercise 5.1 problems 2, 4, 6, 7 and 10.
2. Exercise 5.2 problems 1–3.
3. Exercise 5.13.
4. (Optional challenge / for fun) Exercise 5.9.

Homework 3

- Due 11:59PM on February 12.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.
- Friendly reminder: reductions require a proof of correctness that typically involve an "if and only if (iff)".

1. Exercise 6.5.
2. Exercise 7.1 problems 3 and 5–8.
3. Exercise 7.8.
4. (Optional challenge / for fun) Exercise 23 (Kaniel the Dane's puzzle) in Chapter 5 of [Eri19].

Homework 4

- Due 11:59PM on February 26.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.
- Friendly reminder: reductions require a proof of correctness that typically involve an "if and only if (iff)".

1. Exercise 8.1.
2. Exercise 9.9.
3. Exercise 10.6

Homework 5

- Due 11:59PM on March 5.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.

1. Exercise 11.5.
2. Exercise 12.10.
3. Exercise F.1.
4. (Optional challenge, just for fun) Exercises 11.8 and 11.11.

Homework 6

- Due 11:59PM on March 12.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.

1. Exercise 12.22
2. Exercise 13.4
3. Exercise 13.11
4. (Optional challenge, just for fun) Watch Good Will Hunting.

Homework 7

- Due 11:59PM on March 26.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.

1. Exercise 16.6
2. Exercise 16.10
3. Exercise 17.3
4. (Optional challenge, just for fun) Have a nice spring break!

Homework 8

- Due 11:59PM on April 9.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.

1. Exercise 18.4
2. Exercise 19.2
3. Exercise 19.3

Homework 9

- Due 11:59PM on April 16.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.

1. Exercise 22.3
2. Exercise 25.12
3. Exercise 25.14

Homework 10

- Due 11:59PM on April 23.
- Please be aware of Section I.5 (in the syllabus) regarding homework policies.
- Many problems can be solved by "identifying the right subproblems" or "changing the input", which may be counterintuitive.
- Before submitting, we encourage you to ask yourself, *Is this really the simplest way to solve the problem? Is this really the clearest way to explain the solution?*
- There are also multiple choice questions on Gradescope.

1. Exercise 23.1
2. Exercise 27.5
3. Exercise 27.8

Appendix F

Selected solutions

F.1 Homework 0

Exercise 1.3. Problems 1–5 describe a search problem on a nonempty array $A[1..n]$ of comparable elements.[1] For each of these problems, design a recursive algorithm that solves the search problem as fast as possible.[2] Briefly analyze the running time of each algorithm.[3]

1. Let $A[1..n]$ be in a *rotated* sorted order. That is, there exists an index $i \in [n]$, called the *rotation index*, such that the concatenation of $A[i..n]$ and $A[1..i-1]$ is sorted in increasing order. (One can think of $A[1..n]$ as being sorted initially, and then rotated cyclically to the right by $i-1$ slots). The goal is to compute the unknown rotation index i. For simplicity, you may assume all the elements are distinct.

2. We say that $A[1..n]$ is a *mountain* if there exists an index $i \in [n]$, called the *peak*, such that $A[1..i]$ is sorted in increasing order and $A[i+1..n]$ is sorted in decreasing order. Given a mountain $A[1..n]$, find the peak. For simplicity, you may assume all elements are distinct.

3. We say an index $i \in [n]$ is a *local minimum* if either (a) $i = n = 1$, (b) $i = 1$ and $A[1] \leq A[2]$; (c) $i = n$ and $A[n] \leq A[n-1]$; or (d) $1 < i < n$, $A[i-1] \geq A[i]$ and $A[i] \leq A[i+1]$. The goal is to find a local minimum from an array $A[1..n]$.

4. Let $A[1..n]$ be an array of integers such that $A[1] = 0$ and $A[n]$ is odd. Call a consecutive pair of indices $(i, i+1)$ a *gap*. We say that a gap is *even* if the difference $A[i+1] - A[i]$ is even and *odd* if the difference is odd. Observe that when $A[1] = 0$ and $A[n]$ is odd, there must be an odd gap. Given an array of integers $A[1..n]$ such that $A[1] = 0$ and $A[n]$ is odd, find an odd gap.

5. Let $A[1..n]$ be an array of integers such that $A[1] = 1$ and $A[n] = n$. For $i \in \{1, \ldots, n-1\}$, we say that there is a *jump* at index i if $A[i+1] \geq A[i] + 1$. Observe that since

$$\sum_{i=1}^{n-1} (A[i+1] - A[i]) = A[n] - A[1] \geq n - 1,$$

[1] Given two elements x and y, you can query if $x < y$, $x > y$, or $x = y$.

[2] This includes a recursive spec, and pseudocode implementing the recursive spec.

[3] For the sake of brevity we do not ask for an explicit analysis of correctness. We will treat your recursive spec as the induction hypothesis of a proof by induction, and fill in the rest.

there must be a jump at least one index i. (If there was no jump, then the sum on the left would less than $n-1$, a contradiction.)

Design and analyze an algorithm that, given an array $A[1..n]$ such that $A[1] = 1$, $A[n] = n$, and $n \geq 2$, finds an index i for which there is a jump.

6. Prove that each of the problems above have a lower bound of $\Omega(\log n)$ queries in the comparison model.

7. (Extra credit / just for fun) Invent your own fun and interesting search problem on an array of comparable elements.

Solution to exercise 1.3.

Problem 1.

Recursive spec.

`rotation-index(`$A[1..n]$`)` Given a rotated sorted array $A[1..n]$, returns an index i such that $A[i..n]A[1..i-1]$ forms a sorted array.

Recursive implementation.

`rotation-index(`$A[1..n]$`)`

1. If $n \leq 1$ then return n.
2. If $A[1] < A[n]$ then return 1.
3. Let $m = \lceil n/2 \rceil$.
4. If $A[1] < A[m]$:
   `/*` *Then $A[n] < A[1] < A[m]$, so $A[1..m]$ is in sorted order, so the rotation index is in $A[m+1..n]$.* `*/`
   A. Return
      1. $m +$ `rotation-index(`$A[m+1..n]$`)`. *// Don't forget the $+m$!*
5. Else:
   `/*` *$A[m] < A[1] < A[n]$, hence $A[1..m]$ is also a rotated sorted array containing the rotation index.* `*/`
   A. Return `rotation-index(`$A[1..m]$`)`.

Running time analysis. Each recursive call takes $O(1)$ time excluding work delegated to recursive subcalls. Each recursive decreases the size of the input by half. Thus the recursive depth is $O(\log(n))$, and the running time is $O(\log n)$.

Problem 2.

Recursive spec.

`peak(`$A[1..n]$`)` Given a mountain $A[1..n]$, where $n \geq 1$, returns a peak index $i \in [n]$.

Recursive implementation.

`peak`$(A[1..n])$

1. If $n = 1$ then return 1.
2. If $n = 2$:
   A. If $A[1] \leq A[2]$ then return 2.
   B. Otherwise return 1.
3. Let $m = \lceil n/2 \rceil$.
4. If $A[m-1] \leq A[m]$ and $A[m+1] < A[m]$ then return m.
5. If $A[m-1] > A[m]$:
   /* *Since $A[m-1] > A[m]$, any peak of $A[1..m-1]$ is a peak of $A[1..n]$.* */
   A. Return `peak`$(A[1..m-1])$.

/* *At this point, $A[m+1] > A[m]$. Consequently any peak of $A[m+1..n]$ is a peak of $A[1..n]$.* */

6. Return $m +$ `peak`$(A[m+1..n])$. // *Don't forget the $+m$!*

Running time analysis. Each recursive call takes $O(1)$ time excluding work delegated to recursive subcalls. Each recursive decreases the size of the input by half. Thus the recursive depth is $O(\log(n))$, and the running time is $O(\log n)$.

Problem 3.

Recursive spec.

`local-min`$(A[1..n])$ Given an array $A[1..n]$, where $n \geq 1$, returns the index $i \in [n]$ of a local minimum.

Recursive implementation.

`local-min`$(A[1..n])$

1. If $n = 1$ then return 1.
2. If $A[1] \leq A[2]$ then return 1.
3. If $A[n] \leq A[n-1]$ then return n.

/* *At this point we must have $n \geq 3$.* */

4. Let $m = \lceil n/2 \rceil$.
5. If $A[m-1] \geq A[m]$ and $A[m] \leq A[m+1]$ then return m.
6. If $A[m-1] \leq A[m]$:
   /* *Since $A[m-1] \leq A[m]$, any local minimum of of $A[1..m-1]$ is a local minimum of $A[1..n]$.* */
   A. Return `local-min`$(A[1..m-1])$.

/* *At this point, $A[m+1] \leq A[m]$. Therefore any local minimum of $A[m+1..n]$ corresponds to a local minimum of $A[1..n]$.* */

7. Return $m +$ `local-min`$(A[m+1..n])$.

Running time analysis. Each recursive call takes $O(1)$ time excluding work delegated to recursive subcalls. Each recursive decreases the size of the input by half. Thus the recursive depth is $O(\log(n))$, and the running time is $O(\log n)$.

Problem 4.

Recursive Spec.

> `Gap(`$A[1..n]$`)`: Given an array $A[1..n]$ of integers such that $A[n] - A[1]$ is odd, return the odd gap $(h, h+1)$.

Recursive implementation.

> `Gap(`$A[1..n]$`)`
>
> 1. If $n = 2$:
>   A. Return $(1, 2)$.
> 2. Let $h = \lceil \frac{n+1}{2} \rceil$.
> 3. If $A[n] - A[h]$ is odd:
>   A. Return `Gap(`$A[h..n]$`)`.
>
> /* *Otherwise $A[h] - A[i]$ is odd.* */
>
> 4. Return `Gap(`$A[1..h]$`)`.

The running time is modeled by

$$T(2) = O(1),$$
$$T(n) = T(\lceil n/2 \rceil) + 1 \text{ for } n > 2.$$

This is the same as binary search, so the running time is $O(\log n)$.

Remark F.1. It's critical to specify that $A[n] - A[1]$ is odd.

Problem 5.

> `FindJump(`$A[1..n]$`)`: Given an array $A[1..n]$ such that $n \geq 2$ and $A[n] - A[1] \geq n - 1$, returns an index i such that $A[i+1] \geq A[i] + 1$.

> `FindJump(`$A[1..n]$`)`
>
> 1. If $n = 2$ return 1.
> 2. Let $i = \lceil n/2 \rceil$.
> 3. If $A[i] - A[1] \geq i - 1$:
>   A. Return `FindJump(`$A[1..i]$`)`.
>
> /* *Otherwise*
> $A[n] - A[i] = A[n] - A[1] - (A[i] - A[1]) \geq n - 1 - (i-1) = n - i = (n - (i-1)) - 1.$ */
>
> 4. Return $i - 1 +$ `FindJump(`$A[i..n]$`)`.

Remark F.2. It is critical to observe that $A[n] - A[1] \geq n - 1$ in the recursive spec, since otherwise there may not be a jump in A.

Rubric. (For Problems 1–5.) Out of 1.5 points:
1 pt. For a correct recursive spec.
- **.25 pts.** Any attempt at a recursive spec.
- **.25 pts.** Stating the input
- **.25 pts.** Stating the desired output.
- **.25 pts.** Explicitly stating conditions on the input that lead to the desired output. (E.g., $A[n] - A[1]$ is odd.)

.5 pts. For implementing the recursive spec.
- **.25 pts.** For base cases. (Check $n = 1, 2, 3$.)
- **.25 pts.** For general case.
  - Check there's no possibility of an infinite loop.
  - Make sure to update the returned index when recursing on the second half.

Problem 6. One can follow the exact same structure as the proof for the sorting lower bound. The two key observations are:

1. With only k queries an algorithm can output at most 2^k indices.
2. The solution may be any one of n possible indices.

So we must have $2^k \geq n$; taking logarithms on both sides gives the desired lower bound.

Rubric. Out of 2.5 points:
.5 pts. For anything inspired by the sorting lower bound.
.5 pts. For recognizing that k queries can only lead to 2^k different outputs.
.5 pts. For recognizing that there are n possible outputs.
1 pt. For setting up the inequality $2^k \geq n$ and completing the calculations from there.

Exercise 2.1. For each of the following variations of the Tower of Hanoi problem, give an algorithm that solves the problem with the minimum number of ring moves. Recursive algorithms should have an explicit recursive spec. We will treat the recursive spec as an induction hypothesis to justify the correctness of the algorithm, and no explicit proof is otherwise required.[4] Non-recursive algorithms require a proof (which should be short).

You may assume as fact that the `Towers-of-Hanoi` algorithm (Fig. 2.1, on page 42) moves n rings from one post to another in the minimum number of steps. Also, your algorithms are allowed to be mutually recursive (i.e., can depend on one another) across subproblems.

[4]You may want to work out a full proof for yourself anyway for extra practice. Additional justification or comments might help the grader's understanding in the event of partial credit.

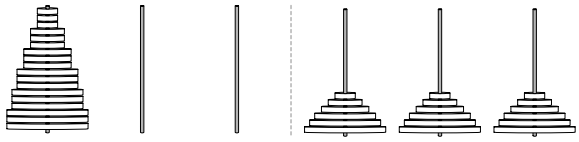
1. In the *easy towers of Hanoi* problem, we remove the restriction that the rings always have to be in sorted order on each post. The goal is still to move n sorted rings from post A to post B, in sorted order on post B.

2. In the *cyclic towers of Hanoi* problem, a ring can only be moved in "cyclic" order from post A to post B, from post B to post C, and from post C to post A. The goal is to move n sorted rings from post A to post B.

3. The *double-cyclic towers of Hanoi* is like the cyclic towers of Hanoi problem except now the goal is to the n sorted rings from post A to post C.

4. In the *thick towers of Hanoi* problem, we have $3n$ rings of n distinct sizes with three copies of each ring. (Rings of the same size can stack on top of each other.) The goal is to move all $3n$ rings from post A to post B.

5. In the *triple towers of Hanoi* problem, we have $3n$ rings of n distinct sizes with three copies of each ring. The goal is to distribute the rings so that each post has n rings, one of each size, in sorted order.



6. In the *American towers of Hanoi* problem, we have n rings with each ring colored red, white or blue. The three posts are also colored red, white, and blue. Initially all the rings are stacked (in order of size) on the red post. The goal is to stack all the red rings on the red post, the blue rings on the blue post, and the white rings on the white post (again, in order of size).

7. In the *messy towers of Hanoi* problem, the n rings are initially distributed arbitrarily over the three posts. (Each post is still in order.) The goal is to move all the rings to a single designated post.[5]

8. (Extra credit / just for fun) Invent a fun and interesting variant of the towers of Hanoi problem.

Solution to exercise 2.1.

Problem 2: cyclic towers of Hanoi.

Recursive spec. `Cyclic-ToH(`n`,`A`,`B`,`C`)`: Given a configuration with the n smallest rings on post A, moves the n smallest ring from post A to post B (while only moving rings in cyclic order), with the minimum number of ring moves.

[5]You are allowed to look at the current configuration and identify which post has the nth smallest ring, for any given n. You may have different cases depending on which post has the nth smallest ring.

Implementation.

`cyclic-ToH`(n, A, B, C)

`/*` *We use* `double-cyclic-ToH`(n, A, B, C) *from problem 3 below.* `*/`

1. If $n > 0$:
   A. `double-cyclic-ToH`$(n-1, A, B, C)$.
   B. Move the top ring from post A to post B.
   C. `double-cyclic-ToH`$(n-1, C, A, B)$.

Remark F.3. One can also implement `cyclic-ToH` replacing each call to `double-cyclic-ToH` with two calls to `cyclic-ToH`; however, this is slower than the approach above.

Problem 3: double-cyclic towers of Hanoi.

Recursive spec. `double-cyclic-ToH`(n, A, B, C): Given a configuration with the n smallest rings on post A, moves the n smallest ring from post A to post C (while only moving rings in cyclic order), with the minimum number of ring moves.

Implementation.

`double-cyclic-ToH`(n, A, B, C)

`/*` *We use* `cyclic-ToH`(n, A, B, C) *from problem 2 above.* `*/`

1. If $n > 0$:
   A. `double-cyclic-ToH`$(n-1, A, B, C)$.
   B. Move the top ring from post A to post B.
   C. `cyclic-ToH`$(n-1, C, A, B)$.
   D. Move the top ring from post B to post C.
   E. `double-cyclic-ToH`$(n-1, A, B, C)$.

Problem 4: Thick towers of Hanoi.

Recursive spec. `thick-ToH`(n, A, B, C): given a configuration $3n$ rings on peg A, with three copies each of n distinct sizes, `thick-ToH`(n, A, B, C) moves all $3n$ rings from peg A to peg B with the minimum number of ring moves.

Implementation.

`thick-ToH`(n, A, B, C)

1. If $n > 0$:
   A. `Thick-Towers-of-Hanoi`$(n-1, A, C, B)$.
   B. Move the top ring from A to B, three times.
   C. `Thick-Towers-of-Hanoi`$(n-1, C, B, A)$.

Problem 5: Triple towers of Hanoi.

Recursive spec. `triple-ToH`(n, A, B, C): Assuming the $3n$ smallest rings are on peg A, with three copies each of n distinct sizes, `triple-ToH`(n, A, B, C) places n rings, one of each size, to each post, with the minimum number of ring moves.

Implementation.

`triple-ToH`$(n-1, A, B, C)$

1. If $n > 0$:
   A. `thick-ToH`$(n-1, A, B, C)$
   B. Move one ring from A to C.
   C. `thick-ToH`$(n-1, B, C, A)$.
   D. Move one ring from A to B.
   E. `triple-ToH`$(n-1, C, A, B)$

Problem 6: American towers of Hanoi.

Recursive spec. `American-ToH`(n, X, A, B, C): Assuming the n smallest rings are on post X (where $X \in \{A, B, C\}$), and where each ring is colored red, white, or blue, moves all the red rings to post A, all the blue rings to post B, and all the white rings to post C, so that the rings on each post are in increasing in size from top to bottom.

Implementation.

`American-ToH`(n, X)

1. If $n > 0$:
   A. Let $Y \in \{A, B, C\}$ be the post with the same color as the nth ring on post X.
   B. If $Y = X$:
      1. `American-ToH`$(n-1, X)$.
   C. Otherwise:
      1. Let $Z \in \{A, B, C\} \setminus \{X, Y\}$ be the third post other than X or Y.
      2. `ToH`$(n-1, X, Z, Y)$.
      3. Move one ring from X to Y.
      4. `American-ToH`$(n-1, Z)$.

Rubric. 2 points per problem.
1 pt. For the recursive spec.
-.25 pt. for omitting "with the minimal number of ring moves."]
1 pt. For the implementation.
-.25 pt. for incorrect base cases.
-.5 pt. for an implementation that correctly moves rings to the desired configuration, but is suboptimal with respect to the number of ring moves.

Exercise 2.2. For each of the recursive specifications below, design a recursive algorithm implementing the specification. (No proof or analysis is needed.)[6]

1. `subset-sum`$(x_1, \ldots, x_n,\ T)$: Given n integers $x_1, \ldots, x_n \in \mathbb{Z}$, and an additional integer T, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = T$, and `false` otherwise.

2. `min-vertex-cover`$(G = (V, E))$: Given an undirected graph $G = (V, E)$, return the minimum size of any vertex cover of G.

   (A vertex cover is a set of vertices $S \subseteq V$ that includes at least one endpoint of every edge $e \in E$.)

3. `max-matching`$(G = (V, E))$: Given an undirected graph $G = (V, E)$, return the maximum size of any matching.

   (A matching is a set of edges $M \subseteq E$ with disjoint endpoints.)

4. `max-independent-set`$(G = (V, E)$: Given an undirected graph $G = (V, E)$, return the maximum size of any independent set of vertices.

   (Given an undirected graph $G = (V, E)$, an *independent set* is a set of vertices $S \subseteq V$ such that no two vertices $u, v \in S$ are connected by an edge.)

5. `longest-increasing-subsequence-including-first`$(x_1, \ldots, x_n)$: Given a sequence of numbers $x_1, \ldots, x_n$, return the length of the longest (strictly) increasing subsequence of $x_1, \ldots, x_n$ over all subsequences that include x_1. (You may assume $n \geq 1$.)

   (A subsequence of $x_1, \ldots, x_n$ is a sequence of the form $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. A sequence of numbers $y_1, \ldots, y_\ell$ is strictly increasing if $y_i < y_{i+1}$ for $i = 1, \ldots, \ell - 1$.)

6. `reachable`$(G = (V, E)$`,`s`,`$t)$: Given a directed graph G and two vertices $s, t \in V$, return `true` if s can reach t in G, and `false` otherwise.

   (s can reach t if either $s = t$ or there is a sequence of (directed) edges of the form $(s, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, t)$. Note that the endpoint of one edge is the initial point of the next edge. This is also called a *(directed) walk* in G from s to t.)

7. `partition`$(x_1, \ldots, x_n)$: Given n integers $x_1, \ldots, x_n \in \mathbb{Z}$, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = \sum_{i \in [n] \setminus S} x_i$, and false otherwise.[7]

Solution to exercise 2.2.

[6] Your algorithms will not be very fast, but their correctness should follow from an induction argument using the recursive spec as the induction hypothesis.

You are welcome to describe your algorithm using high level steps as long as they are clearly easy to implement and would take linear time; e.g., 'let G_1 be the graph obtained by removing such and such vertices and such and such edges." See the sample solutions on page 60.

Some of these problems have well-known efficient algorithms. Efficiency is besides the point; the goal here is to rapidly get some practice thinking recursively.

Lastly, do not change the recursive spec.

[7] This might be the trickiest one. No, you cannot reduce to subset sum. You need to implement *this* recursive spec, recursively.

Problem 1: Subset sum.

`subset-sum`$(x_1, \ldots, x_n,\ T)$

1. If $T = 0$ return `true`.
2. If $n = 0$ return `false`.

`/*` *x_1 is either in the sum or not.* `*/`

3. Return `subset-sum`$(x_2, \ldots, x_n,\ T) \vee$ `subset-sum`$(x_2, \ldots, x_n,\ T - x_1)$.

Problem 2: Vertex cover.

`min-vertex-cover`$(G = (V, E))$

1. If $E = \emptyset$ then return 0.
2. Let $e = \{u, v\} \in E$ be any edge.

`/*` *Any vertex cover must contain either u or v.* `*/`

3. Let $E_u = \{d \in E : u \in d\}$ and $E_v = \{d \in E : v \in d\}$ be the sets of edges incident to u and v, respectively.
4. Let $G_1 = (V - u, E \setminus E_u)$ be the graph obtained by removing u and all edges incident to u.
5. Let $G_2 = (V - v, E \setminus E_v)$ be the graph obtained by removing v and all edges incident to v.
6. Return $1 + \min\{$`min-vertex-cover`$(G_1),$ `min-vertex-cover`$(G_2)\}$.

Problem 3: Maximum matching.

`max-matching`$(G = (V, E))$

1. If $|E| \leq 1$ then return $|E|$.
2. Let $e = \{u, v\}$ be any edge.
3. Let $G_1 = (V, E - e)$ be the graph obtained by removing e.
4. Let $G_2 = (V \setminus \{u, v\}, E_2)$, where $E_2 = \{d \in E : d \subseteq V \setminus \{u, v\}\}$, be the graph obtained by removing the vertices u and v and all incident edges.
5. Return $\max\{$`max-matching`(G_1), `1 +` `max-matching`$(G_2)\}$.

Problem 4: Maximum independent set.

`max-independent-set`$(G = (V, E))$

1. If $|V| \leq 1$ then return $|V|$.
2. Let $v \in V$ be any vertex.
3. Let $G_1 = (V - v, E \setminus \{e \in E : v \in e\})$ be the graph obtained by removing v and all edges incident to v.
4. Let $U = \{u : \{u, v\} \in E\}$ be the set of neighbors in V. Let $G_2 = (V \setminus U, E \setminus \{e \in E : U \cap e \neq \emptyset\})$ be the graph obtained by removing U and all edges incident to U.
5. Return $\max\{$`max-independent-set`$(G_1), 1 +$ `max-independent-set`$(G_2)\}$.

Problem 5: Longest increasing subsequence including the first element. We abbreviate `longest-increasing-subsequence-including-first` as `LISIF`.

`LISIF`$(x_1, \ldots, x_n)$

1. If $n \leq 1$ then return n.
2. $b \leftarrow 1$.
3. For $i = 2, \ldots, n$:
   A. If $x_i > x_1$:
      1. $b \leftarrow \max\{b, 1 + $ `LISIF`$(x_i, \ldots, x_n)\}$.
4. Return b.

Problem 6: Reachability.

`reachable`$(G = (V, E), s, t)$

1. If $s = t$ then return `true`.
2. $E_s = \{(u, s) \in E\} \cup \{(s, v) \in E\}$ be the set of edges incident to s. Let $G' = (V - s, E \setminus E')$ be the graph obtained by removing s and all edges incident to s.
3. For all edges $(s, v) \in E$ leaving s:
   A. If `reachable`(G', v, t) then return `true`.
4. Return `false`.

Problem 7: Partition.

`partition`$(x_1, \ldots, x_n)$

1. If $n = 0$ return `true`.
2. If $x_1 = x_2 + \cdots + x_n$ then return `true`.
3. For $i = 2, \ldots, n$:
   A. If `partition`$(x_2, \ldots, x_1 + x_i, \ldots, x_n)$ is `true`, then return `true`.
4. Return `false`.

Remark F.4. In general, the solutions are not necessarily exhaustive. For example, for problem 7, other approaches include:

- Recursing with the disjunction of `partition`$(x_1 - x_2, x_3, \ldots, x_n)$ and `partition`$(x_1 - x_2, x_3, \ldots, x_n)$.
- Recursing with the disjunction of `partition`$(x_1 + x_2, x_3, \ldots, x_n)$, `partition`$(x_1,\ x_2 + x_3,\ x_1,\ x_4,\ \ldots,\ x_n)$, and `partition`$(x_1 + x_3, x_2,\ x_4,\ \ldots,\ x_n)$.

> *Rubric.* Each problem is worth 2 points:
> **1 point.** Correct base cases.
> **1 point.** Correct induction/recursive step.

F.2 Homework 1

Exercise 3.3. Let P be a set of n points in $\mathbb{R}^2$.[8] Suppose you want to compute a list of all pairs of points with distance with a factor of 2 of the minimum distance between all pairs of points. Design and analyze an algorithm for this problem.

(Food for thought: what does your running time say about the total number of pairs of points with distance within a constant factor of the minimum distance between all pairs of points?)

Solution to exercise 3.3. We first compute the minimum distance δ between any pair of points in P in $O(n \log n)$ time, using the algorithm from Section 3.2. We then sort the points P in decreasing order of y-coordinate, and call the recursive algorithm `all-close-pairs(`P`,`δ`)` described below.

Recursive spec. `all-close-pairs(`P`, `δ`)`: Given a list of n points P sorted in descending order of y-coordinate, and $\delta > 0$, returns a list of all pairs of (distinct) points $\{p, q\} \subseteq P$ where $\|p - q\| \leq 2\delta$.

Imlementation.

`all-close-pairs(`P`, `δ`)`

`/*` *We assume P is sorted by y-coordinate. (Otherwise sort P once in a preprocessing step.)* `*/`

1. If $|P| \leq 1$ then return an empty list.
2. Let a be the median of the x-coordinates. // $O(n)$
3. `/*` *Divide P along the vertical line $x = a$* `*/`
4. Let $P_1 = \{(x_i, y_i) \in P : x < a\}$ and $P_2 = \{(x_i, y_i) \in P : x \geq a\}$ // $O(n)$
5. `/*` *Find a close pairs within each half P_1 and P_2 recursively.* `*/`
6. Let $L_1 =$ `all-close-pairs(`P_1`,`δ`)` and $L_2 =$ `all-close-pairs(`P_2`,`δ`)`. // $2T(n/2)$
7. `/*` *It remains to find all close pairs P_1 and P_2.* `*/`
8. Let $P_3 = \{(x, y) \in P : |x - a| \leq 2\delta\}$ and let $p_1, \ldots, p_k$ list P_3 in decreasing order of y-coordinate. // $O(n)$
9. Let L_3 be an empty list.
10. For each $p_i = (x_i, y_i) \in P_3$:
    A. For $j = i + 1, \ldots, \min\{k, i + 40\}$:
        1. Let $p_j = (x_j, y_j)$. If (a) $\|p_i - p_j\| \leq 2\delta$ and (b) either $x_i < a$ and $x_j \geq a$, or $x_i \geq a$ and $x_j < a$, then append $\{x_i, x_j\}$ to L.
11. Return the concatenation of L_1, L_2, and L_3.

Running time. Letting $T(n)$ denote the running time of `all-close-pairs` on n points, we have

$$T(n) = 2T(n/2) + O(n)$$

and $T(1) = T(0) = O(1)$. This is the same recurrence as `merge-sort` and is solved by $T(n) = O(n \log n)$.

[8] For simplicity you may assume the points are in general position; that is, no two points have the same x-coordinate or the same y-coordinate.

Proof of correctness. To prove the algorithm lemma, we will invoke the following lemma which is similar to Lemma 3.3. We omit the proof as it is the same as for Lemma 3.3.

Lemma F.5. *Let δ be the minimum distance between any pair of points. The in an $\delta \times \delta$ square, there are at most 5 points.*

Now we prove that `all-close-pairs` implements the recursive spec by induction on $|P|$.

In the base case, $|P| < 2$, and there are no pairs of points. `all-close-pairs` returns the empty list.

Now suppose $|P| > 2$. The algorithm divides P into two sets P_1 and P_2 by the median x-coordinate. We have three types of close pairs $\{p, q\}$:

1. Pairs $\{p, q\}$ where both $p, q \in P_1$.
2. Pairs $\{p, q\}$ where both $p, q \in P_2$.
3. Pairs $\{p, q\}$ where both $p \in P_1$ and $q \in P_2$.

By induction on $|P|$, the recursive calls to P_1 and P_2 list all the pairs of the first and second type in L_1 and L_2, respectively.

Consider the pairs of the third type. Any point in such a must have x-coordinate within 2δ of the median a, and is included in P_3. The algorithm iterates through P_3 in descending order of y-coordinate and compares each p_i with the 40 subsequent points in descending order of y-coordinate. We need to show that this suffices.

Let $\{p_i, p_j\}$ be a pair of the third type with $i < j$. We need to show that $j \leq i + 40$. Letting $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$, we know that $|x_j - a| \leq 2\delta$ and $y_j \geq y_i - 4\delta$. Thus p_j must lie in the box B of width 4δ and height 2δ and centered at $(a, y_i - \delta)$. This box B decomposes to 8 square boxes of width and height δ, and by the lemma above, each of these smaller boxes contains at most 5 points. Thus B contains at most 40 points. Since every point p_k with $i \leq k \leq j$ is in B, we have $j - i \leq 40$, as desired.

Thus L_3 does capture all pairs of points of type 3. Altogether L_1, L_2, and L_3 give all pairs of points of distance at most 2δ, as desired.

Rubric.
3 pts. Recursive spec. (Always the most important step.)
3 pts. Recursive implementation
- **-1 pts.** For a slightly slower algorithm with running time $O\left(n \log^{O(1)} n\right)$.
- **-2 pts.** For a slower algorithm with running time a polynomial factor larger than $O(n \log n)$. (e.g., $O\left(n^2\right)$.)

1 pts. Runtime analysis (of the presented algorithm).
3 pts. Proof of correctness (by induction)
- **1 pts.** Packing argument to justify having to only compare a limited number of points in the "merge" step.
- **2 pts.** Proof by induction.
  - **1 pts.** Base case
  - **1 pts.** Induction step
  - **1 out of 2 pts.** For a solution with a clean explanation of the ideas but not a formal proof by induction.

Remark F.6. If you used a hashmap, then you probably got an *expected* running time.

Exercise 3.7. Given a set of numbers $x_1, \ldots, x_n$, the *prefix sums* are the n numbers,

$$x_1,\ x_1 + x_2,\ x_1 + x_2 + x_3,\ \ldots,\ x_1 + \cdots + x_n.$$

Of course the prefix sums can be computed in $O(n)$ time, one prefix sum at a time. What about in *parallel*; e.g., on a GPU?

Here we consider a *parallel* computation model, called *PRAM*. To motivate this model, imagine one had arbitrarily many processors, with shared memory, trying to solve a common problem. Even with tons of computing, there may still be sequential bottlenecks — some steps have to be executed before others — and issues of synchronization.

In the PRAM model, the processors have access to shared memory. To simplify the synchronization, we restrict our model to work in *rounds*. In each round, each processor reads a constant amount of information from memory, does a constant amount of work, and writes to a constant number of locations in memory. We will restrict ourselves to algorithms where in each round, different processors always write to different locations, so we can avoid issues of contention resolution.

The number of rounds needed to complete the task is the *parallel running time*. The total number of operations, over all processes, is called the *total amount of work*. The ideal parallel algorithm has total work similar to the best known sequential algorithm, but has a much faster parallel running time.

For example, here is a first attempt at a parallel divide-and-conquer algorithm for prefix sums. We first define our recursive spec:

> `prefix-sum(`$A[1..n]$`)`: Given an array of numbers $A[1..n]$, returns an array of the prefix sums of A.

`prefix-sum(`$A[1..n]$`)`

`/*` *Given an input array of numbers* $A[1..n]$*, returns an array of the prefix sums of* A*.* `*/`

1. If $n \leq 1$ then return A.
2. Let $k = \lceil n/2 \rceil$.
3. $B[1..k] \leftarrow$ `prefix-sum(`$A[1..k]$`)` and $C[1..n-k] \leftarrow$ `prefix-sum(`$A[k+1..n]$`)` (in parallel).
4. $D[i] \leftarrow B[k] + C[i]$ for $i = 1, \ldots, n-k$. *// $O(n)$ work in $O(1)$ rounds*
5. Return the concatenation of $B[1..k]$ and $D[1..n-k]$.
   // $O(n)$ work in $O(1)$ rounds

Figure F.1: A parallel prefix sum algorithm with $O(\log n)$ time and $O(n \log n)$ work.

Our first implementation divides A into equal-size subarrays and recursively computes the prefix sums on each. Observe that the prefix sums of the second half are missing the sum from the first half. We fix this by taking the last prefix sum of the first half and adding it to each prefix sum in the second half. See Fig. 3.8 for pseudocode.

Let $T(n)$ denote the parallel running time on an input of size n. We have

$$\begin{aligned} T(0) &= T(1) = 1 \\ T(n) &= T(n/2) + O(1) \qquad \text{for } n > 1. \end{aligned}$$

Here we observed that although parallel prefix sum has two recursive calls, they are made in parallel. This is the same recurrence as (sequential) binary search, and implies a $O(\log n)$ parallel running time.

Now we compute the total amount of work. This is the same as the running time if the `parallel-prefix-sum` algorithm was simulated sequentially. Letting $W(n)$ be the total amount of work on an input of size n, we have

$$\begin{aligned} W(0) &= W(1) = 1 \\ W(n) &= 2W(n/2) + n/2 \qquad \text{for } n > 1. \end{aligned}$$

This is the same recurrence as merge sort and other algorithms we've seen, and gives $O(n \log n)$ total work.

This parallel algorithm is very fast, but has a $O(\log n)$-factor more work than the obvious sequential algorithm. The goal of this exercise is to develop a (recursive) parallel divide-and-conquer algorithm that is just as fast — $O(\log n)$ parallel time — but has $O(n)$ total work. In addition to designing the algorithm, you should analyze the running time and the work similar to how we did above. (You are welcome to use the same recursive spec, but don't forget to write it out.)

It may be fun to try to figure this out without any hints. (Maybe you'll come up with a new approach!) But if you want a little help, then there are some guiding questions in the following footnote.[9]

Remark F.7. To generalize beyond summing numbers, observe that the only property of addition we used was the associative property:

$$a + (b + c) = (a + b) + c.$$

So with essentially the same algorithm, one can compute generalized prefix sums for any associative binary operator in $O(\log n)$ rounds and $O(n)$ work.

Solution to exercise 3.7. We use the same recursive spec:

> `prefix-sum(`$A[1..n]$`)`: Given an array of numbers $A[1..n]$, returns an array of the prefix sums of A.

Now we give a new implementation of `prefix-sum`. The high-level idea is to recursively compute the prefix sums

$$A[1] + A[2] + \cdots + A[i]$$

for *only* the even indices i. To do that, we sum the consecutive pairs of A

$$B[1] = A[1] + A[2],\ B[2] = A[3] + A[4],\ B[3] = A[5] + A[6], ...$$

(in parallel) and then recursive compute the prefix sum on these pairwise sums V. This will give us the prefix sums of A for all the even indices i. For odd i, we the ith prefix sum by adding $A[i]$ to the the $(i-1)$th prefix. This can be done in parallel for all i.

The key point is that the pairwise sums decreases the input size by a factor of 2. As we analyze below, this reduces the total amount of work to $O(n)$.

Here is the pseudocode.

[9] Hint: Suppose you had the prefix sums $y_i = x_1 + x_2 + \cdots + x_i$ for all *even* i. Can you get the prefix sums for all *odd* i in $O(1)$ rounds and $O(n)$ work?

And then: how do you get the prefix sums for even i efficiently? Of course you could compute all the prefix sums, and then pick out the even indices i. But that's not going to be help us speed up computing all the prefix sums. Can you get the prefix sums for even i by creating another instance of the prefix sums problem with significantly fewer numbers?

`prefix-sum(`$A[1..n]$`)`

1. If $n \leq 1$ return A.
2. Compute $B[1..\lceil n/2 \rceil]$ where $B[i] = A[2i-1] + A[2i]$. *// $O(1)$ time, $O(n)$ work.*
3. $C[1..\lceil n/2 \rceil] \leftarrow$ `prefix-sum(`B`)`

/ $C[i] = B[1] + \cdots + B[i] = A[1] + A[2] + \cdots + A[2i]$ for each i. */*

4. Compute $D[1..n]$ where *// $O(1)$ time, $O(n)$ work.*

$$D[i] = \begin{cases} A[1] & \text{for } i = 1 \\ B[i/2] & \text{for even } i \\ A[i] + B[\lfloor i/2 \rfloor] & \text{for odd } i \geq 3. \end{cases}$$

5. Return D

Running time and work analysis. To analyze the (parallel) running time, let $T(n)$ be the (parallel) running time (i.e., rounds) on an input of size n. We have

$$\begin{aligned} T(1) &= T(0) = 1 \\ T(n) &= T(n/2) + O(1) && \text{for } n > 1. \end{aligned}$$

This is the same recurrence as binary search, so $T(n) = O(\log n)$.

Now we analyze the total amount of work. Let $S(n)$ be the total amount of work on an input of size n. We have

$$\begin{aligned} T(1) &= T(0) = 1 \\ T(n) &= T(n/2) + O(n) && \text{for } n > 0. \end{aligned}$$

If we draw the recurrence tree, we get a geometric sum of the form

$$T(n) = \sum_i \frac{n}{2^i} = O(n).$$

So this version has $O(n)$ total work.

Proof of correctness. To prove, by induction on n, that given an array of numbers $A[1..n]$, `prefix-sum(`$A[1..n]$`)` returns an array of the prefix sums of A. (Just as the recursive spec claims.)

For $n = 0$ or 1, the prefix sum of A is just A. Indeed this is what the algorithm returns.

Suppose $n > 1$. Let $B[1..\lfloor n/2 \rfloor]$, where $B[i] = A[2i-1] + A[2i]$, and $C =$ `prefix-sum(`B`)`. By induction on n, C is the prefix sums of B. For each i, $C[i] = B[1] + \cdots + B[i] = A[1] + \cdots + A[2i]$ is the $(2i)$th prefix sum. This shows that $D[i]$ is the ith prefix sum of A for all even i. Obviously, $D[1] = A[1]$ is the first prefix sum. For the remaining odd indices $i \geq 3$, $D[i] = D[i-1] + A[i]$ is the ith prefix sum because $D[i-1]$ is the ith prefix sum.

Remark F.8. The more general algorithm is sometimes called "parallel prefix scan" and is a standard primitive in parallel computers. (e.g.: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda.)

Rubric.
1 pt. Recursive spec.
5 pts. Recursive implementation (of the spec).
1 pt. Parallel running time analysis.
1 pt. Total work analysis.
2 pts. Proof of correctness.

Exercise 4.5. Let X and Y be two sets of integers. We define the *unique sums* of X and Y as the set of integers of the form

$$z = x + y$$

where $x \in X$, $y \in Y$, and the choice of $(x, y) \in X \times Y$ is unique.[10] You may assume that all the integers in X are distinct (amongst themselves), and that all the integers in Y are distinct (amongst themselves).[11]

1. (2 pt.) Suppose X and Y each have n integers. Design and analyze a $O\big(n^2 \log(n)\big)$ time algorithm to compute the unique sums of X and Y.

2. (6 pt.) Suppose all the integers in X are Y are between 0 and M for a fixed value $M \in \mathbb{N}$. Design and analyze a $O(M \log M)$ time algorithm to compute the unique sums of X and Y.[12]

3. (2 pt.) Suppose that now we had k sets $X_1, \ldots, X_k$, each consisting of integers between 1 and M. (You may again assume no duplicates within each X_i.) A unique sum of $X_1, \ldots, X_k$ is defined as an integer of the form

   $$x_1 + \cdots + x_k,$$

   where $x_1 \in X_1$, $x_2 \in X_2$, $\ldots$, $x_k \in X_k$, and the choice of $(x_1, \ldots, x_k) \in X_1 \times \cdots \times X_k$ is unique. Design and analyze an algorithm that computes the unique sums of $X_1, \ldots, X_k$ in $O(kM \log(Mk) \log(k))$ time. You may assume for simplicity that k is a power of 2.[13]

Hint: For part 2, try "changing the input" to an algorithm from Section 4.3 (that is not the FFT, at least directly).

Solution to exercise 4.5.

[10] That is, for fixed z, there is only one choice of $x \in X$ and one choice of $y \in Y$ such that $z = x + y$.

[11] For example, for $X = \{0, 1\}$ and $Y = \{0, 1\}$, the unique sums are $\{0, 2\}$. 1 can be obtained by $0 + 1$ or $1 + 0$ so it is not a unique sum.

[12] It might be helpful to work through very simple examples such as $X = Y = \{1\}$ and $X = Y = \{0, 1\}$.

[13] It might help to work through $k = 4$ to build your intuition.

Part 1. We compute all sums in a nested loop $O(n^2)$ time, and then sort them in $O(n^2 \log n)$ time and remove duplicates.

> *Rubric.*
> **1 pt.** For a correct algorithm.
> **1 pt.** For running time analysis.

Part 2. We will use the $O(n \log n)$-time polynomial multiplication algorithm given in Corollary 4.3. We define two polynomials

$$p(x) = a_0 + a_1 x + \cdots + a_M x^M$$
$$q(x) = b_0 + b_1 x + \cdots + b_M x^M$$

where for $i = 0, \ldots, M$ we have:

1. $a_i = 1$ if $i \in X$, and $a_i = 0$ if $i \notin X$.
2. $b_i = 1$ if $i \in Y$, and b_i if $i \notin Y$.

Consider the polynomial $r(x) = p(x)q(x)$ with coefficients

$$r(x) = c_0 + c_1 x + \cdots + c_{2M} x^{2M}.$$

Recall that for each k, we have

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

The RHS gives the number of indices $i \in X$ such that $k - i \in Y$. In particular, k is a unique sum iff $c_k = 1$.

Overall, we apply Corollary 4.3 to compute $r(x) = p(x)q(x)$ in $O(n \log n)$ time, and then return the set of integers k with $c_k = 1$.

> *Rubric.*
> **2 pts.** For anything making the connection (perhaps implicitly) between the polynomial multiplication and addition (via the exponents of the polynomial).
> **2 pts.** Implementation.
> **2 pts.** Proof of correctness.

Part 3. We use divide-and-conquer over the sets $X_1, \dots, X_k$ to create a polynomial representing all possible sums with their multiplicities in the claimed time. We then return the set of integers with coefficient 1, as in part 2.

To compute the polynomial, we define

> `polynomial-of-sums`$(X_1, \dots, X_k)$ to be the polynomial $p(x) = a_0 + a_1 x + \cdots + a_M x^M$ where for each i, a_i is the number of ways to represent i as a sum with exactly one term from each X_i.

Here is an implementation.

`polynomial-of-sums`$(X_1, \dots, X_k)$ $(X_1, \dots, X_k)$

1. If $k = 1$ then return the polynomial $p(x) = a_0 + a_1 x + \cdots + a_{Mk} x^{Mk}$ where $a_i = 1$ if $i \in X_1$, and $a_i = 0$ otherwise.

/* *Otherwise $k \geq 2$.* */

2. Let $p(x) =$ `polynomial-of-sums`$(X_1, \dots, X_{\lceil k/2 \rceil})$
3. Let $q(x) =$ `polynomial-of-sums`$(X_{\lceil k/2 \rceil + 1}, \dots, X_k)$
4. Compute $r(x) = p(x)q(x)$ by Corollary 4.3. // *$O(n \log n)$ time.*
5. Return $r(x)$

The running time obeys the recurrence

$$T(k) = 2T(k/2) + O(Mk \log(Mk))$$

which is solved by $T(k) = O(Mk \log(Mk) \log(k))$.

Proof of correctness. We prove that `polynomial-of-sums`$(X_1, \dots, X_k)$ satisfies the recursive spec by induction on k. The base case for $k = 1$ is immediate. Suppose $k > 1$ and consider the code. Let $\ell_1 = \lceil k/2 \rceil$ and $\ell_2 = k - \ell_1$

Let $p(x) = a_0 + a_1 x + \cdots + a_{\ell_1 M} x^{\ell_1 M}$ and $q(x) = b_0 + b_1 x + \cdots + b_{\ell_2 M} x^{\ell_2 M}$. By induction on k, for all i, a_i is the number of ways to represent i as a sum with exactly one term from each of $X_1, \dots, X_{\ell_1}$. Likewise, by induction on k, for each i, b_i is the number of ways to represent i as a sum with exactly one term from each of $X_{\ell_1 + 1}, \dots, X_k$.

Now, let $r(x) = p(x)q(x) = c_0 + c_1 x + \cdots + c_{kM} x^{kM}$. For each i, consider

$$c_i = \sum_{j=0}^{i} a_j b_{j-i}.$$

Each summand $a_j b_{j-i}$ is the number of ways to represent i as a sum with exactly one term from each X_h, such that the terms from $X_1, \dots, X_{\ell_1}$ sum to a_j, and the terms from $X_{\ell_1 + 1}, \dots, X_k$ sum up to b_{j-i}. The summation over j enumerates over all possible ways to split i into the sum of two nonnegative integers j and $j - i$. The c_i is exactly the number of ways to represent i as a sum with exactly one term from each X_hm as desired.

Rubric.
1 pt. 1 Divide-and-conquer algorithm.
    .5 pts. Correct algorithm.
    .5 pts. Algorithm that runs in the desired time.
.5 pts. Running time analysis (of the described algorithm).
.5 pts. Proof of correctness.

F.3 Homework 2

Exercise 5.1. Each of the following problems takes as input an array $A[1..n]$ of integers and asks for an optimal subsequence.[14] Design and analyze an algorithm for each, addressing Items 1–5 from Section 5.2.[15]

1. A sequence of numbers $x_1, \ldots, x_k$ is (strictly) increasing if $x_i < x_{i+1}$ for $i = 1, \ldots, k-1$. Compute the length of the longest increasing subsequence of A.

2. A sequence of numbers $x_1, \ldots, x_k$ is *convex* if $x_{i+1} - x_i \geq x_i - x_{i-1}$ for $i = 2, \ldots, k-1$. Compute the length of the longest convex subsequence of A.

3. Compute the maximum sum of any increasing subsequence of A.

4. Compute both:

   - the length of the longest increasing subsequence of A where the sum of integers is even,
   - the length of the longest increasing subsequence where the sum of integers is odd.

5. Recall that the Fibonacci sequence

   $$1, 1, 2, 3, 5, 8, \ldots$$

   can be characterized as a sequence starting with 1, 1 and such that every successive number is the sum of the previous two numbers. ($1 + 1 = 2$, $1 + 2 = 3$, $2 + 3 = 5$, ...) Generalizing this notion, we say that a sequence $y_1, y_2, \ldots, y_k$ is *sub-Fibonacci* if $y_i \leq y_{i-2} + y_{i-1}$ for $i = 3, 4, \ldots, k$. That is, each successive number is at most the sum of the previous two.

   Compute the length of the longest increasing and sub-Fibonacci subsequence of A.

6. Suppose each entry in A is also colored red, white, or blue. We say that a sequence is *American* if the colors alternate red, white, blue, red, white, blue, ... The first number in the sequence can be any color. Compute the length of the longest increasing American subsequence of A. (You can assume that you can look up the color of $A[i]$, for any $i \in [n]$, in constant time.)

7. A sequence $x_1, \ldots, x_k$ is a *palindrome* if the reversed sequence is the same; i.e., $(x_1, \ldots, x_k) = (x_k, \ldots, x_1)$. For example,

   mom, *dad*, *racecar*, and *gohangasalamiimalasagnahog*

   are all palindromes. Compute the length of the longest palindrome subsequence of A.

[14] A "subsequence" of $x_1, \ldots, x_n$ is a sequence of the form $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

[15] In particular, no proof is required. We will interpret your recursive spec as the inductive hypothesis of a proof by induction, and fill in the rest of the details ourselves. That said, some additional commentary may help the grader, especially in the event of partial credit.

8. Compute the length of the shortest supersequence of A that is a palindrome.[16]

9. Suppose each entry of A is colored red, white or blue. We say that a sequence is *un-American* if the colors in the sequence never cycle through the American dream; that is, no three consecutive entries have colors that form any of the following three sequences: (red, white, blue), (white, blue, red), or (blue, red, white). Compute the length of the longest increasing un-American subsequence of A. (You can assume that you can look up the color of $A[i]$, for any $i \in [n]$, in constant time.)

10. (Just for fun) Invent your own fun and interesting version of a dynamic programming problem over A.

Solution to exercise 5.1.

Problem 2: Longest convex subsequence.

1. Recursive spec / induction hypothesis. Fix $A[1..n]$. For $i, j \in \{1, \dots, n\}$ with $i < j$, we define

> $\texttt{LCS}(i,j)$ = the length of the longest convex subsequence of $A[i..n]$ where the first two numbers are $A[i]$ and $A[j]$.

2. Recursive implementation.

> $\underline{\texttt{LCS}(i,j)}$
>
> 1. If $A[k] - A[j] < A[j] - A[i]$ for all $k \in \{j+1, \dots, n\}$ then return 2.
> 2. Otherwise return the maximum of $\texttt{LCS}(j,k)$ over all indices $k \in \{j+1, \dots, n\}$ s.t. $A[k] - A[j] \geq A[j] - A[i]$.

3. Running time with caching / dynamic programming. We have $O(n^2)$ subproblems, each of which takes $O(n)$ time. With dynamic programming, we only solve each subproblem once, giving $O(n^3)$ time overall.

4. Solving the original problem. Given A, if $n \leq 1$, then we return 1. If $n = 2$ then return 2. Otherwise we return the maximum of $1 + \texttt{LCS}(i,j)$ over all indices $i, j \in [n]$ with $i < j$.

> *Rubric.*
> **.75 pts.** Recursive spec(s).
> **.75 pts.** Recursive implementation (of the spec).
> **.25 pts.** Base cases.
> **.5 pts.** Inductive step.
> **.25 pts.** Mention dynamic programming.
> **.5 pts.** Running time.
> **.25 pts.** How to solve the original problem.

[16] $x_1, \dots, x_k$ is a *supersequence* of $y_1, \dots, y_\ell$ if $y_1, \dots, y_\ell$ is a subsequence of $x_1, \dots, x_k$.

Problem 4: Longest even-sum and odd-sum increasing subsequence.

1. Recursive spec / induction hypothesis. For $i \in \{1, \dots, n\}$, we define

1. `EvenLIS(`i`)` to be the length of the longest increasing subsequence among those starting with $A[i]$ where the sum is even, (which is 0 if no such subsequence exists).
2. `OddLIS(`i`)` to be the length of the longest increasing subsequence among those starting with $A[i]$ where the sum is odd, (which is 0 if no such subsequence exists).

2. Recursive implementation.

`EvenLIS(`i`)`

1. If $A[i]$ is even:
   A. Return the maximum of 1 or the maximum of $1 +$ `EvenLIS(`j`)` over all $j > i$ with $A[j] > A[i]$.
2. If $A[i]$ is odd:
   A. Return the maximum of 0 or the maximum of $1 +$ `OddLIS(`j`)` over all $j > i$ with $A[j] > A[i]$.

`OddLIS(`i`)`

1. If $A[i]$ is odd:
   A. Return the maximum of 1 or the maximum of $1 +$ `EvenLIS(`j`)` over all $j > i$ with $A[j] > A[i]$.
2. If $A[i]$ is even:
   A. Return the maximum of 0 or the maximum of $1 +$ `OddLIS(`j`)` over all $j > i$ with $A[j] > A[i]$.

3. Running time with caching / dynamic programming. We have $O(n)$ subproblems, each of which takes $O(n)$ time. With dynamic programming, we only solve each subproblem once, giving $O(n^2)$ time overall.

4. Solving the original problem. The length of the longest increasing subsequence with an even sum is obtained by the maximum of `EvenLIS(`i`)` over all $i \in [n]$. The length of the longest increasing subsequence with an odd sum is obtained by the maximum of `OddLIS(`i`)` over all $i \in [n]$.

Rubric.
.75 pts. Recursive spec(s).
.75 pts. Recursive implementation (of the spec).
  .25 pts. Base cases.
  .5 pts. Inductive step.
.25 pts. Mention dynamic programming.
.5 pts. Running time.
.25 pts. How to solve the original problem.

Problem 6: Longest American increasing subsequence.

1. Recursive spec / induction hypothesis. For $i \in \{1, \dots, n\}$, we define

> $\texttt{LAIS}(i)$ = the length of the longest American increasing subsequence of $A[i..n]$ where the subsequence starts with $A[i]$.

2. Recursive implementation.

> $\underline{\texttt{LAIS}(i)}$
>
> 1. Let $X \in \{\texttt{red}, \texttt{white}, \texttt{blue}\}$ be the color that comes after the color of $A[i]$ in the American order.
> 2. If $A[i] \geq A[j]$ for all $j > i$ such that $A[j]$ has color X, then return 1.
> 3. Otherwise return the maximum of
>
> $$1 + \texttt{LAIS}(j)$$
>
> over all indices $j > i$ where $A[j] > A[i]$ and the color of $A[j]$ is X.

3. Solving the original problem. The maximum of $\texttt{LAIS}(i)$ over all $i \in [n]$ gives the length of the longest increasing American subsequence in A.

4. Running time with caching / dynamic programming. With dynamic programming, the running time is $O(n^2)$, where we have n subproblems each of which take $O(n)$ time (to iterate over $j > i$, in steps (2) and (3)).

> *Rubric.*
> **.75 pts.** Recursive spec(s).
> **.75 pts.** Recursive implementation (of the spec).
> **.25 pts.** Base cases.
> **.5 pts.** Inductive step.
> **.25 pts.** Mention dynamic programming.
> **.5 pts.** Running time.
> **.25 pts.** How to solve the original problem.

Problem 7: Longest palindrome subsequence.

1. Recursive spec / induction hypothesis. We define

> $\texttt{LPS}(i,j)$ = the length of the longest palindrome that is a subsequence of $A[i..j]$.

2. Recursive implementation.

`LPS(`i`,`j`)`

1. If $i > j$ then return 0.
2. If $i = j$ then return 1.

/* *Otherwise* $i < j$. */

3. If $A[i] = A[j]$ then return the maximum of $2 +$ `LPS(`$i+1$`,`$j-1$`)`, `LPS(`i`,`$j-1$`)`, and `LPS(`$i+1$`,`j`)`.
4. Otherwise return the maximum of `LPS(`$i+1$`,`j`)` and `LPS(`i`,`$j-1$`)`.

3. Solving the original problem. The original problem is solved by `LPS(1,`n`)`.

4. Running time with caching / dynamic programming. If we apply dynamic programming to save all solutions, the overall running time is $O(n^2)$, broken down as $O(n^2)$ subproblems taking $O(1)$ time each.

Rubric.
.75 pts. Recursive spec(s).
.75 pts. Recursive implementation (of the spec).
  .25 pts. Base cases.
  .5 pts. Inductive step.
.25 pts. Mention dynamic programming.
.5 pts. Running time.
.25 pts. How to solve the original problem.

Exercise 5.2. Let $A[1..m]$ and $B[1..n]$ be two arrays. A common subsequence of A and B is a sequence that is a subsequence of both. Design and analyze an algorithm for each of the following problems, addressing Items 1–5 from Section 5.2.[16]

1. Compute the length of the longest common subsequence of A and B.
2. Compute the length of the shortest common supersequence of A and B.
3. Compute the length of the longest common subsequence of A and B that is a palindrome.
4. Suppose A and B consist of integers. Compute the length of the longest common increasing subsequence of A and B.
5. Compute the length of the shortest common palindrome supersequence of A and B.
6. Suppose A and B consist of integers. Compute the length of the longest common convex subsequence of A and B. (Cf. problem 2)

Solution to exercise 5.2.

Problem 1: longest common subsequence.

1. Recursive spec / induction hypothesis. $\texttt{LCS}(i,j)$= the length of the longest common subsequence of $A[i..m]$ and $B[j..n]$.

2. Recursive implementation.

$\underline{\texttt{LCS}(i,j)}$

1. If $i > m$ or $j > n$ then return 0.
2. If $A[i] = B[j]$ then return $1 + \texttt{LCS}(i+1, j+1)$.
3. If $A[i] \neq B[j]$ then return $\max\{\texttt{LCS}(i, j+1), \texttt{LCS}(i+1, j)\}$.

3. Running time with caching / dynamic programming. We have $O(mn)$ subproblems, each of which takes $O(1)$ time. With dynamic programming, we only solve each subproblem once, giving $O(mn)$ time overall.

4. Solving the original problem. To get the length of the longest common subsequence of A and B, we return $\texttt{LCS}(1,1)$.

Rubric.
1 pt. Recursive spec.
1 pt. Recursive implementation (of the spec.)
.5 pts. Base cases
.5 pts. Inductive step
.25 pts. Mention dynamic programming.
.5 pts. Running time
.25 pts. How to solve the original problem.

Problem 2: shortest common supersequence.

1. Recursive spec / induction hypothesis. $\texttt{LCSuper}(i,j)$= the length of the shortest common supersequence of $A[i..m]$ and $B[j..n]$.

2. Recursive implementation.

$\underline{\texttt{LCSuper}(i,j)}$

1. If $i > m$ and $j > n$ then return 0.
2. If $i > m$ then return $1 + \texttt{LCSuper}(i, j+1)$.
3. If $j > n$ then return $1 + \texttt{LCSuper}(i+1, j)$.
4. If $A[i] = B[j]$ then return $1 + \texttt{LCSuper}(i+1, j+1)$.
5. If $A[i] \neq B[j]$ then return $\min\{1 + \texttt{LCSuper}(i, j+1), 1 + \texttt{LCSuper}(i+1, j)\}$.

3. Running time with caching / dynamic programming. We have $O(mn)$ subproblems, each of which takes $O(1)$ time. With dynamic programming, we only solve each subproblem once, giving $O(mn)$ time overall.

4. Solving the original problem. To get the length of the shortest common supersequence of A and B, we return `LCSuper(1,1)`.

> *Rubric.*
> **1 pt.** Recursive spec.
> **1 pt.** Recursive implementation (of the spec.)
> **.5 pts.** Base cases
> **.5 pts.** Inductive step
> **.25 pts.** Mention dynamic programming.
> **.5 pts.** Running time
> **.25 pts.** How to solve the original problem.

Problem 3: longest common palindrome.

1. Recursive spec / induction hypothesis. `LCP(`i,j,k,l`)`$=$ the length of the longest common palindrome of $A[i..j]$ and $B[k..\ell]$.

2. Recursive implementation.

`LCP(`i,j,k,l`)`

1. If $i > j$ or $k > \ell$ then return 0.
2. If $i = j$ and $k = \ell$ and $A[i] = B[k]$ then return 1.
3. If $i < j$, $k < \ell$, and $A[i] = A[j] = B[k] = B[\ell]$, then return $2 +$ `LCP(`$i+1,j-1,k+1,\ell-1$`)`
4. Otherwise return $\max\{$`LCP(`$i+1,j,k,l$`)`$,$ `LCP(`$i,j-1,k,l$`)`$,$ `LCP(`$i,j,k+1,l$`)`$,$ `LCP(`$i,j,k,\ell-1$`)`$\}$.

3. Running time with caching / dynamic programming. We have $O\big(m^2n^2\big)$ subproblems, each of which takes $O(1)$ time. With dynamic programming, we only solve each subproblem once, giving $O\big(m^2n^2\big)$ time overall.

4. Solving the original problem. To get the length of the longest common palindrome subsequence of A and B, we return `LCP(`$1,m,1,n$`)`.

> *Rubric.*
> **1 pt.** 1.5 Recursive spec.
> **1.5 pts.** Recursive implementation (of the spec.)
> **.75 pts.** Base cases
> **.75 pts.** Inductive step
> **.25 pts.** Mention dynamic programming.
> **.5 pts.** Running time
> **.25 pts.** How to solve the original problem.

Exercise 5.13. Subset Subs™, the world's first dynamically optimal deli, has the following delicious business model. Subset Subs makes only two submarine sandwiches each day:

- A ham sub m inches long, and
- A vegetarian sub n inches long,

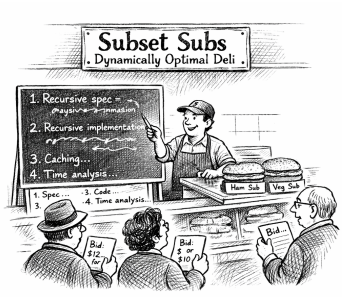
for integers m and n, depending on market conditions. There are also plans to expand to a vegan option, but not hoagies.[17]

Subset Subs sells its sandwiches fractionally and dynamically as follows. Each day, k customers submit bids on the sandwiches, each taking one of the following forms:

1. x dollars for y inches of the ham sandwich.
2. x dollars for y inches of the vegetarian sandwich.
3. A *disjunctive order*, specifying either x_1 dollars for y_1 inches of the ham sandwich, *or* x_2 dollars for y_2 inches of the vegetarian sandwich.

The disjunctive order means the customer would accept either of their listed options. The number of inches is always given as an integer.

There is usually too much demand to serve all of the bids. Each day, Subset Subs decides which of the bids to serve, with the objective of making the maximum amount of money. This means they choose a subset of ham orders, vegetarian orders, and disjunctive orders, choosing one of the two options for each disjunctive order they take, subject to having enough ham sandwich and enough vegetarian sandwich to serve all the orders they take. (We call this a "feasible" subset of orders.)



"Your sandwich will be ready in polynomial time."

In addition to the dollar amounts from the bids, Subset Subs also takes into account the ancillary costs when serving a piece of the sandwich, as a function of its length. This can reflect small things, like the wrapping paper, to the unusual logistics involved when someone orders an extremely large piece.

Ultimately, the input consists of the lengths m and n; k bids, each of one of the three types above; and a lookup function $\texttt{cost} : \mathbb{Z}_{\geq 0} \to \mathbb{R}_{>0}$ that, given x, returns the ancillary cost of producing an x-inch long piece of sandwich, in $O(1)$ time. Design and analyze an algorithm to compute the maximum amount of profit that can be obtained by accepting a feasible set of orders. (As usual, the faster the better. Here we allow running times that are polynomial in m and n, even though they are specified in $\log(m)$ and $\log(n)$ bits, respectively.)[18]

Solution to exercise 5.13.

1. Recursive spec / induction hypothesis. $\texttt{Profit}(i, h, v)$: For $i, h, v \in \mathbb{N}$, $\texttt{Profit}(i, h, v)$ returns the

[17] According the Guinness Book of World Records, the longest sandwich measured 735 m (2,411 ft 5 in) and was created by members of three teams in total. These teams were Groupe Notre Dame Hazmieh-Scouts de L'Independence (Lebanon), Municipality of Hazmieh (Lebanon)and Mini-B chain restaurants (Lebanon). The attempt took place in Hazmieh village, Beirut, Lebanon, on 22 May 2011.

The sandwich started at Notre Dame des Soeurs Antonines School and ended on Elie Street, in Hazmieh, Beirut, Lebanon. The width of the sandwich was 12.5cm and the overall estimated weight of the sandwich is 577.03kg.

[18] You might find it helpful to first solve the simpler case of just one type of sandwich and no "disjunction orders".

maximum profit that can be made from orders $i, i+1, ..., k$ with h inches of ham sandwich and v inches of vegetarian sandwich.

2. Recursive implementation.

`Profit`(i, h, v)

1. If $i > k$ then return 0.
2. Let `best` $\leftarrow$ `Profit`$(i+1, h, v)$. *// Skip order i.*
3. If order i has a ham option of x dollars for y inches, and $y \leq h$:
   A. `best` $\leftarrow \max\{$`best`$,\ x -$ `cost`$(y) +$ `Profit`$(i+1, h-y, v)\}$.
4. If order i has a vegetarian option of x dollars for y inches, and $y \leq v$:
   A. `best` $\leftarrow \max\{$`best`$,\ x -$ `cost`$(y) +$ `Profit`$(i+1, h, v-y)\}$.
5. Return `best`.

3. Solving the original problem. We return `Profit`$(1, m, n)$.

4. Running time with caching / dynamic programming. We apply dynamic programming to the recursive algorithm above. We have mnk subproblems, and each subproblem takes $O(1)$ time, so the algorithm takes $O(mnk)$ time overall.

5. Proof. We prove that the algorithm implements the recursive spec by induction on the number of remaining orders $(k - i + 1)$. In the base case there are no orders $(i > k)$, so no profit can be made, and the algorithm correctly returns 0. In the general case $i \leq k$. By the inductive hypothesis, `Profit`$(i+1, h', v')$ correctly returns the maximum profit from orders $i+1, \ldots, k$ for all values of h' and v'. Any feasible selection from orders $i, \ldots, k$ either skips order i, takes its ham option, or takes its vegetarian option (the latter two when they exist and are feasible). The algorithm computes the profit for each of these choices—using the inductive hypothesis for the recursive calls—and returns the maximum. This is exactly the maximum profit from orders $i, \ldots, k$ with h inches of ham and v inches of vegetarian sandwich.

Rubric.
3 pts. Recursive spec
- **2 pts.** Parametrized by number of orders, inches of ham sandwich, and inches of vegetarian sandwiches.
- **1 pt.** Objective: Maximum profit

3.5 pts. Implementation
- **.5 pts.** Base case: No more orders
- **1 pt.** .5 Recursive call for skipping.
- **1 pt.** .5 Checking enough ham/veggie sandwich remains before considering option.
- **1 pt.** .5 Adding in bid price.
- **1 pt.** .5 Subtracting cost.
- **1 pt.** .5 Updating the remaining ham/veggie sandwich in recursive calls.
- **1 pt.** .5 Correctly handles disjunctive orders.

.5 pts. Solving original problem
1 pts. Running time
- **0.5 pts.** "Dynamic programming"
- **0.5 pts.** Correct running time analysis

2 pts. Proof by induction
- **.5 pts.** Base case
- **1.5 pts.** Inductive step

F.4 Homework 3

Exercise 6.5. Let $G = (V, E)$ be a directed graph. We say two vertices $s, t \in V$ are *half-connected* if either s can reach t or t can reach s. We say that the graph G is *half-connected* if every pair of vertices is half-connected. We say that two vertices s and t are *strictly half-connected* if either s can reach t, or t can reach s, but not both. We say that G is strictly half-connected if every pair of vertices is strictly half-connected.

1. Consider the problem of deciding if G is half-connected. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. Consider the problem of deciding if G is strictly half-connected. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 6.5.

Problem 1.

`half-connected`($G = (V, E)$)

1. Compute the strongly connected components in topological order. // $O(m + n)$
2. Construct the DAG $G_{\text{SCC}} = (V_{\text{SCC}}, E_{\text{SCC}})$ of strongly connected components. // $O(m + n)$
3. $C_1, \ldots, C_k$ be the strongly connected components in topological order. // $O(m + n)$.
4. If there is an edge from C_i to C_{i+1} in G_{SCC} for all $i = 1, \ldots, k - 1$, then return `true`. // $O(m + n)$
5. Otherwise return `false`.

Running time. The running time is $O(m + n)$ as annotated in the code.

Proof of correctness. Let $C_1, \ldots, C_k$ list the strongly connected components in topological order. The algorithm returns `true` iff there is an edge from C_i to C_{i+1} for each $i = 1, \ldots, k - 1$. We want to show that the latter is equivalent to G being weakly connected.

Observe G is weakly half connected iff for every two strongly connected components C_i and C_j, C_i and C_j are weakly connected in the graph G_{SCC} obtained by contracting the strongly connected components. If $i < j$, then since $C_1, \ldots, C_k$ lists the components in topological order, C_i and C_j are weakly connected iff C_i can reach C_j.

For $i = 1, \ldots, k - 1$, C_i can reach C_{i+1} since C_i cannot reach $C_1, \ldots, C_{i-1}$, and $C_{i+2}, \ldots, C_k$ cannot reach C_{i+1}, C_i can reach C_{i+1} iff there is an edge from C_i to C_{i+1}.

Thus if G (hence G_{SCC}) is weakly connected, then there is an edge from C_i to C_{i+1} for each i.

Conversely, if there is an edge from C_i to C_{i+1} for each i, then C_i can reach C_j for all $i < j$ (along the path $(C_i, C_{i+1}, \ldots, C_j)$). □

Rubric.
3 pts. Correct and efficient algorithm.
-1.5 pts. For a correct but inefficient algorithm.
1 pt. Correct running time analysis of the described algorithm.
1 pt. Proof of correctness.

Problem 2.

`strictly-half-connected(`$G = (V, E)$`)`

1. Compute the strongly connected components (SCCs) of G. // $O(m + n)$
2. If any SCC has more than one vertex, return `false`. // $O(m + n)$
3. Otherwise return `half-connected(`G`)`. // $O(m + n)$

Running time. The running time is $O(m + n)$ as annotated in the code.

Proof of correctness. G is strictly half-connected if it is half-connected, but no two vertices are strongly connected. We test the latter by computing the strongly connected components and seeing if there is any SCC with more than one vertex. We test the former by calling `half-connectivity`.

Rubric.
3 pts. Correct and efficient algorithm.
-1.5 pts. For a correct but inefficient algorithm.
1 pt. Correct running time analysis of the described algorithm.
1 pt. Proof of correctness.

Exercise 7.1. Each of the following problems describes a graph problem over an unweighted directed graph $G = (V, E)$ with m edges and n vertices. Each problem can be solved by constructing an auxiliary graph and calling a shortest path algorithm from this chapter as a black box. For each problem, (a) design an auxiliary graph and shortest path problem, (b) indicate which algorithm to apply and how, and (c) analyze the running time. No proof of correctness is required.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[19]

2. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is a multiple of 5.

3. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is either a multiple of 2, or a multiple of 3.

4. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is either a multiple 3 or a multiple of 4, but not a multiple of 12.

[19] For this and the other problems, if there is no such walk, then your algorithm should return $+\infty$.

5. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is of the form $5x + 8y$ for $x, y \in \mathbb{Z}_{\geq 0}$.

6. Suppose G is a patriotic directed graph where all the edges are colored either red, white, or blue. An *American walk* is a walk where the edges alternate in color in the order of the American dream: red, white, blue, red, white, blue... Here the first edge could be any color and then the colors have to cycle through the American dream thereafter. Compute the length of the shortest American walk from s to t.

7. Suppose G is again a patriotic directed graph where all the edges are colored either red, white, or blue. An *un-American walk* is a walk that never cycles through the American dream; that is, a walk where no three consecutive edges in the walk have colors that form any of the following three sequences: $(\text{red}, \text{white}, \text{blue})$, $(\text{white}, \text{blue}, \text{red})$, or $(\text{blue}, \text{red}, \text{white})$. Compute the length of the shortest un-American walk from s to t.

8. Let $s, t \in V$ be two vertices. Consider the following game with two people Alice and Bob. Initially, Alice is on s, and Bob is on t. Each round, Alice and Bob are on two vertices, and they simultaneously take an outgoing edge to another vertex. The goal is to guide Alice and Bob to the same vertex with the minimum number of rounds or declare that it is impossible to have them meet. Compute the minimum number of rounds needed to have Alice and Bob meet at the same vertex.

Solution to exercise 7.1.

Below we assume that BFS reports the distance from s to t as $+\infty$ when s cannot reach t. The leading high-level descriptions are not formally part of the solution, and are included to help explain the intuition behind the construction. (We will look for an explicit and unambiguous construction of the auxiliary graph when grading.)

Problem 3. The high-level idea is simple. We know how to do multiples of 2 – this is just an even length walk. Similarly one can handle multiples of 3, by having 3 layers instead of 2. So build those two auxiliary graphs separately, run BFS on each, and report back the shorter of the two solutions.

Formally we create two auxiliary graphs $G_2 = (V_2, E_2)$ and $G_3 = (V_3, E_3)$. G_2 is the exact same graph as from the even length walk; see Page 159. For G_3, we define:

$$V_3 = \{v_0, v_1, v_2 : v \in V\}$$
$$E_3 = \{(u_i, v_{i+1 \bmod 3}) : (u, v) \in E \text{ and } i = 0, 1, 2, 3\}.$$

We run BFS in G_2 to find the distance from s_{EVEN} to t_{EVEN} (in the terminology of the solution on Page 159). We run BFS in G_3 from s_0 to find the distaince from S_0 to t_0. We return the smaller of the two values.

Problem 5. There are multiple ways to do this and here we describe one. Informally, from previous problems, we know how to deal with multiples of 5 and multiples of 8, separately. For multiples of 5, we have 5 layers of vertices with arcs between consecutive layers (modulo 5). For multiples of 8, we have 8 layers of vertices with arcs between consecutive layers (modulo 8). I visualize this as two vertical stacks of pancakes, one with 5 pancakes and one with 8 pancakes, side by side, with arcs going down the pancake and looping back from bottom to top. Now to allow multiple of 5 *or* 8, I

also add arcs from the "bottom" (i.e., last layer) of the 5-layer graph to the top (first layer) of the 8-layer graph, and from the bottom of the 8-layer graph to the top of the 5 layer graph. This way, as a vertex takes its 5th step, it can switch into the other " mod 8" auxiliary graph, and vice versa. We also add a super source $s^\star$ to allow the walk to start with a multiple of 5 or a multiple of 8, and a super sink $t^\star$ to allow the walk to end at a multiple of either 5 or 8.

Let $G' = (V', E')$ where

$$\begin{aligned}
V' &= \{v_{a,i} : v \in V,\, i = 0,1,2,3,4\} \\
&\quad \cup \{v_{b,i} : v \in V,\, i = 0,1,2,3,4,5,6,7\} \\
&\quad \cup \{s^\star, t^\star\} \\
E' &= \{(u_{a,i}, v_{a,i+1 \bmod 5}) : (u,v) \in E,\, i = 0,\dots,4\} \\
&\quad \cup \{(u_{b,i}, v_{b,i+1 \bmod 8}) : (u,v) \in E,\, (i = 0,\dots,7)\} \\
&\quad \cup \{(u_{a,4}, v_{b,0}) : (u,v) \in E\} \\
&\quad \cup \{(u_{b,7}, v_{a,0}) : (u,v) \in E\} \\
&\quad \cup \{(s^\star, s_{a,0}),\, (s^\star, s_{b,0}),\, (t_{a,0}, t^\star),\, (t_{b,0}, t^\star)\}
\end{aligned}$$

We run BFS from $s^\star$ and return the distance to $t^\star$, minus 2. The running time is $O(m+n)$ as G' has $O(m)$ edges and $O(n)$ vertices.

Problem 6. The high-level idea is to have 3 layers of vertices, which we imagine being colored "red", "white", and "blue", to remember the color of the last edge taken. This way we know the color of the next edge we are supposed to take. Edges are added between layers based on the color. (White arcs go from the red layer to the white layer, blue arcs go from the white layer to the blue layer, and so forth.)

Let $G' = (V', E')$ where

$$\begin{aligned}
V' &= \{v_{\texttt{red}}, v_{\texttt{white}}, v_{\texttt{blue}} : v \in V\} \cup \{s^\star, t^\star\} \\
E' &= \{(u_{\texttt{blue}}, v_{\texttt{red}}) : e = (u,v) \in E \text{ is } \texttt{red}\} \\
&\quad \cup \{(u_{\texttt{red}}, v_{\texttt{white}}) : e = (u,v) \in E \text{ is } \texttt{white}\} \\
&\quad \cup \{(u_{\texttt{white}}, v_{\texttt{blue}}) : e = (u,v) \in E \text{ is } \texttt{blue}\} \\
&\quad \cup \{(s^\star, s_{\texttt{red}}), (s^\star, s_{\texttt{blue}}), (s^\star, s_{\texttt{white}})\} \cup \{(t_{\texttt{red}}, t^\star), (t_{\texttt{blue}}, t^\star), (t_{\texttt{white}}, t^\star)\}.
\end{aligned}$$

We run BFS from $s^\star$ and return the distance from $s^\star$ to $t^\star$, minus 2. The running time is $O(m+n)$ is $O(m+n)$ as G' has $O(m)$ edges and $O(n)$ vertices.

Problem 7. Let $G' = (V', E')$ where

$$V' = \{v_{XY}, : v \in V,\ X, Y \in \{\texttt{red}, \texttt{white}, \texttt{blue}\}\} \cup \{s^\star, t^\star\},$$

$$E' = \left\{ \begin{array}{ll} (u_{XY}, v_{Y\texttt{red}}) : & e = (u,v) \in E \text{ is } \texttt{red}, \\ & X, Y \in \{\texttt{red}, \texttt{white}, \texttt{blue}\}, \\ & (X,Y) \neq (\texttt{white}, \texttt{blue}) \end{array} \right\}$$

$$\cup \left\{ \begin{array}{ll} (u_{XY}, v_{Y\texttt{white}}) : & e = (u,v) \in E \text{ is } \texttt{white}, \\ & X, Y \in \{\texttt{red}, \texttt{white}, \texttt{blue}\}, \\ & (X,Y) \neq (\texttt{blue}, \texttt{red}) \end{array} \right\}$$

$$\cup \left\{ \begin{array}{ll} (u_{XY}, v_{Y\texttt{blue}}) : & e = (u,v) \in E \text{ is } \texttt{blue}, \\ & X, Y \in \{\texttt{red}, \texttt{white}, \texttt{blue}\}, \\ & (X,Y) \neq (\texttt{red}, \texttt{white}) \end{array} \right\}$$

$$\cup \{(s^\star, s_{XY}) : X, Y \in \{\texttt{red}, \texttt{white}, \texttt{blue}\}\}$$

$$\cup \{(t_{XY}, t^\star) : X, Y \in \{\texttt{red}, \texttt{white}, \texttt{blue}\}\}.$$

We run BFS from $s^\star$ and return the distance from $s^\star$ to $t^\star$, minus 2. The running time is $O(m+n)$ is $O(m+n)$ as G' has $O(m)$ edges and $O(n)$ vertices.

Problem 8. You might be tempted to run BFS from both Alice and Bob, and compare the results to see if there is a vertex that has the same distance from both Alice and Bob. But BFS only reports back shortest path lengths, and its possible that for Alice and Bob to meet up, neither should take a shortest path.

The key idea is to observe that the "states" are the *simultaneous* locations of Alice and Bob. Each "step" corresponds to both Alice and Bob moving one arc. We can model a state as the pair of vertices (u, v) that Alice and Bob are on. We have edges from one coordinate/vertex pair to another based on if each coordinate corresponds to an arc in the input graph.

Let $G^2 = (V^2, E^2)$ where

$$V^2 = \{(u,v) : u, v \in V\} \cup \{z^\star\},$$

$$E^2 = \{((u_1, u_2), (v_1, v_2)) : (u_1, v_1), (u_2, v_2) \in E\}$$
$$\cup \{((v,v), z^\star) : v \in V\}.$$

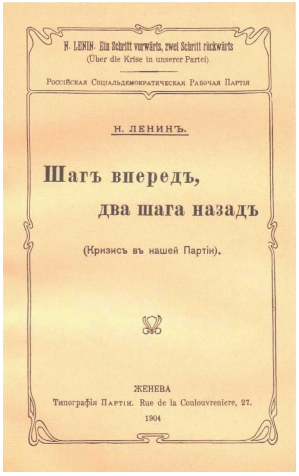
We run BFS from (s,t) and return the distance from (s,t) to $z^\star$, minus 1. The running time is $O\big(m^2 + n^2\big)$ as G^2 has m^2 edges and n^2 vertices.

Rubric.
1 pt. Correct and efficient graph construction.
- **-.5 pts.** If the graph construction leads to a correct solution, but is inefficient. (e.g., creates a graph a polynomial factor larger than necessary.)

.5 pts. Applying BFS correctly.
- **-.25 pts.** If the output is off by one or two (e.g., forgetting to account for extra super-source and super-sink edges).
- **-.25 pts.** Apply Dijkstra instead of BFS.

.5 pts. Correctly analyzing the running time of the described algorithm.

Exercise 7.8. *One Step Forward, Two Steps Back: The Crisis in Our Party* (Russian, romanized: *Shag vperyod, dva shaga nazad (Krizis v nashey partii)*) is a work written by Vladimir Lenin and published on May 6/19, 1904. In it Lenin defends his role in the 2nd Congress of the Russian Social Democratic Labour Party, held in Brussels and London from July 30 to August 23, 1903. Lenin examines the circumstances that resulted in a split within the party between a Bolshevik ("majority") faction, led by himself, and a Menshevik ("minority") faction, led by Julius Martov.



Written in 1904 in response to controversies within the Social Democratic Labour Party's Second Congress regarding the status of party membership and organization, Lenin frames this conflicting factionalism within the Party in terms of dialectics. According to Lenin, there are two conflicting factions within the party: "the revolutionaries", which consists of the majority of party members (the Bolsheviks) and "the opportunists", which are the minority (the Mensheviks).
Rosa Luxemburg, a Marxist living in Germany at the time, responded to the article in *Organizational Questions of the Russian Social Democracy* (1904). She criticized Lenin's attitude towards democratic centralism and wrote about the role of "spontaneity" among the working class. However, different authors make varying claims about her precise attitude towards Lenin, the Bolshevik faction and the revolutionary situation in Russia.

Let $G = (V, E)$ be a directed graph. A *Lenin walk* is defined as a sequence of vertices $v_1, ..., v_k$ such that for each consecutive pair of vertices (v_i, v_{i+1}):

- If $i \mod 3 = 1$, then $(v_i, v_{i+1}) \in E$.
- If $i \mod 3 \in \{0, 2\}$, $(v_{i+1}, v_i) \in E$.

If we call each pair (v_i, v_{i+1}) in the Lenin walk a "step", then a Lenin walk alternates between one step going forwards along an edge and two steps each going "backwards", in the opposite direction of an edge.

Given G and $s, t \in V$, consider the problem of computing the infimum number of vertices along any Lenin walk from s to t. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 7.8. We build an auxiliary graph H consisting of:

- Three copies of V, called V_0, V_1, and V_2. We let v_i denote the ith copy of V (in V_i).
- For each edge $(u, v) \in E$, we add the arcs (u_0, v_1), (v_1, u_2), and (v_2, u_0).

H has $O(n)$ vertices and $O(m)$ edges. We run BFS from s_0 and return the shortest distance to either t_0, t_1, or t_2, plus 1.

For every Lenin walk from s to t with ℓ edges, we have a corresponding walk in H from s to t_{ℓ_i}, with the same number of edges, where the "forward" steps go from V_0 to V_1 and the two backwards edges go from V_1 to V_2 and then from V_2 to V_0.

Conversely, given a walk from s to t_i in H (for any i), projecting onto the underlying vertices gives a Lenin walk from s to t with the same number of edges.

This bijection shows that computing the length of the shortest walk in H from s to any copy of t is equivalent to computing the minimum number of steps in any Lenin walk from s to t. The number of vertices in a Lenin walk is the number of steps plus 1.

Rubric.
4 pts. Build auxiliary graph
- **1 pt.** Any attempt to build an auxiliary graph
- **2 pts.** 3 layer graph
- **1 pt.** Correct edges (correctly flipped)

1.5 pts. Apply BFS instead of Dijkstra
- **.5 pts.** Use BFS instead of Dijkstra
- **1 pt.** From s_0 to t_0, t_1, t_2 (or super sink, but should be careful about output value).

.5 pts. Plus one to count vertices
2 pts. Running time analysis
- **1 pt.** Count number of edges in auxiliary graph
- **1 pt.** Plug into BFS

2 pts. Proof
- **1 pt.** Go from Lenin walks to auxiliary graph
- **1 pt.** Go from auxiliary graph to Lenin walks

F.5 Midterm 1

Exercise 5.15. John McCarthy[20] invented Lisp in 1960. Lisp has an extraordinarily simple syntax: a Lisp program is a nested composition of *S-expressions*. An S-expression is either

1. an *atom*: a single token representing a name or value (such as `x`, `42`, or `nil`), or
2. a *list*: written in prefix notation as $(f\ e_1\ e_2\ e_3\ \cdots\ e_k)$ where f is a token (the *operator*) and $e_1, \ldots, e_k$ are S-expressions (the *arguments*). In this problem we restrict to $k \in \{0, 1, 2\}$.[21]

This uniform representation, in which code and data are the same kind of object, makes Lisp *homoiconic*, and is the source of much of its expressive power.[22]

The parentheses make the structure of a Lisp program completely unambiguous. But suppose a Lisp program has been "flattened" and lost all of its parentheses, leaving only the flat sequence of tokens read left to right. For example, the S-expression

```
(cons (quote x) (quote y))
```

flattens to the sequence

```
cons quote x quote y
```

Can we recover a valid S-expression from such a sequence?

We model this as follows. Let $C[1..n]$ be an array of n tokens. We say that $C[1..n]$ is *feasible* if there exists a single S-expression e where the flattened sequence of e equals C. We say that C is *uniquely feasible* if the choice of e is unique.

Suppose you have access to a constant time function `type` that returns, for each token $C[i]$, the set of roles it can play:

$$\texttt{type}(C[i]) \subseteq \{\texttt{atom}, 0, 1, 2\}.$$

Here `atom` means the token can serve as an atom, and $k \in \{0, 1, 2\}$ means it can serve as an operator taking k arguments. A single token may have multiple roles; for example, a token like "`-`" might have one argument (`(- x)` negates x) or two (`(- x y)` subtracts y from x). A 0-argument operator represents a function that takes no arguments.

1. (9 points) Design and analyze an algorithm (the faster the better) that, given $C[1..n]$, decides whether C is feasible.

2. (1 point) Briefly describe how to modify your algorithm to decide whether C is *uniquely* feasible.

Solution to exercise 5.15.

[20] McCarthy's 1971 Turing Award lecture, "Generality in Artificial Intelligence," is available at https://dl.acm.org/doi/epdf/10.1145/33447.33448.

[21] For simplicity, our operators are restricted to single tokens, and not nested S-expressions.

[22] See "The Roots of Lisp", by Paul Graham, available at https://paulgraham.com/rootsoflisp.html

Part 1. We use dynamic programming.

1. Recursive spec / induction hypothesis. Let $C[1..n]$ be the token array. We define

> $\texttt{expr}(i, j) = \texttt{true}$ if and only if the subsequence $C[i..j]$ can be parsed as a valid S-expression consistent with `type`.

2. Recursive implementation.

> `expr`(i, j)
>
> 1. If $i = j$, and either `atom` $\in$ `type` $(C[i])$ or $0 \in \texttt{type}(C[i])$, then return `true`.
> 2. If $j > i$, $1 \in \texttt{type}(C[i])$, and $\texttt{expr}(i + 1, j)$ is `true`, then return `true`.
> 3. If $j > i + 1$ and $2 \in \texttt{type}(C[i])$:
>   A. If $\texttt{expr}(i + 1, k)$ and $\texttt{expr}(k + 1, j)$ are both `true` for some $k \in \{i + 1, \ldots, j - 1\}$, then return `true`.
> 4. Otherwise return `false`.

3. Solving the original problem. $\texttt{expr}(1, n)$ returns whether $C[1..n]$ is feasible.

4. Running time with caching / dynamic programming. We apply dynamic programming to the recursive algorithm above. There are n choices of i and n choices of j, hence $O(n^2)$ subproblems. Each subproblem takes $O(n)$ time (excluding recursive calls) where the bottleneck is iterating over k. Thus the total running time is

$$(\#\text{ subproblems}) \times \left(\begin{array}{c} \text{time per subproblem} \\ \text{excl. recursive calls} \end{array} \right) = O(n^2) \cdot O(n) = O(n^3).$$

Part 2. We extend the recursive spec and algorithm from returning `true`/`false` to instead return the number of S-expressions that flatten to C. We then check to see if that number over all of S is 1.

Exercise 6.10. Let $G = (V, E)$ be a directed graph. We say an edge (u, v) is *irreversible* if there is no walk from v to u in the graph. Design and analyze an algorithm (the faster the better) that computes all the irreversible edges in G.

Solution to exercise 6.10. We compute the strongly connected components of G, and return the set of all edges that go across components. This takes $O(m + n)$ time.

To prove correctness, we claim that an edge is irreversible iff it is not contained in a strongly connected component.

Suppose an edge (u, v) is irreversible. Then v cannot reach u, so u and v are in different connected components.

Suppose an edge (u, v) is contained in a strongly connected component. Then v can reach u, hence (u, v) is reversible (i.e., not irreversible).

Exercise 7.10. The Economist recently reported[23] that the number of toll roads in America is increasing rapidly. It used to be hard enough to find the shortest path. Now you have to do it within a budget.

Right now, drivers pay to access 6,300 miles of Americas 160,000 miles (or so) of highway, but that number is going up. Here in Indiana, in June, the state passed a law giving the state authority to raise tolls on all its existing interstate highways. The Economist says that this trend is broadly due to a parallel trend of decreasing taxes on gasoline, which used to pay a larger share of the cost for maintaining roads and highways.

Let $G = (V, E)$ be a directed graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$ and *nonnegative integer edge costs* $\texttt{cost} : E \to \mathbb{Z}_{\geq 0}$, which model toll roads. We define the *cost* of a walk in G as the sum of edge costs (with repetition) along that walk. Design and analyze an algorithm (the faster the better) that, given $k \in \mathbb{N}$, and $s, t \in V$, computes the length of the shortest (s, t)-walk that costs at most k. Your running may depend polynomially on k (even though k only takes $\log(k)$ bits to represent). (It might help to imagine $k \leq O(n)$.)

Solution to exercise 7.10. At a high level, we create a $(k+1)$-layer auxiliary graph H, where the layers encode the amount of money spent so far, and edges between layers model token costs.

For each vertex $v \in V$ and $i \in \{0, \dots, k\}$, we create an auxiliary vertex v_i. For every edge $e = (u, v)$, and every $i \in \{0, \dots, k - c(e)\}$, we add the arc $(u_i, v_{i+c(e)})$ with length $\ell(e)$. In total, H has $O(kn)$ vertices and $O(km)$ edges.

We run Dijkstra's algorithm from s_0 in H, which takes $O(mk + nk \log(nk))$ time. We return the minimum distance from s_0 to t_i over all i.

To prove correctness, we give a length-preserving bijection between (s, t) walks of cost $\ell \leq k$ and walks from (s_0, t_ℓ). It follows that the length of the shortest (s, t)-walk with cost at most k is equal to the minimum distance from s_0 to t_ℓ in H over all $\ell \leq k$.

Given an (s, t)-walk of cost $\ell \leq k$ in G, we have the (s_0, t_ℓ)-walk in H where for each arc $e = (u, v)$ along the walk, if the prefix of the walk cost i, we have the arc $(u_i, v_{i+c(e)})$. We always have $i + c(e) \leq k$ because (s, t) costs at most k. This defines an (s_0, t_ℓ)-walk in H with the exact same length.

Conversely, given an (s_0, t_ℓ)-walk W in H for some ℓ, consider the walk in G where each auxiliary vertex u_i is replaced with the underlying vertex u. An arc (u_i, v_j) in W maps to the arc (u, v) with cost $j - i$, so the total cost is a telescoping sum equal to ℓ.

This gives the length-preserving bijection between (s, t) walks of cost $\ell \leq k$ and (s_0, t_ℓ)-walks in H, completing the proof.

Exercise 1.13. Let $A[1..n]$ be an array where each entry is colored `red` or `blue`, such that $A[1]$ and $A[n]$ have different colors. (This also implies that $n > 1$.) We say that a pair of consecutive indices $(i, i+1)$ is *colorful* if $A[i]$ and $A[i+1]$ have different colors. Design and analyze an algorithm (the faster the better) that computes a colorful consecutive pair of indices $(i, i+1)$ in A, or `nil` if there is no colorful pair of indices.

Solution to exercise 1.13.

[23] *Toll roads are spreading in America*, December 18, 2025. https://www.economist.com/united-states/2025/12/18/toll-roads-are-spreading-in-america

Recursive spec. `search`$(A[1..n])$: Given an array $A[1..n]$ where $A[1]$ and $A[n]$ have different colors, returns a colorful pair of consecutive indices $(i, i+1)$.

Recursive implementation.

`search`$(A[1..n])$

1. If $n = 2$ then return $(1, 2)$.
2. Otherwise let $i = \lceil n/2 \rceil$.
3. If $A[1]$ and $A[i]$ have different colors then return `search`$(A[1..i])$.

/* *Otherwise $A[i]$ and $A[n]$ have different colors.* */

4. Return $i - 1 +$ `search`$(A[i..n])$.

Running time analysis. The running time is modeled by

$$T(n) = T(n/2) + O(1),$$

hence $T(n) = O(\log n)$.

F.6 Homework 4

Exercise 8.1. Recall the $O(n^3)$ dynamic programming algorithm for APSP. If the edge lengths were nonnegative, then we could have just run Dijkstra's algorithm from every vertex in $O(mn + n^2 \log n)$ time, which is never worse than $O(n^3)$, and significantly better when the graph is sparse. Here we will develop a classic algorithm due to Johnson [Joh77b] that builds on this observation. (Obviously you can find this algorithm in any book, but the point is that you have the background to discover it yourself.)

Let $G = (V, E)$ be a directed graph with real-valued edge weights $\ell : E \to \mathbb{R}$. For simplicity we assume there are no negative cycles in G. Let $d(s, t)$ denote the distance with respect to ℓ. We define *vertex potentials* as a set of values $\varphi : V \to \mathbb{R}$ for each vertex. For a set of vertex potentials φ, consider the modified edge weights $\ell_\varphi : E \to \mathbb{R}$ defined by

$$\ell_\varphi(u, v) = \varphi(u) + \ell(u, v) - \varphi(v).$$

Let d_φ denote distances with respect to ℓ_φ.

1. Prove that for any walk $w : s \rightsquigarrow t$,

$$\bar{\ell}_\varphi(w) = \varphi(s) + \bar{\ell}w - \varphi(t).$$

Since all (s, t)-walks shift by the same quantity $\varphi(s) - \varphi(t)$, the shortest paths stay the same. We also have $d_\varphi(s, t) = \varphi(s) + d(s, t) - \varphi(t)$ for all s, t.

2. Consider the potentials defined by $\varphi(v) = \min_{s \in V} d(s, v)$. Prove that for these potentials, ℓ_φ is nonnegative.

3. Clearly the potentials above can be computed in $O(mn^2)$ time by running Bellman-Ford for every choice of source s. Design and analyze an algorithm computing φ in $O(mn)$ time.

4. Finally, put everything together to obtain Johnson's $O(mn + n^2 \log n)$-time algorithm for APSP.

Solution to exercise 8.1.

Part 1. Let $(v_0 = s), v_1, \ldots, v_{k-1}, (v_k = t)$ be the vertices along the walk. We have

$$\begin{aligned}\bar{\ell}_\varphi(w) &= \sum_{i=0}^{k-1} \ell_\varphi(v_i, v_{i+1}) = \sum_{i=0}^{k-1} \varphi(v_i) + \ell(v_i, v_{i+1}) - \varphi(v_{i+1}) \\ &= \bar{\ell}(w) + \sum_{i=0}^{k-1} \varphi(v_i) - \varphi(v_{i+1}) = \bar{\ell}(w) + \varphi(s) - \varphi(t)\end{aligned}$$

by telescoping series.

Part 2. Consider an edge (u, v). Let $\varphi(u) = d(s, u)$ and $\varphi(v) = d(t, v)$. We have

$$\ell_\varphi(u, v) = \varphi(u) + \ell(u, v) - \varphi(v) = d(s, u) + \ell(u, v) - d(t, v) \overset{\text{(a)}}{\geq} d(s, v) - d(t, v) \overset{\text{(b)}}{\geq} 0.$$

Here (a) is by the triangle inequality, and (b) is by minimizing choice of t.

Part 3. Consider the auxiliary graph H obtained from G by introducing a super-vertex $s^\star$, and add an edge $(s^\star, v)$ of length 0 for all $v \in V$. We run SSSP from $s^\star$ in $O(mn)$ time; the distances from $s^\star$ give out potentials.

To prove correctness, fix a vertex v. For any walk ending at v, in G, the walk in H to v prepending $s^\star$ has the same length. Conversely, any $s^\star \rightsquigarrow v$ walk in H has the same length as the walk in G starting at the vertex after $s^\star$. It follows that $\varphi(v) = \min_{s \in V} d(s, v)_G = d_H(s^\star, v)$.

Remark F.9. We've done the $s^\star$ trick so many times, I would understand if many folk skip a more "complete proof" of how it works. (On exams we might be more careful.)

Part 4. We first compute the potentials $\varphi(v) = \min_{s \in V} d(s, v)$ in $O(mn)$ time as described in part 3. Consider the reweighted graph G_φ with edge lengths ℓ_φ. By part 2, the edges in G_φ are nonnegative. We run Dijkstra's algorithm in G_φ from every vertex in the graph. This gives us distances $d_\varphi(s, t)$ for all $s, t \in V$. We recover the true distances by computing $d(s, t) = \varphi(t) + d_{pote}(s, t) - \varphi(s)$ for all $s, t \in V$. The bottleneck is the n invocations of Dijkstra's algorithm in G_φ, giving $O\big(mn + n^2 \log n\big)$ time overall.

Rubric.
2 pts. Part 1.
- **1 pt.** Write length of walk as sum of edge lengths.
- **1 pt.** Telescoping series.

2.5 pts. Part 2.
- **1 pt.** Expand out potentials to distances from various vertices.
- **1.5 pts.** Apply the triangle inequality.

3 pts. Part 3.
- **1 pt.** Add super-source with length-0 auxiliary arcs.
- **1 pt.** Bellman-Ford from super-source.
- **1 pt.** Some justification for the super-source.

2.5 pts. Part 4.
- **1.5 pts.** Algorithm
  - **.5 pts.** Compute potentials via part 3.
  - **.5 pts.** Dijkstra's algorithm from all sources.
  - **.5 pts.** Subtract out potentials in the output.
- **1 pt.** Running time analysis

Exercise 9.9. Consider the following special case of SAT, which we will call *k-occurrence-SAT* for a fixed parameter $k \in \mathbb{N}$. The input consists of a SAT formula $f(x_1, \dots, x_n)$ in CNF such that every variable x_i appears (as is, or negated) in at most k clauses. The problem is to decide whether there is a satisfying assignment. For $k = 3$, either (a) design and analyze a polynomial time algorithm, or (b) show that a polynomial time algorithm for k-occurrence-SAT implies a polynomial time algorithm for (CNF-)SAT. [24]

Solution to exercise 9.9. We will show that both problems are still hard. The high level idea is to replace each variable x appearing in many clauses with a copy for each appearance, and force all the copies to be equal with equality gadgets.

Let $f(x_1, \dots, x_n)$ be a fixed formula in CNF. Consider any x_i, and suppose x_i appears in k_i clauses. We create k new variables $y_{i,1}, \dots, y_{i,k_i}$. Each appearance of x_i is replaced by a distinct $y_{i,j}$. At this point each $y_{i,j}$ appears exactly once. We then add equality gadgets for the following pairs:

$$y_{i,1} = y_{i,2},\ y_{i,2} = y_{i,3},\ y_{i,3} = y_{i,4},\ \dots,\ y_{i,k_{i-1}} = y_{i,k_i}.$$

Recall that an equality gadget can be encoded in CNF (e.g., for $y_{i,1} = y_{i,2}$) as

$$(y_{i,1} \vee \bar{y}_{i,2}) \wedge (y_{i,2} \vee \bar{y}_{i,1});$$

In particular, each variable appears *twice*. Thus encoding all of the equality constraints with the gadget above adds (at most) 4 appearances for each $y_{i,j}$; in total, each $y_{i,j}$ appears at most 5 times.

If we do this transformation for all x_i, we obtain a new formula $g(y_{1,1}, y_{1,2}, \dots, y_{n,k_n})$. Observe that g has total size at most a polynomial of the size of f because we only add a constant number of constant-size constraints for each occurrence of each variable. We submit g to a black box for 5-occurrence SAT, and declare that f is satisfiable iff the black box declares that g is satisfiable.

We claim that f has a satisfying solution iff $g/$ does.

If f is satisfiable, then let $x_1, \dots, x_n$ be a satisfying assignment for f. Let $y_{i,j} = x_i$ for all i and j. Then $y_{i,j} = y_{i,k}$ for all i, j, k, so we satisfy all the equality constraints. For the constraints in g corresponding to constraints in f, since these constraints are obtained by replacing each instance of x_i with some $y_{i,j}$, and $x_i = y_{i,j}$, the $y_{i,j}$'s will also satisfy these constraints.

Conversely suppose g is satisfiable. Let $y_{1,1}, \dots, y_{n,k_n}$ by a satisfying solution for g. For each i, all $y_{i,j}$ (over j) must have the same value to satisfy the equality constraints. Define $x_1, \dots, x_n$ by setting $x_i = y_{i,j}$ for all i. (The choice of j doesn't matter since all the values are the same.) For each constraint C_i of f, since the corresponding constraint in g is satisfied by $y_{i,j}$'s, and the x_i's have the same values as the $y_{i,j}$'s, $x_1, \dots, x_n$ also satisfies f. Thus f is satisfiable.

To reduce the number of occurrences to 3, we will combine all the equality gadgets for a single variable into a single gadget that is a little more clever. Taking again x_i above as an example, consider the clause:

$$(y_{i,1} \vee \bar{y}_{i,2}) \wedge (y_{i,2} \vee \bar{y}_{i,3}) \wedge \dots \wedge (y_{i,k_i-1} \vee \bar{y}_{i,k_i}) \wedge (y_{i,k_i} \vee \bar{y}_{i,1}).$$

The above generalizes the two-variable equality gadget to k_i-variables; one can verify that it is satisfied iff all $y_{i,j}$'s are equal.[25] Observe also that each $y_{i,j}$ appears twice. If for each x_i we

[24] As a warmup, it might be helpful to first consider the case $k = 5$. If you figure out 5-occurrence SAT, but don't figure out 3-occurrence SAT, we will give partial credit for a solution to 5-occurrence SAT.

[25] If not, then there is some j such that $y_{i,j} = \texttt{false}$ and $y_{i,j+1} = \texttt{true}$ (where we take $j + 1 \mod k_i$); for this index we have $y_{i,j} \vee \bar{y}_{i,j+1} = \texttt{false}$.

instead add the "cyclic" equality gadget above to force the $y_{i,j}$'s to be equal, the total number of occurrences for each variable is reduced to 3. Essentially the same iff argument as above shows that this transformation preserves satisfiability.

Rubric.
5 pts. For transforming a SAT to a 3-occurrence SAT formula while preserving satisfiability and total size up to polynomial factors.
-2 pts. For transforming to 5-occurrence SAT instead of 3-occurrence SAT.
1 pts. For indicating how a black box for 3-occurrence-SAT is used to give an answer to the original SAT problem.
4 pts. Proof of correctness.
- Two points for each direction.

Exercise 10.6. Kenjaku has initiated the *Culling Games*, a brutal competition designed to enforce his twisted vision of survival of the strongest. In the game, there are m sorcerers and n "cursed spirits", where n is even. Kenjaku assigns a set of spirits to eliminate each sorcerer. A spirit may be assigned to multiple sorcerers.

Each sorcerer can only defend themselves against a limited number of spirits. (This number may vary by sorcerer.) The cunning Kenjaku strategically assigned exactly one more spirit to each sorcerer than they can handle.

As Satoru Gojo's successor, your goal is to save the overwhelmed sorcerers. To this end, you've partitioned the spirits into $n/2$ disjoint pairs, and you can destroy one spirit out of every pair. Your task is to choose one spirit from each pair to destroy so that each sorcerer can withstand the rest of the cursed spirits assigned to them.

Below are four different variations of the Culling Games varying in certain details. The input always consists of a list S_i of spirits assigned to each sorcerer i, where the length of a list is always one more than the sorcerer can handle, and the partition of spirits into $n/2$ pairs (as a list of pairs). For each, determine whether there exists a polynomial-time algorithm, or whether such an algorithm would imply a polynomial-time algorithm for (Boolean) SAT.[26]

1. In the most general setting, each sorcerer can handle some given number spirits, and this number varies by sorcerer. Determine whether there exists a choice of one cursed spirit to destroy from each pair such that every sorcerer can defend themselves against the rest of the spirits.

2. Suppose each sorcerer can fend off at most 1 spirit. (So Kenjaku assigns two spirits to each sorcerer.) Again, determine whether there exists a choice of one cursed spirit to destroy from each pair such that every sorcerer can defend themselves against the rest of the spirits.

[26] I (Kent) have no idea what's going on here, and I have never felt so old. I refer you to the teaching assistants for more background.

3. Suppose again that each sorcerer can fend off at most 1 spirit, but you realized you cannot save all the sorcerers. (E.g., via the previous problem.) Consider the problem of choosing one spirit to destroy from each pair to save as many sorcerers as possible.

4. Suppose again that each sorcerer can handle at most 1 spirit (and therefore has two spirits assigned to attack them), but also needs to fight a spirit to justify their life choices. Determine whether there exists a choice of one spirit to destroy per pair such that every sorcerer has exactly one spirit remaining to fight.

Solution to exercise 10.6.

Solution. These problems are essentially a SAT problem under different names. Clauses correspond to sorcerers, variables correspond to pairs of ghosts representing the two literals. The literals in a clause correspond to ghosts.

For parts 1, 2, and 4: Let $f(x_1, \ldots, x_n)$ be a SAT formula with m clauses. For each clause C_i ($i \in [m]$) we have a sorcerer S_i. For each variable x_j we have two spirits which we call A_j and B_j. A_j corresponds to the (positive) literal x_j and B_j corresponds to the negated literal $\bar{x}_j$. Destroying A_j corresponds to setting $x_j = \text{true}$, and destroying B_j corresponds to setting $x_j = \text{false}$.

For each clause C_i, sorcerer S_i is assigned the spirits corresponding to the literals of C_i. Destroying one spirit assigned to sorcerer S_i corresponds to choosing at least one of the literals satisfying clause C_i.

For part 3, we note that this construction is a bijection. Given an instance of the CullingGames problem with m sorcerers and n spirits, we generate a SAT formula with m clauses and $n/2$ variables. For the jth pair of spirits, we create a variable x_j, and identify one spirit as the literal x_j (or setting $x_j = \texttt{true}$) and the other spirit as the literal $\bar{x}_j$ (or setting $x_j = \texttt{false}$.) For the ith sorcerer, we create a clause C_i containing all the literals corresponding to spirits assigned to C_i. Observe that these two constructions – from CNF-SAT to Culling Games, and from Culling Games to CNF-SAT – are inverse to one another. Clauses with k literals are in bijection with sorcerers with k spirits assigned to them.

We claim that for corresponding instances of SAT and Culling Games, there is a feasible solution for one instance iff there is a feasible solution for the other.

First, suppose you have a satisfying assignment to a SAT instance. For each variable x_j, if $x_j = \text{true}$, destroy spirit A_j; if $x_j = \text{false}$, destroy spirit B_j. Since the assignment is satisfying, each clause C_i has at least one true literal, so sorcerer S_i has at least one of its assigned spirits destroyed and can defend against the rest.

Conversely, suppose you have a valid choice of spirit destructions in the Culling Games instance. For each pair, exactly one of A_j or B_j is destroyed; set $x_j = \text{true}$ if A_j is destroyed and $x_j = \text{false}$ if B_j is destroyed. Since every sorcerer survives, at least one spirit assigned to each sorcerer is destroyed. For each sorcerer, a chosen spirit maps to a literal in the clause set to true, giving a satisfying assignment overall.

More generally, the same argument shows that one can satisfying k clauses iff one can save k sorcerers, for all k.

1. In the general setting, lists/clauses may have arbitrary size, so this corresponds to general SAT. That is, given an instance of SAT with m clauses and $nvariables$, one can create an instance of Culling games with m sorcerers and $2n$ spirits as described above. A polynomial

algorithm for this problem, applied to this instance, gives a polynomial time algorithm solving SAT.

2. Each sorcerer has exactly 2 spirits assigned, so this maps to 2-SAT. More formally, given such an instance of Culling Games, we create a 2-SAT formula as described above. We apply the $O(m+n)$-time algorithm for 2-SAT in class. If the algorithm returns a satisfying assignment, we returning the corresponding selection of spirits; if the algorithm says its unsatisfiable, we return that the sorcerer's cannot all be saved. Correctness follows from the proofs above.

3. Each sorcerer has exactly 2 spirits, so this maps to MAX-2-SAT. Given an instance of MAX-2-SAT, defined by a 2-SAT formula $f(x_1, \ldots, x_n)$ with n variables and m clauses, we create an instance of Culling Games with m sorcerers and $2n$ spirits as described above. By the argument above, one can satisfy k clauses iff one can save k sorcerers. Thus a polynomial-time algorithm for the optimization version of Culling Games gives a polynomial-time algorithm for MAX-2-SAT, which in turn gives a polynomial-time algorithm for SAT.

4. Each sorcerer has exactly 2 spirits and must have *exactly one* remaining. This is essentially 2-XOR-SAT. Sorcerer S_i with spirits corresponding to literals a and b survives iff $a \oplus b = 1$. Each such XOR constraint $a \oplus b = 1$ is equivalent to the conjunction of two 2-SAT clauses: $(a \vee b) \wedge (\bar{a} \vee \bar{b})$. Thus, we convert the Culling games instance to 2-SAT, and apply the $O(m+n)$-time 2-SAT algorithm from class.

   This reduces the problem to a 2-SAT instance, which is solvable in polynomial time.

Rubric. There is a lot of redundancy across subproblems, for the construction and for the iff proofs. We will generally be flexible about reusing descriptions, so to speak. There are still some important points to fill out, though: when using 2-SAT to solve parts 2 and 4, you are mapping sorcerers and spirits to clauses and variables, while parts 1 and 3 go in the other direction, so to speak. For part 3, you need to observe that the construction preserves the number of satisfied clauses. (The sample solution analyzed the construction in both directions, once, and then leveraged the proven properties in individual problems.)

2.5 pts. Part 1.
- **.5 pts.** Any connection to general SAT
- **.5 pts.** Map clauses to sorcerers.
- **.5 pts.** Map variables to ghosts.
- **1 pt.** 1 Both directions of iff proof.
  - **.5 pts.** If formula is satisfiable, then there is a culling games solution.
  - **.5 pts.** If there is a culling games solution, then the SAT formula is satisfiable.

2.5 pts. Part 2.
- **.5 pts.** Any connection to 2-SAT
- **.5 pts.** Map sorcerers to clauses.
- **.5 pts.** Map ghosts to variables.
- **1 pt.** .5 Both directions of iff proof.
  - **.25 pts.** If 2-SAT formula is satisfiable, then there is a culling games solution.
  - **.25 pts.** If there is a culling games solution, then the 2-SAT SAT formula is satisfiable.
- **.5 pts.** Running time analysis.

2.5 pts. Part 3.
- **.5 pts.** Any connection to Max 2-SAT
- **.5 pts.** Map clauses to sorcerers.
- **.5 pts.** Map variables to ghosts.
- **1 pt.** Prove # sorcerers saved = # clauses satisfied.

2.5 pts. Part 4.
- **.5 pts.** Any connection to XOR-SAT
- **.25 pts.** Map sorcerers to clauses.
- **.25 pts.** Map ghosts to variuables.
- **.25 pts.** Identify the sorcerer's requirements with XOR-SAT.
- **.25 pts.** Write XOR as 2-SAT formulas.
  - **.25 pts.** If the 2-SAT formula is satisfiable, then there is a culling games solution.
  - **.25 pts.** If there is a culling games solution, then the 2-SAT formula is satisfiable.

F.7 Homework 5

Exercise 11.5. Let $x_1, \ldots, x_n \in \mathbb{N}$. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.[27]

1. The *partition problem* asks if one can partition $x_1, \ldots, x_n$ into two parts such that the sums of each part are equal.

2. The 3-*partition problem* asks if one can partition $x_1, \ldots, x_n$ into 3 parts such that the sums of each part are all equal.

3. The *any-k-partition problem* asks one can partition $x_1, \ldots, x_n$ into k parts, for any integer $k \geq 2$, such that the sums of each part are all equal.

4. The *almost-partition* problem asks if one can partition $x_1, \ldots, x_n$ into two parts such that the two sums of each part differ by at most 1.

5. [28]Let n be even. The *perfect partition problem* asks if one can partition $x_1, \ldots, x_n$ into two parts such that

   (a) Each part has the same sum.

   (b) Each part contains the same number of x_i's.

Solution to exercise 11.5.

Problem 1: Partition. We show that a polynomial time algorithm for the partition problem implies a polynomial time algorithm for SAT by a reduction from subset sum.

Suppose we have an instance of subset sum defined by $x_1, \ldots, x_n \in \mathbb{N}$ and $T \in \mathbb{N}$. Let $S = \sum_{i=1}^{n} x_i$ be the total sum of the x_i's. Note that there is a subset summing to T iff there is a subset summing to $S - T$. Replacing T with $S - T$ if necessary, we may assume that $T \geq S/2$.

Consider the partition problem with $n+1$ numbers $x_1, \ldots, x_n, 2T - S$. The total sum is $2T$. We claim that there is an equal-sum partition of $x_1, \ldots, x_n, 2T - S$ iff there is a subset of $x_1, \ldots, x_n$ summing to T.

Suppose there is a partition of $x_1, \ldots, x_n, 2T - S$ into two parts so that both partition have an equal sum. Since the total is $2T$, this means that both parts sum to T. In particular, the part excluding $2T - S$ is a subset of $x_1, \ldots, x_n$ summing to T.

Conversely, suppose there is a subset $x_{i_1}, \ldots, x_{i_k}$ of $x_1, \ldots, x_n$ summing to T. Consider the partition $x_1, \ldots, x_n, 2T - S$ where one part consists of $x_{i_1}, \ldots, x_{i_k}$ and the other part has all remaining numbers. Both parts sum to T, so there is an equal-sum partition.

Remark F.10. Be careful: it matters whether or not T is bigger than $S/2$.

[27]You can use the solution of one subproblem to solve another, as long as there's no circular dependencies overall.
[28]IMO, this one is the trickiest.

Problem 2: 3-partition. We show that a polynomial time algorithm 3-partition problem implies a polynomial time algorithm for SAT by a reduction from the previous partition problem.

Suppose we have an instance of the partition problem defined by $x_1, \dots, x_n$. Let $S = \sum_{i=1}^n x_i$ be the total sum. If S is odd then there is no partition and we answer no. Otherwise, suppose S is even. Consider the three-partition problem defined by the $n+1$ numbers $x_1, \dots, x_n, S/2$. We claim that there is an equal-sum (2-)partition for $x_1, \dots, x_n$ iff there is an equal-sum 3-partition of $x_1, \dots, x_n, S/2$.

If we can partition $x_1, \dots, x_n$ into two parts with equal sums, then the common sum is $S/2$. We obtain a three-partition $x_1, \dots, x_n$ by taking the partition of $x_1, \dots, x_n$ and adding $\{S/2\}$ as the third part.

Conversely, suppose we can partition $x_1, \dots, x_n, S/2$ into three parts with equal sum. Then the common sum is $S/2$. Since each of the x_i's are positive, one of the parts must be $\{S/2\}$. The remaining two parts partition $x_1, \dots, x_n$ into two parts of equal sum.

Problem 3: Any k-partition. We show that a polynomial time algorithm for any-k-partition implies a polynomial time algorithm for SAT by a reduction from the partition problem.

Suppose we have an instance of the partition problem defined by $x_1, \dots, x_n$. Let $S = \sum_{i=1}^n x_i$ be the total sum. If S is odd then there is no partition and we answer no. Otherwise, suppose S is even.

Consider the instance of any-k-partition defined by $x_1, \dots, x_n, 2S, 2S$. We claim that the given 2-partition problem has a feasible solution iff this any-k-partition problem has a feasible solution.

First, suppose there is a 2-partition of $x_1, \dots, x_n$ where the two parts have equal sums. If we add one copy of $2S$ to each part, then we obtain a partition of $x_1, \dots, x_n, 2S, 2S$ with equal sums.

Conversely, suppose there is a feasible solution to the any-k-partition problem. That is, there is a partition of $x_1, \dots, x_n$ into k parts, for some $k \geq 2$, such that each part has the same sum.

We first observe that this sum is at least $2S$ since $2S$ is one of the input numbers. If $k \geq 3$, then the sum is at most $(5/3)S$, a contradiction. So we must have $k = 2$.

Thus we have a 2-partition of $x_1, \dots, x_n, 2S, 2S$ where the two parts have equal sum, $5S/2$. This implies that each part must have one copy of $2S$. Then each part, excluding its copy of $2S$, sums to $S/2$. That is, removing $2S$ from each part gives a 2-partition of $x_1, \dots, x_n$ with equal sums.

Problem 4: Almost-partition. We show that a polynomial time algorithm for almost-partition implies a polynomial time algorithm for SAT by a reduction from the partition problem.

Given an instance of partition defined by $x_1, \dots, x_n$, consider the instance of almost-partition with integers $y_1, \dots, y_n$ where $y_i = 2x_i$ for all i.

If there is an equal-sum partition of the x_i's, then the corresponding partition for the y_i's has equal sum.

Conversely suppose there is an almost-partition of $y_1, \dots, y_n$. That is, there is a partition of $y_1, \dots, y_n$ into two parts so that the sums are within 1. Since all y_i's are even, these sums are also even, in which case the only way to be within 1 is if the sums are equal. The corresponding partition of $x_1, \dots, x_n$ is an equal-sum partition of $x_1, \dots, x_n$.

Problem 5: Perfect-partition. We show that a polynomial time algorithm for perfect-partition implies a polynomial algorithm for SAT by a reduction from the partition problem.

Suppose we have an instance of partition defined by $x_1, \dots, x_n \in \mathbb{N}$. Let $S = \sum_{i=1}^n x_i$ be the total sum.

For each i, let $y_i = 2Sx_i$. Consider the perfect partition instance defined by $x_1, \dots, x_n, y_1, \dots, y_n$. We claim that there is a feasible solution to the partition problem over $x_1, \dots, x_n$ iff there is a feasible solution to the perfect partition problem over $x_1, \dots, x_n, y_1, \dots, y_n$.

Suppose there is a feasible solution to the partition problem over $x_1, \dots, x_n$. Let $A \subseteq [n]$ be the indices of one part and $B \subseteq [n]$ be the indices of the other part.

Consider the partition of $x_1, \dots, x_n, y_1, \dots, y_n$ where one part consists of

$$\{x_i : i \in A\} \sqcup \{y_i : i \in B\},$$

and the other consists of

$$\{x_i : i \in B\} \sqcup \{y_i : i \in A\}.$$

These two parts have the same sum (namely, $(2S+1)S/2$) and the same cardinalities, as desired.

Conversely, suppose there is a feasible solution to the perfect partition problem over $x_1, \dots, x_n, y_1, \dots, y_n$. Let $A \subseteq [n]$ be the indices of the x_i's indices of one part, and let $B = [n] \setminus A$ be the indices of the x_i's of the other part.

Since the 2 parts have equal sum, they also have an equal sum modulo $2S$. Since every $y_i = 0 \mod 2S$, since implies that

$$\sum_{i \in A} x_i = \sum_{i \in B} x_i.$$

That is, (A, B) define a 2-partition with equal sum.

Remark F.11. $\mathbb{N}$, which stands for the natural numbers, excludes 0.

> *Rubric.* 2 points for each problem.
> **1 pt.** Correct algorithm/construction.
> **1 pt.** Proof
> **.5 pts.** For each direction of an if and only if.

Exercise 12.10. Let $G = (V, E)$ be a directed graph with m edges and n vertices, where each vertex $v \in V$ is given an integer label $\ell(v) \in \mathbb{N}$. The goal is to find the length of the longest path[29] in G where the labels of the vertices are (strictly) increasing.

1. Suppose G is a DAG. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. Consider now the problem for general graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

3. Suppose instead we ask for the length of the longest path in G where [30] is weakly increasing if $x_1 \le x_2 \le \cdots \le x_k$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 12.10.

[29] Recall that a path is a walk that does not repeat vertices.

[30] A sequence G is a general graph and the labels of the vertices are weakly increasing.$x_1, \dots, x_k$

Problem 1: Longest increasing path in a DAG. There are (at least) two approaches to this problem. One is to drop from G all "bad" edges (u, v) where $\ell(u) \geq \ell(v)$. Such an edge is never in any of the paths we are interested in. Conversely, after dropping these edges from G, any path in G has strictly increasing labels.

The overall algorithm, then, is to drop these edges and run the longest path algorithm from the lecture notes.

Alternatively, we can modify our longest path dynamic program to avoid the bad edges. Here I will only list the spec and implementation; as the rest is very similar to the longest path problem.

1. Recursive spec / induction hypothesis. For $v \in V$, we define

> `LIP(`v`)` = the length of the longest path starting from v with strictly increasing labels.

2. Recursive implementation.

> `LIP(`v`)`
>
> 1. Return 0 if v is a sink or if $\ell(v) \geq \ell(u)$ for all edges (v, u) leaving v.
> 2. Otherwise return the maximum of
>
> $$1 + \texttt{LIP}(w)$$
>
> over all edges (v, w) with $\ell(w) > \ell(v)$.

3. Solving the original problem. We return the packing of `LIP(`v`)` over all $v \in V$.

4. Running time with caching / dynamic programming. We apply dynamic programming to the recursive spec above. The subproblem for a vertex v takes time proportional to the out-degree of v. Summing these all together gives a $O(m + n)$ running time.

5. Proof. We prove that the pseudocode for `LIP(`v`)` implements the recursive spec by induction on the topological order. The base case is the last vertex v in topological order, which is necessarily a sink. The `LIP(`v`)` is 0, since there is no edge to take starting from v.

Now let v be a vertex, and assume by induction that for all vertices w that come after v in topological order, `LIP(`w`)` returns the length of the longest path starting from w with strictly increasing labels.

If v does not have any edges to vertices with higher labels, then the algorithm returns the correct solution of 0. Otherwise the algorithm returns the maximum of $1 + \texttt{LIP}(w)$ over all out-neighbors w of v. For all such w, the inductive hypothesis implies that `LIP(`w`)` returns the length of the longest increasing-label path from w. In particular, $1 + \texttt{LIP}(w)$ always represents the length of some path from v. So `LIP(`v`)` returns at least the length of the longest increasing-label path from v. Moreover, for the true longest increasing-label path from v, there is some outneighbor w over v that is the second vertex on the path, and this w is among the choices in the maximum. So `LIP(`v`)` returns the exact length of the longest increasing-label path from v.

Rubric.

1 pt. Recursive spec.

1 pt. Recursive implementation.

.5 pts. Solving original problem.

.5 pts. Running time analysis.

1 pt. Proof.

Remark F.12. One can also try to the use the longest increasing path algorithm as a black box. (In class and in these notes we only really do the unweighted version, so maybe a comment at least addressing this gap would be nice...)

Problem 2: Longest increasing path in general graphs. The difference between part 1 and part 2 is that G may have cycles. However, if we drop all "bad" edges from G, then there are no cycles in G and G is an acyclic graph, to which we can apply our solution from part 1. The running time is the same — $O(m+n)$.

Rubric.

2 pts. Correct, $O(m+n)$ algorithm.

.5 pts. Running time analysis.

.5 pts. Justification.

Problem 3: Longest weakly increasing path. A polynomial time algorithm for the length of the longest path with weakly increasing labels implies a polynomial time algorithm for SAT.

Recall that a polynomial time algorithm deciding if a G graph G has a Hamiltonian path implies a polynomial time algorithm for SAT.

Given an instance of Hamiltonian path defined by a directed graph G, we assign vertex the same label; say, 0. Then every path has weakly increasing labels. Consequently G has a Hamiltonian path iff it has a path with weakly increasing vertex labels of length $n-1$.

Given a black box that computes the length of the longest path with weakly increasing vertex labels, we submit G with vertex labels 0. If the black box returns $n-1$, we declare that G has a Hamiltonian path; otherwise we declare that G does not have a Hamiltonian path.

Rubric.

2 pts. Description of a correct reduction.

1 pt. Proof of correctness.

Exercise F.1. Recall that the Economist recently reported[31] that the number of toll roads in America is increasing rapidly. It used to be hard enough to find the shortest path. Now you have to do it within a budget.

Right now, drivers pay to access 6,300 miles of Americas 160,000 miles (or so) of highway, but that number is going up. Here in Indiana, in June, the state passed a law giving the state authority to raise tolls on all its existing interstate highways. The Economist says that this trend is broadly due to a parallel trend of decreasing taxes on gasoline, which used to pay a larger share of the cost for maintaining roads and highways.

Let $G = (V, E)$ be a directed graph with nonnegative integer edge lengths $\ell : E \to \mathbb{Z}_{\geq 0}$ and *nonnegative integer edge costs* $\texttt{cost} : E \to \mathbb{Z}_{\geq 0}$, which model toll roads. We define the *cost* of a walk in G as the sum of edge costs (with repetition) along that walk. Let $k \in \mathbb{N}$ be a given budget. Consider the problem of computing the length of the shortest (s, t)-walk that costs at most k. (Unlike the midterm, we do not assume that k is polynomially bounded by m and n.) For this problem, either design and analyze a polynomial time algorithm, or prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Solution to exercise F.1. We prove that this problem gives a polynomial time algorithm for Partition. Suppose we have an instance of partition with n numbers $x_1, \dots, x_n$. We can assume that their sum is even; let T be half their sum.

We create $n + 1$ vertices $v_0, \dots, v_n$. For $i = 1, \dots, n$, we add two arcs from v_{i-1} to v_i. The first arc has length x_i and cost 0. The second arc has length 0 and cost x_i. We set a budget of T, and see if there is a walk from v_0 to v_n with cost at most T and distance at most T. More specifically, we compute the length of the shortest $v_0 \rightsquigarrow v_n$ walk with budget at most T. We return `true` for the Partition problem if this length is at most T, and `false` otherwise.

At a high-level, for each $i \in [n]$, taking the first copy of (v_{i-1}, v_i) corresponds to adding x_i to the first part of our partition, and taking the second copy corresponds to adding x_i to the second part.

To prove our reduction is correct, we first prove the following.

Claim. For any $v_0 \rightsquigarrow v_n$ walk W, $\bar{\ell}(W) + \bar{c}(W) = 2T$, where $\bar{\ell}(\cdots)$ and $\bar{c}(\cdots)$ denote the sum length and sum cost, respectively.

To this end, observe that every such walk W is a sequence of choices between choosing one of the two arcs between each consecutive pair of vertices. For a consecutive pair (v_{i-1}, v_i), taking the first arc adds x_i to the length, and taking the second arc adds x_i to the cost. Either way, we add x_i to $\bar{\ell}(W) + \bar{c}(W)$. The sum of x_i's is $2T$.

Now we prove the main claim:

Claim. There is a partition of $x_1, \dots, x_n$ iff there is a $v_0 \rightsquigarrow v_n$ walk of length at most T and cost at most T.

First, suppose there is a partition of $x_1, \dots, x_n$ into equal sum parts. Let $S \subseteq [n]$ be the indices of one part and let $\bar{S} = [n] \setminus S$ be the complement. Consider the walk W that takes the first arc (length x_i, cost 0) for each $i \in S$, and the second arc (length 0, cost x_i) for each $i \in \bar{S}$. Then $\bar{\ell}(W) = \sum_{i \in S} x_i = T$ and $\bar{c}(W) = \sum_{i \in \bar{S}} x_i = T$. So W has length at most T and cost at most T.

Conversely, suppose there is a $v_0 \rightsquigarrow v_n$ walk W with $\bar{\ell}(W) \leq T$ and $\bar{c}(W) \leq T$. By the previous claim, $\bar{\ell}(W) + \bar{c}(W) = 2T$. Combined with both inequalities, we get $\bar{\ell}(W) = T$ and $\bar{c}(W) = T$. Let

[31] *Toll roads are spreading in America*, December 18, 2025. https://www.economist.com/united-states/2025/12/18/toll-roads-are-spreading-in-america

S be the set of indices i for which W takes the first arc from v_{i-1} to v_i. Then $\sum_{i \in S} x_i = \bar{\ell}(W) = T$ and $\sum_{i \in \bar{S}} x_i = \bar{c}(W) = T$, giving a valid partition.

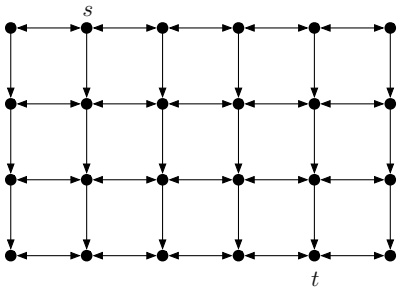
Rubric. Of course there may be multiple ways to prove the problem is hard. This rubric is tailored to the approach (based on partition) above.
3 pts. Anything indicating NP-Hard.
3 pts. Transforming partition to toll road problem.
- **1 pt.** Map each number to two different edges (corresponding to taking or not taking the number).
- **1 pt.** Appropriately assigning edge lengths.
- **1 pt.** Appropriately assigned toll costs.

4 pts. Proof
- **2 pts.** Mapping in-budget (s,t)-walk of length T to partition solution.
  - **1 pt.** Map edges taken to a set of numbers S.
  - **1 pt.** Rigorously explaining why S has sum T.
- **2 pts.** Mapping partition solution to in-budget (s,t)-walk of length T.
  - **1 pt.** Map a set of numbers S to an (s,t)-walk.
  - **.5 pts.** Rigorously explain why the walk has length T.
  - **.5 pts.** Rigorously explain why the walk has cost at most T.

F.8 Homework 6

Exercise 12.22. Let $G = (V, E)$ be a directed graph with real-valued (and possibly negative) edge lengths $\ell : E \to \mathbb{R}$, and $s, t \in V$. We are interest in the length of the shortest (s,t)-path for a special class of "downwards grid graphs" G.

A "downward grid graph" G is parametrized by a height $m \in \mathbb{N}$ and a width $n \in \mathbb{N}$. All the vertices are arranged in an $m \times n$ grid. We let (i,j) denote the vertex in row i and column j. There are edges between every adjacent pair of edges except going up. More specifically, for $i \in [m]$ and $j \in [n]$, there is an arc



- (Down) to $(i+1, j)$ when $i < m$;
- (Right) to $(i, j+1)$ when $j < n$; and
- (Left) to $(i, j-1)$, when $j > 1$.

In particular, there are no edges going "up" from (i,j) to $(i-1,j)$. Note that G has $O(mn)$ vertices and $O(mn)$ edges.

Consider the problem of computing the length of the shortest (s,t)-path in a downward grid graph. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 12.22. One can compute this as either a shortest path problem in an auxiliary DAG, or with dynamic programming. We first describe the shortest path approach.

We construct an auxiliary graph H as follows.

1. For each vertex (i, j), we create two (i, j, LEFT) and (i, j, RIGHT).
2. For all vertices (i, j) with $i < m$, and any two direction $X, Y \in \{\text{LEFT}, \text{RIGHT}\}$, we create arcs from (i, j, X) to $(i + 1, j, Y)$ of length $\ell((i, j), (i + 1, j))$.
3. For all vertices (i, j) with $j < 1$, we create an arc from (i, j, RIGHT) to $(i, j + 1, \text{RIGHT})$ of length $\ell((i, j), (i, j + 1))$.

We add a supersource $s^\star$ and a supersink $t^\star$. Assuming $s = (i_s, j_s)$, we add arcs $(s^\star, (i_s, j_s, \text{LEFT}))$ and $(s^\star, (i_s, j_s, \text{RIGHT}))$ of length 0. Assuming $t = (i_t, j_t)$, we add arcs $((i_t, j_t, \text{LEFT}), t^\star)$ and $((i_t, j_t, \text{RIGHT}), t^\star)$ of length 0.

Observe that any vertex (i, j, RIGHT) can only reach vertices of the form (i, j', RIGHT) with $j' \geq j$ and (i', j', X) (for any $X \in \{\text{LEFT}, \text{RIGHT}\}$) with $i' > i$. Similarly any vertex (i, j, LEFT) can only reach vertices of the form (i, j', LEFT) with $j' \leq j$ and (i', j', X) (for any $X \in \{\text{LEFT}, \text{RIGHT}\}$) with $i' > i$. It follows that H is a DAG.

Finally, we compute the length of the shortest path from $s^\star$ to $t^\star$ with the following dynamic program.

1. Recursive spec / induction hypothesis. $\mathsf{SP}(v)$: For $v \in V$, $\mathsf{SP}(v)$ returns the length of the shortest path from v to $t^\star$.

2. Recursive implementation.

$\underline{\mathsf{SP}(v)}$

1. If $v = t^\star$ return 0.
2. Otherwise return the infimum, over all arcs (v, w) leaving v, of $\ell(v, w) + \mathsf{SP}(w)$.

3. Solving the original problem. We return $\mathsf{SP}(s^\star)$.

4. Running time with caching / dynamic programming. We apply dynamic programming to the recursive algorithm above. We have $O(mn)$ subproblems because there are $O(mn)$ vertices, and each subproblem $\mathsf{SP}(v)$ takes time constant time, so the total running time is $O(mn)$.

Now here is a purely dynamic programming solution. Let $s = (i_s, j_s)$ and $t = (i_t, j_t)$.

1. Recursive spec / induction hypothesis. $\mathsf{SP}(i, j, X)$: For $i \in [m]$, $j \in [n]$ and $X \in \{\text{LEFT}, \text{RIGHT}\}$, returns the length of the shortest path from (i, j) to t where the first edge (if any) either goes down or in the direction X.

2. Recursive implementation.

$\underline{\mathsf{SP}(i, j, X)}$

1. If $(i, j) = t$ return 0.
2. $A = +\infty$.
3. If $X = \text{RIGHT}$ and $j < n$, set $A = \mathsf{SP}(i, j + 1, X)$
4. If $X = \text{LEFT}$ and $j > 1$, set $\mathsf{SP}(i, j - 1, X)$.
5. If $i > m$ set $A = \min\{A, \mathsf{SP}(i + 1, j, \text{LEFT}), \mathsf{SP}(i + 1, j, \text{RIGHT})\}$.
6. Return A

3. Solving the original problem. We return the minimum of $\mathsf{SP}(i_s, j_s, \text{LEFT})$ and $\mathsf{SP}(i_s, j_s, \text{RIGHT})$.

4. Running time with caching / dynamic programming. We apply dynamic programming to the recursive algorithm above. We have $O(mn)$ subproblems because there are $O(mn)$ vertices, and each subproblem takes constant time, so the total running time is $O(mn)$.

Rubric. Auxiliary-graph based solutions
1 pt. Polynomial time
5 pts. Auxiliary graph
- **2 pts.** Vertices encode directions.
- **1 pt.** Arcs going left or right depending on the "direction" of the node, except at the boundary.
- **1 pt.** Arcs going down to both LEFT/RIGHTnodes except for the last row.
- **1 pt.** Super-source, super-sink.

1 pt. Explicitly explain why the auxiliary graph is a DAG.
3 pts. DP in DAG
- **1 pt.** Recursive spec
- **1 pt.** Recursive implementation
  - **.5 pts.** Base case.
  - **.5 pts.** Recursive calls.
- **.5 pts.** Running time.
- **.5 pts.** Driver

DP solutions.
1 pt. Polytime.
4.5 pts. Recursuve spec
- **1 pt.** Objective: Shortest distance to t
- **1.5 pts.** Current vertex of the subproblem.
- **2 pts.** Direction (left/right) of the subproblem.

2.5 pts. Recursive implementation
- **.5 pts.** Base case.
- **.5 pts.** Try LEFT/RIGHTbased on the current direction.
- **.5 pts.** Maintain direction when going LEFTor RIGHT.
- **.5 pts.** Try going down.
- **.5 pts.** Try both directions when going down.

1 pt. Driver (start from s with either direction).
1 pt. Running time ($O(mn)$ with dynamic programming.)

Exercise 13.4. Let $G = (V, E)$ be an undirected graph. We say that a set of vertices is *almost independent* if each vertex $v \in S$ has at most one neighbor in S.[32] Consider the problem of computing the maximum cardinality of any almost independent set of vertices. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 13.4. We prove that the problem is NP-Hard by reduction from independent set.

Suppose we have an instance of independent set defined by an undirected graph $G = (V, E)$.

[32]Two vertices u and v are neighbors if they are connected by an edge.

Let $G' = (V \cup V', E \cup E')$ be the graph extending G by adding, for each vertex $v \in V$, and auxiliary vertex v' and an edge from v' to v. We compute the size of the maximum almost independent set in G', and return this quantity minus n.

Let k denote the maximum size of an independent set in G, and ℓ the maximum size of an independent set in G'. We claim that $\ell = n + k$.

First we show that $\ell \geq n + k$. Let $S \subseteq V$ be an independent set of size k in G. Then $S \cup V'$ is an almost independent set of size $n + k$ in G'.

Let $S \cup S'$ be an almost independent set of size ℓ in G', where $S \subseteq V$ and $S' \subseteq V'$.

For any vertex $v \in V$ such that $v \in S$ and $v' \notin S'$, we can replace v with v' and still have an almost independent set of size ℓ. Applying this to all $v \in V$, we may assume that $S' = V'$.

We claim that since $S' = V'$, S is an independent set. Indeed, each vertex $v \in S$ is already adjacent to $v' \in S'$, so it cannot be adjacent to any other $u \in S$.

Thus S is an independent set. It has size $\ell - n$. Thus $k \geq \ell - n$, as desired.

Exercise 13.11. Can't we all just get along? You want to throw a big party to celebrate finishing midterm 2, but not all of your friends get along. So you resolve to invite as many friends as possible while making sure there are no problems within the party.

Formally, we assume that you have n friends indexed by $1, \ldots, n$. Your input consists of a list L of length m of all pairs of friends who don't get along. The goal is to compute the largest cardinality (of any) subset of friends, $S \subseteq [n]$, that does not contain any pair of friends in the list L.

1. (5 points) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Now suppose there are no "cycles" among the pairs of enemies: that is, there is no sequence of friends $f_0, \ldots, f_{k-1}$, where $k \geq 3$, such that f_i does not get along with $f_{i+1 \mod k}$ for all i. For this setting, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 13.11.

Problem 1. This problem is a disguised instance of independent set. Indeed, given an instance of independent set defined by an undirected graph $G = (V, E)$, we make an instance of Problem 1 where we number the vertices $1, \ldots, n$, and for each edge between vertices numbered i and j, we add $\{i, j\}$ to L.

We claim that a set of vertices S is an independent set iff the corresponding group of friends contains no enemies. On one hand, if S is independent, then consider the party of friends corresponding to the vertices in S. There are no enemies in the part because that would imply an edge, whereas S is independent.

Conversely, suppose we have a feasible party of friends X. Let S be the set of vertices corresponding to X. Then S is independent, because an edge would correspond to two friends who don't get along, but everyone in X gets along.

Thus the largest cardinality subset of friends with no enemies corresponds to the maximum cardinality independent set.

Problem 2. Above we argued that the problem is just a maximum cardinality independent set problem. In this setting, the corresponding graph G is a forest, and we can call the black box algorithm computing the maximum independent set in a tree, for each tree in the forest. We construct the graph $G = (V, E)$ where V is the set of indices $1, \dots, n$ and we have an edge for every pair of friends that don't get along. (Technically we did maximum weight independent set in a tree, so we can assign weight 1 to every vertex.) We now apply the maximum (weight) independent set algorithm as a black box to each tree in G, and combine the solutions. The running time is $O(m + n)$.

F.9 Homework 7

Exercise 16.6. Let $G = (V, E)$ be an undirected graph with distinct nonnegative edge weights $w : E \to \mathbb{R}$. For a spanning tree T, we say that the *bottleneck weight* of T is the maximum weight edge in T, $\max_{e \in T} w(e)$.

1. Prove that the MST is also a minimum bottleneck weight spanning tree of G.

2. Design and analyze a $O(m + n)$-time algorithm for computing a minimum bottleneck weight spanning tree of G. (This is faster than any of our algorithms for MST.)[33]

Solution to exercise 16.6. The first step is to identify the minimum bottleneck weight, so to speak. This is the minimum value of x such that the set of edges with weight $\leq x$ connects all of V.

Once we know the minimum bottleneck weight x, then we can run BFS/DFS among the edges with weight $\leq x$, to get a minimum bottleneck weight tree.

Below we focus on identifying the minimum bottleneck weight x.

Given a connected undirected graph $G = (V, E)$ with distinct edge weights $w : E \to \mathbb{R}$, we define

`bottleneck(`$G = (V, E)$`)` = the bottleneck weight of G.

We require the input graph G to be connected, which ensures that $m \geq n - 1$.

`bottleneck(`$G = (V, E)$`)`

1. If $|E| \leq O(1)$ then brute force.
2. Let x be the median weight of E. Let $E_0 = \{e \in E : w(e) \leq x\}$ and $G_0 = (V, E_0)$. // $O(m)$
3. If G_0 is connected: // $O(m)$ *with BFS/DFS*
   A. Return `bottleneck(`G_0`)` // $O(T(m/2))$.

/* *Otherwise* G_0 *is not connected, and the bottleneck weight is* $> x$. */

4. Let $G_1 = (V_1, E_1)$ be the graph obtained by contracting all the edges in E_0. // $O(m)$.

/* G_1 *is a connected graph with distinct edge weights and at most* $m/2$ *edges.* */

5. Return `bottleneck(`G_1`)` // $O(T(m/2))$.

[33] Here's step 1: compute the median edge weight in $O(m)$ time.

By guaranteeing the input graph is connected, we ensure that $n \leq O(m)$. Then the running time can be modeled recursively by

$$T(m) = T(m/2) + O(m),$$

which is solved by $T(m) = O(m)$.

Exercise 16.10. Let $G = (V, E)$ be an undirected graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. The *traveling salesman problem* (TSP) asks for the minimum-length tour in G. (A tour starts and ends at the same vertex, and visits each vertex at least once.)

The traveling salesman problem is extremely useful as a prototypical planning and routing problem (broadly construed). It has inspired a number of techniques both in theory and in practice, and also captured the popular imagination. See https://www.math.uwaterloo.ca/tsp/history/index.html for more on the history of this problem.

1. Prove that TSP is NP-Hard (in undirected graphs).[34]

2. Let $\text{OPT}_{\mathsf{TSP}}$ be the minimum TSP value and let $\text{OPT}_{\mathsf{MST}}$ be the weight of the MST with respect to ℓ. Prove that $\text{OPT}_{\mathsf{MST}} \leq (1 - 1/n)\,\text{OPT}_{\mathsf{TSP}}$.[35]

3. Given TSP is NP-Hard, we might try to get a polynomial time *approximation algorithm* instead. For $\alpha > 1$, an *α-approximation* for TSP is a tour of cost at most $\alpha\,\text{OPT}_{\mathsf{TSP}}$. Unlike exact algorithms, a polynomial time α-approximation algorithm does not immediately imply an exact polynomial time algorithm for SAT.

   Design and analyze a polynomial time 2-approximation algorithm for TSP.[36]

Solution to exercise 16.10.

Part 1. We first prove that for an unweighted graph $G = (V, E)$, there is a TSP of length n iff there is a Hamiltonian cycle. So it suffices to show that Hamiltonian cycle is NP-Hard in undirected graphs.

We prove that a polynomial time algorithm for undirected Hamiltonian cycle implies a polynomial time algorithm for undirected Hamiltonian path (which was shown to be NP-Hard in lecture).

Let $G = (V, E)$ be an instance of undirected Hamiltonian path. Let H be the graph obtained by G by introducing a new vertex $s^\star$, attached to every vertex in G. Note that H has 1 more vertex and n more edges than G, so it is still polynomial in the size of G. We claim that G has a Hamiltonian path iff H has a Hamiltonian cycle. If so, then a polynomial time algorithm deciding if there is an undirected Hamiltonian cycle, applied to H, gives a polynomial time algorithm deciding if there is an undirected Hamiltonian path in G.

Suppose G has a Hamiltonian path $P = (v_1, v_2, \ldots, v_n)$. Then $(v_1, v_2, \ldots, v_n, s^\star, v_1)$ is a Hamiltonian cycle in H.

Suppose H has a Hamiltonian cycle C. Since $s^\star$ has degree 2 in C, removing $s^\star$ from C gives a Hamiltonian path in G.

[34] You can assume that Hamiltonian path in undirected graphs is NP-Hard. (Part 1 of exercise 12.4; it was briefy covered at the end of Kent's "Longest Path" lecture..)

[35] *Hint:* You may first want to show that $\text{OPT}_{\mathsf{MST}} \leq \text{OPT}_{\mathsf{TSP}}$.

[36] *Hint:* By part 2, the MST has weight less than $\text{OPT}_{\mathsf{TSP}}$. So two copies of the MST have weight...

Part 2. The key observation is that the TSP solution represents a spanning set of edges, hence has weight at least the MST. (Although it may not be a tree.) In fact, removing any edge from the TSP solution is spanning.

Thus, let T be the optimum tour, and let e be the maximum length edge in T. We have $\ell(e) \geq \ell(T)/n$ by choice of e. Meanwhile, $T - e$, as a walk through traversing all the edges, spans G. The MST is the minimum weight spanning set in G, so $\ell(T - e) \geq \mathrm{OPT}_{\mathsf{MST}}$. Altogether, we have

$$\mathrm{OPT}_{\mathsf{MST}} \leq \ell(T - e) \leq (1 - 1/n)\ell(T) = (1 - 1/n)\,\mathrm{OPT}_{\mathsf{TSP}}\,.$$

Part 3. We compute the MST and traverse it through the graph, going in and out of every subtree. This gives a tour that traverses each edge exactly twice (entering and leaving each subtree). The total length is $2\,\mathrm{OPT}_{\mathsf{MST}} \leq 2(1 - 1/n)\,\mathrm{OPT}_{\mathsf{TSP}} < 2\,\mathrm{OPT}_{\mathsf{TSP}}$.

Exercise 17.3. Let $G = (V, E)$ be an undirected graph, and let $a, b, c \in V$ be three distinct vertices. We define an (a, b, c)-*path* as a path from a to c that goes through b. Consider the problem of deciding if there exists an (a, b, c)-path. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 17.3. We reduce finding an (a, b, c)-path to finding 2 vertex disjoint paths. We claim that there is an (a, b, c)-paths iff there is a pair of paths, consisting of one path from b to a and another path from b to c, that are vertex disjoint except at b.

On one hand, suppose there is an (a, b, c)-path. This path splits into a path p_1 from a to b and a path p_2 from b to c, that are vertex-disjoint except at b. Then the path *reversing* p_1 (note that G is undirected) and the path p_2 give the desired pair of paths.

Conversely, suppose we have a pair of paths p_1 from b to a and and p_2 from b to c, that are vertex disjoint except at b. Then the reverse of p_1, followed by p_2, gives an (a, b, c)-path.

Thus it suffices to find paths from b to a and from b to c that are vertex-disjoint except at b. We reduce this to vertex-disjoint paths, where b is a source, and we introduce an auxiliary sink t with directed edges (a, t) and (b, t). There is a 1-1 mapping between pairs of vertex-disjoint (except at b) paths from b to a and b to c in G, and vertex disjoint paths from b to t in the flow network, where we either append or remove the sink edges (a, t) and (b, t).

Since the maximum flow has size ≤ 2, Ford-Fulkerson takes $O(m)$ time to either compute the desired paths, or decide that no such paths exist.

Rubric.
4 pts. Correct algorithm.
1 pt. -1 If the algorithm computes edge-disjoint paths instead of vertex-disjoint paths.
2 pts. Running time.
4 pts. Proof of correctness (2 points for each direction of an if and only if).

F.10 Midterm 2

Exercise 16.12. Let $G = (V, E)$ be an undirected graph with edge weights $w : E \to \mathbb{R}$. Recall that the minimum weight spanning tree can be computed in polynomial time by a number of different algorithms. Suppose we imposed an additional constraint that the spanning tree has maximum (unweighted) degree at most Δ for some given integer $\Delta \in \mathbb{N}$.

1. (5 points) Consider the problem of computing the minimum weight spanning tree with maximum degree at most 2. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Consider the problem of computing the minimum weight spanning tree with maximum degree at most Δ, for $\Delta \geq 3$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 16.12.

Part 1. We prove the problem is NP-hard by reduction from Hamiltonian path.

The key observation is that a spanning tree with maximum degree 2 is exactly a Hamiltonian path. (A tree on n vertices with max degree 2 is a path visiting all n vertices.)

Given an instance of Hamiltonian path defined by an undirected graph $G = (V, E)$, we assign weight 1 to all edges. We then query a black box for minimum weight spanning tree with max degree at most 2. If a degree-2 spanning tree is returned, we declare that G has a Hamiltonian path. Otherwise we declare that G does not have a Hamiltonian path.

We claim that G has a Hamiltonian path iff G has a spanning tree with maximum degree at most 2.

Suppose G has a Hamiltonian path $v_1, v_2, \ldots, v_n$. The edges of this path form a spanning tree with maximum degree 2.

Conversely, suppose G has a spanning tree T with maximum degree at most 2. Since T is a tree on n vertices where every vertex has degree at most 2, T is a path visiting all n vertices, i.e., a Hamiltonian path.

Part 2. We prove the problem is NP-hard by reduction from Hamiltonian path.

Given an undirected graph $G = (V, E)$, we construct an augmented graph G' as follows. For each vertex $v \in V$, we add $(\Delta - 2)$ new *auxiliary leaf* vertices, each connected only to v. All edges (original and auxiliary) are given weight 1. We then query a black box for the minimum weight spanning tree of G' with max degree at most Δ.

We claim that G has a Hamiltonian path iff G' has a spanning tree with max degree at most Δ. (Note that every auxiliary leaf has degree 1 in G', so its unique edge must be in any spanning tree.)

Suppose G has a Hamiltonian path P. We extend P to a spanning tree T' of G' by adding all auxiliary leaf edges. Each original vertex v has degree at most 2 in P plus $(\Delta - 2)$ auxiliary leaf edges, for a total degree of at most Δ. Each auxiliary leaf vertex has degree 1. So T' is a spanning tree of G' with max degree at most Δ.

Conversely, suppose G' has a spanning tree T' with max degree at most Δ. As noted, each auxiliary leaf's unique edge must be in T', accounting for $(\Delta - 2)$ of the degree budget at each original vertex v and leaving at most $\Delta - (\Delta - 2) = 2$ edges in T' between original vertices. Let $P = T' \cap E$ be the restriction of T' to the original edges. P has max degree at most 2. Moreover, P

must be connected: if P had two components, then T' would also be disconnected (since auxiliary leaves don't bridge original vertices), contradicting that T' is a spanning tree. So P is a spanning tree of G with max degree at most 2, i.e., a Hamiltonian path (by the same observation as part 1).

Rubric.

Part 1 (5 points).
1 pt. Identify the problem as NP-hard.
2 pts. Key insight: a degree-2 spanning tree is a Hamiltonian path.
1 pt. Reduction construction (assign uniform weights, query the black box, return the answer).
1 pt. Correctness:
- **.5 pts.** Forward direction (Hamiltonian path is a degree-2 spanning tree).
- **.5 pts.** Reverse direction (degree-2 spanning tree is a Hamiltonian path).

Part 2 (5 points).
1 pt. Identify the problem as NP-hard.
.5 pts. Choose a reasonable NP-hard problem to reduce from.
2 pts. Auxiliary leaf gadget: attaching $(\Delta - 2)$ degree-1 vertices to each original vertex.
1.5 pts. Correctness:
- **.5 pts.** Forward direction (Hamiltonian path extends to degree-Δ tree).
- **.5 pts.** Reverse direction: auxiliary leaf edges are forced, consuming $\Delta - 2$ of the budget, leaving degree ≤ 2 on original vertices.
- **.5 pts.** Reverse direction: restricted tree is connected (auxiliary leaves don't bridge original vertices).

Note: This rubric is based on reducing from Hamiltonian path, which seemed the simplest approach. Other reductions are possible and the rubric should be adapted on a case-by-case basis.

Exercise 14.1. In *Minesweeper 2: On Graphs*, we have an undirected graph $G = (V, E)$ whose vertices are partitioned into *open* vertices V_{OPEN} and *hidden* vertices V_{HIDDEN}. Hidden vertices may contain mines. Each open vertex tells you how many of its hidden neighbors are mines.

Each open vertex $v \in V_{\text{OPEN}}$ has a label $\ell(v) \in \mathbb{Z}_{\geq 0}$. A *mine placement* is a function $M : V_{\text{HIDDEN}} \to \{0, 1\}$. A mine placement is *consistent* if for every open vertex v,

$$\ell(v) = \sum_{\substack{w \in V_{\text{HIDDEN}} \\ \{v,w\} \in E}} M(w).$$

That is, each label equals the number of adjacent hidden vertices containing a mine.

Given $G = (V_{\text{OPEN}} \sqcup V_{\text{HIDDEN}}, E)$ and $\ell : V_{\text{OPEN}} \to \mathbb{Z}_{\geq 0}$, consider the problem of determining whether there is a consistent mine placement. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 14.1.

We prove the problem is NP-hard by using it to solve 1-in-3 SAT.

Suppose we have a 1-in-3 SAT instance with variables $x_1, \dots, x_n$ and clauses $C_1, \dots, C_m$, where each clause contains exactly 3 literals.

We construct a Minesweeper instance as follows.

1. **Variable gadget.** For each variable x_i, we create two hidden vertices T_i and F_i, and one open vertex v_i with label $\ell(v_i) = 1$, connected to both T_i and F_i. Since the label is 1, any consistent mine placement has exactly one of T_i, F_i as a mine, modeling the assignment $x_i \in \{\texttt{true}, \texttt{false}\}$.

2. **Clause gadget.** For each clause $C_j = (\ell_1, \ell_2, \ell_3)$, we create one open vertex c_j with label $\ell(c_j) = 1$. For each literal ℓ_k in the clause:

   - If $\ell_k = x_i$ (positive), connect c_j to T_i.
   - If $\ell_k = \neg x_i$ (negative), connect c_j to F_i.

   Since the label is 1, any consistent placement has exactly one of these three literal-vertices as a mine, corresponding to exactly one literal being true.

We claim the 1-in-3 SAT instance is satisfiable iff the Minesweeper instance has a consistent mine placement.

Suppose the 1-in-3 SAT instance has a satisfying assignment. Set $M(T_i) = 1$ iff x_i is true, and $M(F_i) = 1$ iff x_i is false. Each variable vertex v_i has exactly one mine neighbor (since exactly one of T_i, F_i is a mine), matching its label of 1. Each clause vertex c_j has exactly one mine among its three literal-vertex neighbors (the one corresponding to the unique true literal), matching its label of 1. So the placement is consistent.

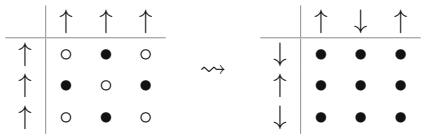
Conversely, suppose we have a consistent mine placement. For each variable vertex v_i (label 1, connected to T_i and F_i), exactly one of T_i, F_i is a mine. Set $x_i = \texttt{true}$ iff $M(T_i) = 1$. For each clause vertex c_j (label 1, connected to three literal-vertices), exactly one literal-vertex is a mine. If that vertex is T_i (where the literal is x_i), then x_i is true, so the literal x_i is true. If it is F_i (where the literal is $\neg x_i$), then $M(F_i) = 1$, so $M(T_i) = 0$ (by the variable gadget), meaning x_i is false and $\neg x_i$ is true. In both cases the literal is true, and exactly one literal per clause is true. So the assignment satisfies the 1-in-3 SAT instance.

Rubric.
1 pt. Identify the problem as NP-hard.
1 pt. Choose to reduce from 1-in-3 SAT (or similar constraint satisfaction problem).
2 pts. Variable gadget: two hidden vertices T_i, F_i per variable, with an open vertex of label 1 enforcing exactly one mine.
2 pts. Clause gadget: open vertex of label 1 per clause, connected to T_i for positive literals and F_i for negative literals.
2 pts. Forward direction: satisfying assignment gives a consistent mine placement.
2 pts. Reverse direction: consistent placement gives a satisfying assignment (using variable gadget to argue T_i and F_i are complementary).
Note: This rubric is based on reducing from 1-in-3 SAT. Other reductions are possible and the rubric should be adapted on a case-by-case basis.

Exercise 10.8. Let there be light. n^2 lightbulbs are arranged in an $n \times n$ grid. Each row has a switch and each column has a switch, for $2n$ switches in total. Each switch can be `up` or `down`. Flipping a row switch toggles every bulb in that row. Flipping a column switch toggles every bulb in that column.

The input is an $n \times n$ array A describing the state of each lightbulb (`on` or `off`) when every switch is `up`. (In general, some lightbulbs may be on, while others are off, initially.) The goal is to decide if there is a configuration of switches that turns on all the lightbulbs simultaneously.

For example, consider the 3×3 instance below ($\bullet$ = `on`, $\circ$ = `off`), with row switches on the left and column switches on top. The configuration on the right turns on all the lightbulbs.



For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 10.8. We solve the problem with 2-SAT.

Construction. Introduce a boolean variable r_i for each row switch ($r_i =$ `true` means the switch is `down`) and a boolean variable c_j for each column switch ($c_j =$ `true` means the switch is `down`). Since toggling is addition modulo 2, lightbulb (i, j) is `on` iff

$$A[i][j] \oplus r_i \oplus c_j = 1,$$

i.e., $r_i \oplus c_j = 1 - A[i][j]$. Each such XOR constraint translates to two 2-SAT clauses:

- If $r_i \oplus c_j = 1$: $(r_i \vee c_j) \wedge (\overline{r_i} \vee \overline{c_j})$.
- If $r_i \oplus c_j = 0$: $(r_i \vee \overline{c_j}) \wedge (\overline{r_i} \vee c_j)$.

We create one such pair of clauses for each of the n^2 lightbulbs, giving a 2-SAT instance with $2n$ variables and $2n^2$ clauses. We return the answer to this 2-SAT instance.

Correctness. We claim there is a switch configuration that turns on all the lightbulbs iff the 2-SAT instance is satisfiable.

Suppose there is a switch configuration that turns on all the lightbulbs. Let $r_i = \texttt{true}$ iff row switch i is `down`, and $c_j = \texttt{true}$ iff column switch j is `down`. Since every lightbulb is on, $A[i][j] \oplus r_i \oplus c_j = 1$ for all i, j, so every XOR constraint is satisfied and the corresponding 2-SAT clauses are satisfied.

Conversely, suppose the 2-SAT instance has a satisfying assignment to $r_1, \ldots, r_n, c_1, \ldots, c_n$. Set each row switch i to `down` iff $r_i = \texttt{true}$, and each column switch j to `down` iff $c_j = \texttt{true}$. Each pair of clauses encodes $r_i \oplus c_j = 1 - A[i][j]$, so $A[i][j] \oplus r_i \oplus c_j = 1$ and lightbulb (i, j) is on.

Running time. The reduction produces a 2-SAT instance with $2n$ variables and $2n^2$ clauses in $O(n^2)$ time. Solving 2-SAT takes time linear in the instance size, which is $O(n^2)$. Total: $O(n^2)$.

Rubric.
1 pt. Identify the problem as polynomial-time solvable.
1 pt. Model each switch as a boolean variable.
2 pts. Derive the XOR constraint $r_i \oplus c_j = 1 - A[i][j]$ for each bulb.
2 pts. Encode each XOR as two 2-SAT clauses (or otherwise correctly reduce to 2-SAT).
2 pts. Correctness:
- **1 pt.** Forward direction (valid switch configuration satisfies all clauses).
- **1 pt.** Reverse direction (satisfying assignment gives a valid configuration).

2 pts. Running time:
- **1 pt.** Size of the 2-SAT instance ($2n$ variables, $2n^2$ clauses).
- **1 pt.** Invoking 2-SAT in $O(n^2)$ time.

Exercise 17.6. As you know well, this class is a lot: lots of students, lots of homework, and a huge staff. To divvy up the grading load, we want to divide the staff into two teams, Team Divide and Team Conquer, which will alternate homework grading week by week. Some staff are already committed to a team. The rest are flexible. The staff also has many pairwise friends between them. We prefer to keep friends together; friendly grading teams make for more friendly grading. (The teams do not have to be the same size.)

Formally, let S denote the set of n staff members. We have as input a list of m pairwise friendships among the staff. We also have two disjoint subsets $D, C \subseteq S$ of staff members preassigned to Team Divide and Team Conquer, respectively. The goal is to assign the remaining staff members in $S \setminus (C \cup D)$ so as to minimize the number of friendships split across the teams. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 17.6. We reduce to minimum (s, t)-cut.

Construction. We build a flow network H as follows.

1. Create a supersource $s^\star$ by taking all staff in D and contracting them into a single vertex. Similarly, create a supersink $t^\star$ by contracting all staff in C.

2. For each friendship $\{u, v\}$, add edges (u, v) and (v, u), each with capacity 1. (Friendships among D members or among C members vanish under contraction.)

We compute the minimum $(s^\star, t^\star)$-cut in H. The $s^\star$-side of the cut is Team Divide, and the $t^\star$-side is Team Conquer.

Correctness. We claim that the minimum number of split friendships equals the minimum $(s^\star, t^\star)$-cut in H.

By contraction, every $d \in D$ is on the $s^\star$-side and every $c \in C$ is on the $t^\star$-side. The remaining vertices (flexible staff) can be on either side.

Consider a friendship $\{u, v\}$ split across the cut, say with u on the $s^\star$-side and v on the $t^\star$-side. The directed edge (u, v) crosses the cut (from $s^\star$-side to $t^\star$-side) and contributes 1 to the cut value. The edge (v, u) goes from $t^\star$-side to $s^\star$-side and does not contribute. So each split friendship contributes exactly 1 to the cut value.

Conversely, every capacity-1 edge crossing the cut corresponds to one direction of a split friendship.

Therefore the cut value equals the number of split friendships, and minimizing the cut minimizes the number of split friendships.

Running time. The flow network has $O(n)$ vertices and $O(m)$ edges. All capacities are 1 (or ∞, which are never saturated), so the max flow λ is at most m. Using Ford-Fulkerson, the minimum cut can be computed in $O(m\lambda) = O\big(m^2\big)$ time.

Rubric.
1 pt. Identify the problem as polynomial-time solvable.
1 pt. Connection to min-cut / max-flow.
1 pt. Supersource and supersink.
1 pt. Contracting preassigned members into supersource/supersink (or equivalently, ∞-capacity edges).
1 pt. Friendship edges with capacity 1 (in both directions).
2 pts. Correctness (both directions of the iff):
- **1 pt.** Forward direction (team assignment $\to$ cut with matching value; each split friendship contributes exactly 1).
- **1 pt.** Reverse direction (cut $\to$ team assignment; preassigned members forced to correct side).

1 pt. Driver: returning the $s^\star$-side as Team Divide and $t^\star$-side as Team Conquer.
2 pts. Running time:
- **2 pts.** Ford-Fulkerson with $\lambda \le m$, giving $O(m\lambda) = O\big(m^2\big)$.
- **1 pts.** (partial) Ford-Fulkerson with incorrect λ bound.
- **1 pts.** (partial) Edmonds-Karp / shortest augmenting paths, giving $O\big(m^2 n\big)$.

Exercise 12.24. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two directed graphs with $n_i = |V_i|$ vertices and $m_i = |E_i|$ edges each. Suppose we have a labeling $\ell : V_1 \cup V_2 \to \mathcal{A}$, where $\mathcal{A}$ is an alphabet. (Different vertices may share the same letter.) A *common path* is defined as a path p_1 in G_1 and a path p_2 in G_2 such that the letters along the paths are the exact same. (The first vertices of p_1 and p_2 have the same letter, the second vertices have the same letter, etc. p_1 and p_2 must have the same length.) Consider the problem of computing the length of a longest common path in G_1 and G_2.

1. (5 points) Suppose G_1 and G_2 are both DAGs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Now suppose G_1 and G_2 are general directed graphs. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 12.24.

Part 1 (product graph approach). We build an auxiliary *product graph* $H = (V_H, E_H)$:

- $V_H = \{(u, v) : u \in V_1,\ v \in V_2,\ \ell(u) = \ell(v)\}$.
- $((u_1, v_1), (u_2, v_2)) \in E_H$ iff $(u_1, u_2) \in E_1$ and $(v_1, v_2) \in E_2$.

Since G_1 and G_2 are both DAGs, H is also a DAG (any cycle in H would project to a cycle in G_1 or G_2). We compute the longest path in H (using topological sort and dynamic programming, in $O(|V_H| + |E_H|)$ time) and return its length.

We claim that the length of the longest common path in (G_1, G_2) equals the length of the longest path in H.

A common path of length k consists of paths $u_0, u_1, \ldots, u_k$ in G_1 and $v_0, v_1, \ldots, v_k$ in G_2 with $\ell(u_i) = \ell(v_i)$ for all i. The sequence $(u_0, v_0), (u_1, v_1), \ldots, (u_k, v_k)$ is a path of length k in H. Conversely, any path in H of length k yields a common path of length k by projecting onto the two coordinates.

The product graph has at most $n_1 n_2$ vertices and at most $m_1 m_2$ edges, so the total running time is $O(n_1 n_2 + m_1 m_2)$.

Part 1 (direct DP approach). *1. Recursive spec / induction hypothesis.* For $u \in V_1$ and $v \in V_2$ with $\ell(u) = \ell(v)$, let $\texttt{LCP}(u, v)$ be the length of a longest common path starting at u in G_1 and v in G_2.

2. Recursive implementation.

```
LCP (u,v)
1. Let best = 0.
/* A common path of length 0 (just the starting vertices). */
2. For each (u,u') ∈ E_1 and (v,v') ∈ E_2 with ℓ(u') = ℓ(v'):
   A. best = max{best, 1 + LCP(u',v')}.
3. Return best.
```

3. Solving the original problem. Return $\max_{u \in V_1,\, v \in V_2,\, \ell(u)=\ell(v)} \mathsf{LCP}(u, v)$.

4. Running time with caching / dynamic programming. There are at most $n_1 n_2$ subproblems. Each subproblem $\mathsf{LCP}(u, v)$ iterates over all pairs of outgoing edges from u and v, taking $O(\deg(u) \cdot \deg(v))$ time (excluding recursive calls). Summing over all subproblems:

$$\sum_{\substack{u \in V_1,\, v \in V_2 \\ \ell(u)=\ell(v)}} \deg(u) \cdot \deg(v) \;\leq\; \left(\sum_{u \in V_1} \deg(u) \right) \left(\sum_{v \in V_2} \deg(v) \right) = m_1 \cdot m_2.$$

The driver takes $O(n_1 n_2)$ time. Total: $O(n_1 n_2 + m_1 m_2)$.

Part 2. We prove the problem is NP-hard by reduction from longest path.

Given a directed graph $G = (V, E)$, we set $G_1 = G_2 = G$ and assign a distinct label to each vertex (e.g., $\ell(v) = v$). We query a black box for the longest common path in (G_1, G_2) and return its length.

We claim that the longest path in G has the same length as the longest common path in (G_1, G_2).

Suppose G has a path $v_0, v_1, \ldots, v_k$ of length k. Using this path in both G_1 and G_2 gives a common path of length k (the labels match since $G_1 = G_2$ and the same vertices are visited).

Conversely, suppose there is a common path of length k, consisting of $p_1 = u_0, \ldots, u_k$ in G_1 and $p_2 = v_0, \ldots, v_k$ in G_2 with $\ell(u_i) = \ell(v_i)$ for all i. Since labels are distinct, $u_i = v_i$ for all i. So $p_1 = p_2$ and this is a path of length k in G.

Rubric.

Part 1, product graph approach (5 points).
.5 pts. Identify the problem as polynomial-time solvable.
2 pts. Product graph construction:
- **1 pt.** Vertices are pairs (u, v) with matching labels.
- **1 pt.** Edges from pairs of edges in G_1 and G_2.

.5 pts. Observe the product graph is a DAG and invoke longest path in DAG.
1 pt. Correctness (both directions): common path $\leftrightarrow$ path in product graph.
1 pt. Running time: $O(n_1 n_2 + m_1 m_2)$.

Part 1, direct DP approach (5 points).
.5 pts. Identify the problem as polynomial-time solvable.
2 pts. Recursive spec: $\texttt{LCP}(u, v)$ is the length of a longest common path starting at u in G_1 and v in G_2 (with $\ell(u) = \ell(v)$). (This serves as the induction hypothesis and substitutes for a correctness proof.)
1 pt. Recursive implementation (iterate over pairs of outgoing edges with matching labels).
.5 pts. Driver: maximize $\texttt{LCP}(u, v)$ over all (u, v) with $\ell(u) = \ell(v)$.
1 pt. Running time: $O(n_1 n_2)$ subproblems, $O(m_1 m_2)$ total work, with DP/caching.

Part 2 (5 points).
1 pt. Identify the problem as NP-hard.
1 pt. Choose to reduce from longest path.
1 pt. Construction: $G_1 = G_2 = G$ with distinct labels.
1 pt. Forward direction (path in G gives a common path).
1 pt. Reverse direction (distinct labels force $p_1 = p_2$).

F.11 Homework 8

Exercise 18.4. This exercise develops a $O\big((m^2 + mn\log(n))\log(\lambda)\big)$-time algorithm for maximum (s,t)-flow and builds on ideas from exercise 18.3.

1. Prove the following: Given any (s,t)-flow problem with max flow value $\lambda > 0$, there exists an (s,t)-path where the minimum capacity edge is at least λ/m.

2. Describe a $O(m + n\log(n))$-time algorithm to find the path described above.[37]

3. Based on the two parts above, design and analyze an (s,t)-max flow algorithm that runs in $O(m(m + n\log(n))\log(\lambda))$ time for integer capacities, where λ denotes the value of the maximum flow.[38][39] (The algorithm does not know the true value of λ *a priori*.)

Solution to exercise 18.4. This algorithm is well-known and also due to [EK72]. It is also covered in [Eri19]. (Here we give a slightly slower version; the running time can be improved to a cleaner $O\big(m^2\log(\lambda)\big)$ by improving part 2 to $O(m)$ time.)

Part 1. Let $f : E \to \mathbb{R}_{\geq 0}$ be the maximum flow, of size λ. By exercise 18.3, f decomposes to a path packing of at most m distinct paths of total size λ. One of these paths has weight at least λ/m in the path packing, hence the minimum edge capacity along that path is also at least λ/m.

Part 2. One can make a procedure similar to the search algorithm for MST and Dijkstra's algorithm for shortest paths, where starting from $S = \{s\}$, we repeatedly add the opposite endpoint of the maximum capacity edge in $\delta^+(S)$, until we find a path to t. We can use Fibonacci heaps to quickly obtain each subsequent edge in $O(\log n)$ amortized time per edge.

The high-level algorithm is as follows.

`max-bottleneck-path(`$G = (V,E)$`, `$c : E \to \mathbb{R}_{\geq 0}$`, `$s,t \in V$`)`

1. If s cannot reach t then return `false`.
2. Set $T \leftarrow \emptyset$ and $S \leftarrow \{s\}$.
3. While $t \notin S$.
   A. Let $e = \{u,v\}$ be the maximum capacity edge in $\delta^+(S)$.
   B. Set $T \leftarrow T + e$ and $S \leftarrow S \cup e$.
4. Return the (unique) (s,t)-path in T.

To accelerate the algorithm above, we maintain a Fibonacci max-heap over all $v \in V \setminus S$, where the priority of a vertex v is the maximum capacity of any edge from S to v (or $-\infty$ if there isn't any). We also keep track of the maximum capacity edges that define the priorities for each $v \in V \setminus S$. In each iteration of (3.A), the maximum priority vertex v in the heap is the terminating point of

[37] $O(m\log n)$ time is a little easier and this running time would still get partial credit. Even if the $O(m + n\log(n))$-running time eludes you, you can assume it as a black box for the next part.

[38] This is polynomial with respect to the bit complexity of the input.

[39] A possibly helpful bit of math: for (small) $\epsilon > 0$, $\log_{1+\epsilon}(x) \leq O(\log(x)/\epsilon)$ is a good approximation (which you may want to verify for yourself).

the maximum capacity edge $e = (u, v)$ in $\delta^+(S)$. In addition to adding e to T and v to S, we update the priorities based on all the edges (v, w) leaving v. Overall, we make $O(m)$ updates and remove $O(n)$ elements from the top of the heap, so by fact 7.2, we spend $O(m + n \log(n))$ total time on the heap operations.

Part 3. We repeatedly invoke the procedure above to find the maximum bottleneck path in the residual graph, and augment as much as possible along this path. Note that for integral capacities, we always route an integral amount of flow, and maintain integral capacities in the residual graph.

To bound the amount of flow routed per iteration, suppose that at a particular iteration, we are x units of flow away from the maximum flow. Part 1 implies that there is an (s, t)-path in the residual graph where the minimum (residual) capacity is at least x/m. We can find such a path in $O(m + n \log(n))$ time using part 2. After augmenting along this path, we are at most $(1 - 1/m)x$-units of flow from a maximum flow. Put another way, we have reduced the maximum flow in the residual graph by a $m/(m-1) = (1 + 1/(m-1))$-multiplicative factor.

Now we apply the hint. Recall that for $\epsilon > 0$, $\log_{1+\epsilon}(\lambda)$ is defined by

$$(1 + \epsilon)^{\log_{1+\epsilon}(\lambda)} = \lambda.$$

For us, setting $\epsilon = 1/(m-1)$, $\log_{1+\epsilon}(\lambda)$ is the number of times we can reduce the residual max flow by a $(1 + \epsilon)$-multiplicative factor before the residual max flow is 1 or less. In particular, we need

$$\log_{1+\epsilon}(\lambda) = O(\log(\lambda)/\epsilon) = O(m \log(\lambda)).$$

before the residual maximum flow is less than 1. Since the flow is always integral, a residual max flow less than 1 is the same as a residual max flow of size 0; that is, we have a max flow.

Thus we need $O(m \log(\lambda))$ iterations. The total running time is $O(m(m + n \log(n)) \log(\lambda))$.

Exercise 19.2. Every semester we have to schedule PSO's for all the students, which is complicated by (a) room availability mixed with (b) time availability both with respect to the students and the rooms. Suppose there are m total students, n available time slots, and p different rooms. Suppose we had the following information.

1. For each student (indexed by) $i \in [m]$, there is a subset of times $T_i \subseteq [n]$ of times when they can attend PSO.

2. For each time slot $j \in [n]$, there is a subset of rooms $R_j \subseteq [p]$ available at that time.

3. For each room $k \in [p]$ there is a maximum capacity C_k of the number of students that can fit in the room at any given time.

From this information we have the following scheduling problems.

1. Consider the problem of deciding if there is a way to schedule all the students to times and rooms while respecting all the constraints listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. In addition to making sure each room can fit all its students, we also want to minimize the maximum number of students in any room at any time, to decrease the maximum load on the TA's. Consider the problem of creating a schedule that minimizes the maximum number of students in any room, while also respecting the constraints listed above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 19.2.

Part 1. We reduce the problem to generalized assignment. The "individuals" correspond to the m students. The "jobs" correspond to combinations (j, k) of a time slot $j \in [n]$ and a room $k \in R_j$ available at time j. A student is "qualified" for time-and-room (j, k) if $j \in T_i$. We want to assign each student i to one time-and-room (j, k), and each time-and-room (j, k) can be assigned at most C_k sutdents.

We can model this problem by the following flow network.

1. For every student $i \in [m]$, we create a vertex a_i.
2. For every time slot $j \in [n]$, and every room $k \in R_j$ available at time j, we create a vertex $b_{j,k}$.
3. For every student i, every time slot $j \in T_i$, and every room $k \in R_j$, we have an edge of capacity 1 from a_i to $b_{j,k}$.
4. We have a source vertex s and a sink vertex t.
5. For each student i, we have an edge (s, a_i) of capacity 1.
6. For each time slot $j \in [n]$ and every room $k \in R_j$ available at time j, we have an edge $(b_{j,k}, t)$ with capacity equal to the capacity C_k of room k.

We compute a maximum integral (s, t)-path packing (alternatively, a maximum integral flow) in the graph above with the Ford-Fulkerson algorithm, and return `true` iff the algorithm computes a flow of size m. The maximum flow is at most m, and there are $O(mnp)$ edges, so this algorithm takes $O\big(m^2np\big)$ time.

We argue that there is a packing of m (s, t)-paths iff there is a feasible scheduling.

Suppose we have a packing of m (s, t)-paths. Each path is of the form $(s, a_i, b_{j,k}, t)$, where $i \in [m]$, $j \in [n]$, and $k \in R_j$. Consider the schedule where we interpret each path $(s, a_i, b_{j,k}, t)$ as assigning student i to room j and time k.

We verify that this schedule satisfies each of the requirements.

- Since there are m edges leaving s, each with unit-capacity and of the from (s, a_i) for a student $i \in [m]$, we must have one path for each arc (s, a_i). This implies that each student is assignment some room and time.
- For each student i, since a_i is only connected to time-and-room (j, k) if $j \in T_i$, student i assigned to a time j where $j \in T_i$.
- Since the arc $(b_{j,k}, t)$ has capacity R_k, at most R_k paths go through $b_{j,k}$, hence room k is assigned at most R_k students at time j.

Conversely, suppose we have a feasible schedule of all the students. Consider the path packing where for each student i assigned to room j and time k, we have a path $(s, a_i, b_{j,k}, t)$. This gives m paths. We verify that these paths satisfy all the capacity constraints.

- Each edge (s, a_i) is used once, corresponding to the assignment for student i.
- For each path $(s, a_i, b_{j,k}, t)$, $(a_i, b_{j,k})$ is an arc because we must have $j \in T_i$.
- Each edge $(b_{j,k}, t)$ is used at most R_k times, because at most R_k students are assigned to room k.

So these m paths give a feasible path packing.

Part 2. The high-level idea is to reduce to part 1 as follows. Suppose we knew the true minimum-maximum capacity, which we denote $\mu^\star$. Then we could call part 1 except replacing each room capacity R_k with the minimum of R_k and $\mu^\star$.

Of course we do not know $\mu^\star$ *a priori*. However, we can binary search for the smallest value $\mu \in [m]$ such that, replacing each R_k with the minimum of R_k and μ, there is a feasible schedule. This takes $O(\log m)$ calls to the algorithm from part 1, so the total running time is $O\big(m^2 np \log(m)\big)$.

Exercise 19.3. Suppose we are given a two-dimensional array $A[1..m][1..n]$ of non-negative real values such that each row and column sum is an integer. We want to round A to an integer matrix, replacing each entry $x \in A$ with either $\lceil x \rceil$ or $\lfloor x \rfloor$, while maintaining the sum in any row or column of A. For example,

$$\begin{pmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{pmatrix} \text{ rounds to } \begin{pmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{pmatrix}.$$

Consider the problem of computing such a rounding, or declaring that none exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 19.3. By subtracting $\lfloor A[i][j] \rfloor$ from every entry $A[i][j]$, we may assume that every entry $A[i][j]$ is between 0 and 1.

We construct a graph as follows.

1. A source s and a sink t.
2. For each row i, a vertex a_i, and an edge (s, a_i) with capacity equal to the sum of row i.
3. For each column j, a vertex c_j, and an edge (b_j, t) with capacity equal to the sum of column j.
4. For each row i and column j with $A[i][j] > 0$, we have an edge (a_i, c_j) with capacity 1.

Let F be the total sum of all the coefficients $A[i][j]$; F is an integer since each row sum is an integer. F is also the capacity of the (s,t)-cuts $\delta^+(s)$ and $\delta^-(s)$, since each edge (s,a_i) has capacity equal to the sum of row i, and each edge (b_j,t) has capacity equal to the sum of column j. We claim that there is a packing of F (s,t)-paths iff there is a rounding of coordinates that preserves all row and column sums.

First, suppose we had a path packing of F (s,t)-paths. Each path is of the form (s,a_i,c_j,t) for a row i and a column j. The arc (a_i,c_j) can appear in at most one path, because (a_i,c_j) is capacity 1.

Consider the rounding where for each path $(s,a_i,,c_j,t)$, we round $A[i][j]$ up to 1. The remaining coordinates are rounded down. Because $\delta^+(s)$ has capacity F, the path packing must use up the capacity of every edge of the form (s,a_i). This means that the number of coordinates in row i that are rounded up equals the sum of row i, as desired. Similarly because $\delta^-(t)$ has capacity F, the path packing must use up the capacity of every edge of the form (c_j,t). This means that the number of coordinates in column j that are rounded up equals the sum of column j. So our rounding preserves all row sums and column sums.

Conversely, suppose we can round the coordinates to nearest integers while preserving all row and column sums. In particular exactly F coordinates get rounded up. We create F (s,t)-paths, where for each coordinate $A[i][j]$ that was rounded up, we add the path (s,a_i,c_j,t). Each (a_i,c_j) appears in at most one path. For each row i, since the number of coordinates rounded up in row i equals the row sum, the number of paths using (s,a_i) is exactly equal to its capacity. Similarly the number of paths using each (c_j,t) equals the column sum of c_j. So these F (s,t)-paths obey all capacity constraints in our flow network.

Ultimately we construct the graph as described above, compute the maximum flow, and return `true` (indicating that there is a feasible rounding) iff we find a flow of size F. Since F is bounded above by mn, that there are $O(mn)$, the augmenting paths algorithm (Ford-Fulkerson) would take $O\big(m^2n^2\big)$ time.

F.12 Homework 9

Exercise 22.3. Section 22.5.1 showed how to use two (immutable) stacks to implement a queue with $O(1)$ amortized performance. Here we will develop a more sophisticated, *double-ended queue* using three immutable stacks. (It's mostly using two stacks, but occasionally needs a third stack as a buffer.)

A double ended queue is like a queue except we can insert and delete from both ends of the queue. Here we designate one end as the "front" and the other end as the "back", and define the following four operations.

1. `push-front(`x`)`: inserts element x at the front of the queue.
2. `push-back(`x`)`: inserts element x at the back of the queue.
3. `pop-front()`: removes the first element of the queue.
4. `pop-back()`: removes the last element of the queue.

(The data structure throws an error if the user calls either `pop` operation and there are no elements remaining.)

1. The natural extension to the two-stack approach is to flip all the elements over whenever you are popping from a side with an empty stack. Prove that this approach does *not* get $O(1)$ amortized time.
2. Design and analyze a different data structure, using at most 3 immutable stacks, that obtains $O(1)$ amortized time for each operation. (Our solution is somewhat similar to the two-stack approach, but the modification requires using a third stack on occasion as a buffer.)

Solution to exercise 22.3.

Part 1. We first push n elements in the back in $O(1)$ time per element. Then we alternatively pop from the front and pop from the back. Each pop forces the data structure to move all the elements from one stack to the other, and takes $O(k)$ time if there are k elements remaining. The total running time of these n operations is proportional to

$$n + \sum_{k=1}^{n} k = \Omega\big(n^2\big).$$

So the amortized running times of one of these operations (`push-back`, `pop-front`, or `pop-back`) must be at least $\Omega\big(n^2/n\big) = \Omega(n)$.

Part 2. We analyze a data structure that maintains two stacks that we call the "front stack" and the "back stack". At a high-level, `push-front` and `pop-front` try to push-onto and pop-from the front stack, and `push-back` and `pop-back` try to push onto and pop from the back stack.

If we call `pop-front` and the front stack is empty, then we split the back stack in half (with the bottom half bigger, if the size is odd). The bottom half of the back stack is reversed and moved to the front stack. The top half of the back stack remains the back stack. The front stack is now nonempty, and we pop the top element.

Similarly if we call `pop-back` and the back stack is empty, then we move the bottom half (rounded up) from the front stack to the back stack.

Let S_{F} denote the stack at the front of the queue (so to speak) and let S_{B} dnote the stack at the back of the queue. We let $|S_{\mathrm{F}}|$ and $|S_{\mathrm{B}}|$ denote the number of elements in each queue. We define a potential function by

$$\Phi = \max\{|S_{\mathrm{F}}|, |S_{\mathrm{B}}|\}.$$

Each push (front and back) takes $O(1)$ (real) time and increases Φ by 1.

Setting aside rebalancing, each pop (front and back) takes $O(1)$ time and does not increase Φ. When rebalancing (say) n elements, Φ equals n, and decreases to $\lceil n/2 \rceil$. The decrease of $\Omega(n)$ pays for the $O(n)$ time to rebalance the stacks.

(An alternative approach uses the potential function $\Phi = ||S_{\mathrm{F}}| - |S_{\mathrm{B}}||$.)

Rubric.

Part 1 (3 points).
2 pts. Adversarial sequence forcing a full flip on every pop.
1 pt. Conclude that some operation must take $\Omega(n)$ amortized time.

Part 2 (7 points).
3 pts. Data structure design.
- **1 pt.** Maintain a front stack and a back stack.
- **1 pt.** Handle pushes and the easy (non-empty) case of pops.
- **1 pt.** Rebalance by splitting the nonempty stack in half when the other is empty.

4 pts. Amortized analysis.
- **1 pt.** Choose a valid potential function.
- **1 pt.** Prove that pushes take $O(1)$ amortized time.
  - **.5 pts.** Observe that pushing takes constant real time.
  - **.5 pts.** Observe that the potential increases by $O(1)$.
- **1 pt.** Prove that pops take $O(1)$ amortized time, setting aside rebalancing.
  - **.5 pts.** Observe that popping takes constant real time.
  - **.5 pts.** Observe that the potential does not increase.
- **1 pt.** Show that the decrease in potential pays for the $O(n)$ rebalance.

Note: This rubric assumes a potential function argument. Other approaches (e.g., charging) are possible and the rubric should be adapted on a case-by-case basis.

Exercise 25.12. Recall the `quick-select` algorithm introduced in Section 25.1.3. The goal of this exercise is to prove that `quick-select` takes $O(n)$ time in expectation. Below we present two different approaches which offer two different perspectives. We ask you to submit solutions to just one of the two approaches, though we strongly recommending trying to solve and understanding both. Both analyses should use linearity of expectation and we ask you to point this out explicitly.

1. *Approach 1.* Analyze `quick-select` similarly to `quick-sort`, based on the sum of indicators X_{ij}.

   One approach is to reduce to a separate analysis for each of the following 4 classes of pairs:

   (a) X_{ij} where $i < j < k$,

   (b) X_{ij} where $i < k < j$,

   (c) X_{ij} where $k < i < j$, and

   (d) X_{ij} where either $i = k$ or $j = k$.

   For each case, show that the expected sum is $O(n)$.

2. *Approach 2.* The following approach can be interpreted as a randomized divide and conquer argument. We are arguing that with constant probability, we decrease the input by a constant factor, from which the fast (expected) running time follows.

   (a) Consider a coin that flips heads with probability p. Suppose we repeatedly flip the coin until it comes up heads. What is the expected number of coin tosses until it comes up heads? Prove your answer.[40]

   (b) Consider again `quick-select`. Consider a single iteration where we pick a pivot uniformly at random and throw out some elements. Prove that with some constant probability p, we either sample the kth element or throw out at least $1/4$ of the remaining elements.

   (c) For each integer i, prove that the expected number of iterations (i.e., rounds of choosing a pivot) of `quick-select`, where the number of elements remaining is in the range $[(4/3)^i, (4/3)^{i+1})$, is $O(1)$.

   (d) Fix an integer i, and consider the amount of time spent by `quick-select` while the number of elements remaining is greater than $(4/3)^{i-1}$ and at most $(4/3)^i$. Show that that the expected amount of time is $\leq O\big((4/3)^i\big)$

   (e) Finally, use the preceding part to show that the expected running time of `quick-select` is $O(n)$.

Solution to exercise 25.12.

[40] There is a slick proof.

Approach 1. The running time, up to constants, is given by $\sum_{i<j} X_{ij}$. The expected running time is given by $\mathbf{E}\left[\sum_{i<j} X_{ij}\right]$. By (a) linearity of expectation, we have

$$\begin{aligned}\mathbf{E}\left[\sum_{i<j} X_{ij}\right] &\overset{\text{(a)}}{=} \sum_{i<j} \mathbf{E}[X_{ij}] \\ &= \sum_{i,j:i<j<k} \mathbf{E}[X_{ij}] + \sum_{i,j:i<k<j} \mathbf{E}[X_{ij}] + \sum_{i,j:k<i<j} \mathbf{E}[X_{ij}] + \sum_{\substack{i,j:i<j,\\ i=k \text{ or } j=k}} \mathbf{E}[X_{ij}].\end{aligned}$$

Each sum in the RHS above represents one of the four cases. Below, we show that each sum is at most $O(n)$.

(a) $\boldsymbol{i<j<k}$**:** i and j are compared iff either is chosen as a pivot before any other element with rank between i and k. That is,

$$\mathbf{E}[X_{ij}] = \frac{2}{k-i+1}.$$

We have

$$\sum_{i=1}^{k-2}\sum_{j=i+1}^{k-1} \mathbf{E}[X_{ij}] = \sum_{i=1}^{k-2}\sum_{j=i+1}^{k-1} \frac{2}{k-i+1} \leq \sum_{i=1}^{k-2} \frac{2(k-i-1)}{k-i+1} \leq O(k) \leq O(n).$$

(b) $\boldsymbol{i<k<j}$**:** i and j are compared iff either is selected as a pivot before any element between i and j. That is,

$$\mathbf{E}[X_{ij}] = \frac{2}{j-i+1}.$$

Consider the sum

$$\sum_{i=1}^{k-1}\sum_{j=k+1}^{n} \mathbf{E}[X_{ij}] = \sum_{i=1}^{k-1}\sum_{j=k+1}^{n} \frac{2}{j-i+1}$$

For $h \in [n]$, the term $2/(h+1)$ appears at most $h-1$ times (as there are $h-1$ intervals $[i,j]$ of length $j-i=h$, containing k strictly in between). Consequently

$$\sum_{i=1}^{k-1}\sum_{j=k+1}^{n} \frac{2}{j-i+1} \leq \sum_{h=1}^{n} \min\{k,h\} \cdot \frac{2}{h+1} \leq O(n).$$

(c) $\boldsymbol{k<i<j}$**:** i and j are compared iff either are selected as a pivot before any element between k and j. That is,

$$\mathbf{E}[X_{ij}] = \frac{2}{j-k+1}.$$

Similar to the first case, we have

$$\sum_{j=k+2}^{n}\sum_{i=k+1}^{j-1}\mathbf{E}[X_{ij}] = \sum_{j=k+2}^{n}\sum_{i=k+1}^{j-1}\frac{2}{j-k+1} \leq \sum_{j=k+2}^{n}\frac{2(j-k-2)}{j-k-1} \leq O(n),$$

as desired.

(d) **Either $i = k$ or $j = k$:** i and j are compared iff either are selected as a pivot before any element between them. We have

$$\mathbf{E}[X_{ij}] = \frac{2}{j-i+1}.$$

We have

$$\sum_{i=1}^{k-1}\frac{2}{k-i} + \sum_{j=k+1}^{n}\frac{2}{j-i} = O(\log n) \leq O(n),$$

as desired.

Approach 2.

(a) Suppose there are n elements remaining and we want to seek the kth element. With probability $1/2$, we pick from one of the middle $\lceil n/2 \rceil$ elements, in which case we either identify the kth element, or throw out at least $n/4$ elements.

(b) Suppose the number of elements remaining is in the range $[(4/3)^i, (4/3)^{i+1})$. By exercise J.7, it takes $O(1)$ iterations in expectation to pick from one of the middle $\lceil n/2 \rceil$ elements, in which case we throw out at least $(1/4)$th of the elements, and the total number of elements drops out of the range.

(c) Let X_i be the expected number of iterations where the number of elements is in the range $[(4/3)^i, (4/3)^{i+1}])$. By part (b), $\mathbf{E}[X_i] = O(1)$. Each such iteration takes $O\big((4/3)^i\big)$ time. The expected time over all such iterations (up to constants) is given by

$$\mathbf{E}\big[(4/3)^i X_i\big] = (4/3)^i\,\mathbf{E}[X_i] = O\big((4/3)^i\big),$$

as desired.

(d) For $i \in \mathbb{Z}_{\geq 0}$, let T_i be the total time spent where the total number of elements lies in the range $[(4/3)^i, (4/3)^{i+1}]$. By part (c), we have

$$\mathbf{E}[T_i] = O\big((4/3)^i\big)$$

for $i \leq \log_{4/3}(n)$. By linearity of expectation, the total expected running time is given by

$$\mathbf{E}\left[\sum_{i=1}^{\lfloor \log_{4/3} n \rfloor} T_i\right] = \sum_{i=1}^{\lfloor \log_{4/3} n \rfloor}\mathbf{E}[T_i] = \sum_{i=1}^{\lfloor \log_{4/3} n \rfloor}(4/3)^i = O\Big((4/3)^{\log_{4/3}(n)}\Big) = O(n),$$

as desired.

Rubric.
5 pts. Approach 1
- **1 pt.** Part (a)
  - **.5 pts.** Analyzing $\mathbf{E}[X_{ij}]$.
  - **.5 pts.** Summing over $\mathbf{E}[X_{ij}]$'s.
- **2 pts.** Part (b)
  - **.5 pts.** Analyzing $\mathbf{E}[X_{ij}]$.
  - **1 pt.** 1.5 Summing over $\mathbf{E}[X_{ij}]$'s with justification.
- **1 pt.** Part (c)
  - **.5 pts.** Analyzing $\mathbf{E}[X_{ij}]$.
  - **.5 pts.** Summing over $\mathbf{E}[X_{ij}]$'s.
- **1 pt.** Part (d)
  - **.5 pts.** Analyzing $\mathbf{E}[X_{ij}]$.
  - **.5 pts.** Summing over $\mathbf{E}[X_{ij}]$'s.

5 pts. Approach 2
- **1 pt.** Part (a)
  - **.5 pts.** Explaining the pivot-logic.
  - **.5 pts.** Getting $p = 1/2$.
- **2 pts.** Part (b)
  - **1 pt.** Observe that $(3/4) \cdot (4/3)^{i+1} = 4/3$, so one "good" pivot drops you into the next range.
  - **1 pt.** Apply exercise J.7 to getting a good pivot.
- **1 pt.** Part (c)
  - **1 pt.** .5 Observe that each iteration/trial takes $O\big((4/3)^i\big)$ time.
  - **1 pt.** .25 Model as $\mathbf{E}\big[(4/3)^i X_i\big]$
  - **1 pt.** .25 $\mathbf{E}\big[(4/3)^i X_i\big]$ using previous part.
- **1 pt.** Part (d)
  - **.5 pts.** Model total expected running time as expected sum of running times for each i.
  - **.25 pts.** Apply linearity of expectation.
  - **.5 pts.** Plug in and correctly calculate the sum is $O(n)$.

Exercise 25.14. This exercise is about a simple randomized algorithm for *verifying* matrix multiplication. Suppose we have three $n \times n$ matrices A, B, C. We want to verify if $AB = C$. Of course one could compute the product AB and compare it entrywise to C. But multiplying matrices is slow: the straightforward approach takes $O\big(n^3\big)$ time and there are (more theoretical) algorithms with running time roughly $O\big(n^{2.37\cdots}\big)$. We want to test if $AB = C$ in closer to n^2 time.

The algorithm we analyze is very simple. Select a point $x \in \{0,1\}^n$ uniformly at random. (That is, each $x_i \in \{0,1\}$ is an independently sampled bit.) Compute $A(Bx)$ and Cx, and compare their entries. (Note that it is much faster to compute $A(Bx)$ than AB.) If they are unequal, then certainly $AB \neq C$ and we output `false`. Otherwise we output `true`. Note that the algorithm is always correct if $AB = C$, but could be wrong when $AB \neq C$. We will show that if $AB \neq C$, the algorithm is correct with probability at last $1/2$.

1. Let $y \in \mathbb{R}^n$ be a fixed nonzero vector, and let $x \in \{0,1\}^n$ be drawn uniformly at random. Show that $\langle x, y \rangle \stackrel{\text{def}}{=} \sum_{i=1}^n x_i y_i \neq 0$ with probability at least $1/2$. [41]

2. Use the preceding result to show that if $AB \neq C$, then with probability at least $1/2$, $ABx \neq Cx$.[42]

3. Suppose we want to decrease our probability of error to (say) $1/n^2$. Based on the algorithm above, design and analyze a fast randomized algorithm with the following guarantees.

   - If $AB = C$, then it always reports that $AB = C$.
   - If $AB \neq C$, then with probability of error at most $1/n^2$, it reports that $AB \neq C$.

   (Your analysis should include the running time as well.)[43]

Solution to exercise 25.14.

1. Let y_k be the last nonzero coefficient of y. Let $\alpha = \sum_{i=1}^{k-1} x_i y_i$ be the partial sum over the first $k-1$ coordinates, and consider the probability of $\langle x, y \rangle = 0$ conditional on α. If $\alpha = 0$, then $\langle x, y \rangle \neq 0$ if $x_k = 1$. If $\alpha \neq 0$, then $\langle x, y \rangle \neq 0$ if $x_k = 0$. Thus

$$\begin{aligned}
\mathbf{P}[\langle x, y \rangle \neq 0] &= \mathbf{P}[\langle x, y \rangle \neq 0 \,|\, \alpha = 0] \, \mathbf{P}[\alpha = 0] + \mathbf{P}[\langle x, y \rangle \neq 0 \,|\, \alpha = 1] \, \mathbf{P}[\alpha = 1] \\
&\geq \mathbf{P}[x_k = 1] \, \mathbf{P}[\alpha = 0] + \mathbf{P}[x_k = 0] \, \mathbf{P}[\alpha = 1] \\
r &\geq \frac{1}{2} \mathbf{P}[\alpha = 0] + \frac{1}{2} \mathbf{P}[\alpha = 1] = \frac{1}{2},
\end{aligned}$$

   as desired.

2. If $AB \neq C$, then there is some row i such that the ith row of AB does not equal the ith row of C. Recall that $(ABx)_i$ and $(Cx)_i$ are the inner product of x with the ith row of AB and C, respectively. By the previous part, with probability at least $1/2$, $(ABx)_i \neq (Cx)_i$ (in which case $ABx \neq Cx$).

3. We repeat the algorithm from part 2 $O(\log n)$ times, answering that $AB = C$ iff $ABx = Cx$ for each randomly sampled x. If $AB = C$, then $ABx = Cx$ for all x and we correctly respond that $AB = C$. If $AB \neq C$, then for each x, we have $ABx = Cx$ with probability at most $1/2$. The probability that $ABx = Cx$ for all n trials is therefore $1/\operatorname{poly}(n)$. The total running time is $O(n^2)$, since it takes $O(n^2)$ time to compute ABx and Cx, and we rerun the algorithm $O(\log n)$ times.

[41] *Hint: Suppose for simplicity that the last coordinate of y is nonzero. It might help to imagine sampling the first $n-1$ bits and computing the partial sum $S_{n-1} = \sum_{i=1}^{n-1} x_i y_i$ first, before sampling x_n and adding $x_n y_n$. Formally your analysis may involve some conditional probabilities. (And what about the case where $y_n = 0$?)*

[42] Even if you haven't solved part 1 you may assume it to be true.

[43] *Hint:* How can you decrease the probability of error to 1/4? 1/8? 1/16?

Rubric.
4 pts. Part 1.
- **1 pts.** Generalizing from the special case where the last coefficient is nonzero by focus on the last nonzero coefficient.
- **1.5 pts.** Analysis conditioning on when the previous sum is zero.
- **1.5 pts.** Analysis conditioning on previous sum is nonzero.

3 pts. Part 2.
- **1.5 pts.** Recognizing that if $A \neq BC$, at least one row of the difference is a nonzero vector.
- **1.5 pts.** Correctly applying part 1 to this row.

3 pts. Part 3.
- **1 pt.** Repeating the algorithm $O(\log n)$ times.
- **1 pt.** Accept iff all independent trials accept.
- **.5 pts.** Show that probability of error becomes $1/\operatorname{poly}(n)$.
- **.5 pts.** Analyze the running time.

F.13 Homework 10

Exercise 23.1. Recall that a (binary) min-heap maintains a collection of elements and supports insertion and removing the minimum element in $O(\log n)$ worst-case time. Here we will develop and analyze a more sophisticated data structure called a *binomial heap*, which obtains $O(1)$ amortized time for insertion and $O(\log n)$ amortized time for deletion.

At a high-level, a binomial heap maintains a family of rooted trees over its elements, where each tree is in min-heap order. (Each node is less than or equal to any of its children). We will call each of these individual trees "heaps". The heaps in a binomial heap have a specific structure based on the following notion of "ranks".

- A rank 0 heap consists of a single node.
- For $i > 0$, a rank-i heap consists of two rank $i - 1$ heaps, where one heap is the child of the root of the other.

Two rank-i-heaps can be merged into a rank-$(i + 1)$-heap in constant time. Binomial heaps maintain the invariant that there is at most one heap of any rank i.

Now we give a high-level sketch of the `insert` and `remove-min` subroutines. Both have a postprocessing step that fixes the heap invariant. You will flesh out the implementation details in part 4 below.

`insert(`x`)`

/ High-level sketch; cf. part 4. */*

1. Create a rank 0 heap with node x.
2. Restore the heap invariant by repeatedly merging heaps of the same rank.

`remove-min`

/ High-level sketch; cf. part 4. */*

1. Scan the roots of all the heaps. Let H be the heap with the smallest root, let x be the root.
2. Remove H from the collection of heaps, and add the sub-heaps corresponding to the children of x.
3. Fix the heap invariant by repeatedly merging heaps of the same rank.
4. Return x.

Implement and analyze the binomial heap data structure via the following steps.

1. How big is a heap of rank i?
2. Suppose there are n elements. How many heaps are in the binomial heap?
3. Prove that for $i > 0$, a heap of rank i consists of a root node with i children, where the jth child is a heap of rank $j - 1$.

4. Flesh out the implementation details for `insert` and `remove-min`[44]. Prove that, for your implementation, `insert`(x) has $O(1)$ amortized time and `remove-min` has $O(\log n)$ amortized time.[45]

Solution to exercise 23.1.

Part 1. A heap of rank i has 2^i nodes. (A heap of rank i is twice the size of a heap of rank $i-1$.)

Part 2. All heaps have rank at most $\lfloor \log n \rfloor$ because a heap of rank $\lfloor \log n \rfloor + 1$ has more than n elements. We have at most one heap of each rank. So there are at most $\lfloor \log n \rfloor + 1$ heaps.

Part 3. We prove the claim by induction on $i \geq 1$.

For the base case $i = 1$: a rank-1 heap is formed from two rank-0 heaps by making the root of one a child of the root of the other. So the root has one child, which is a rank-0 heap. The claim holds: the first ($j = 1$) child has rank $j - 1 = 0$.

For the inductive step, suppose the claim holds for rank i. A rank-$(i+1)$ heap H is formed from two rank-i heaps H_1 and H_2 by making the root of H_1 a child of the root of H_2. By the inductive hypothesis, the root of H_2 already has i children, where the jth has rank $j - 1$. Adding H_1 gives the root a new $(i+1)$th child of rank $i = (i+1) - 1$. This matches the claim for rank $i + 1$.

Part 4. We maintain the heaps in a doubly-linked list in increasing order of rank. At each node, we also maintain the list of child heaps in increasing order of rank. Note that we can append a higher-rank heap as a child in constant time.

We merge heaps in two different scenarios. The first comes from an insertion, where we merge in a heap of rank 0. We "bubble up" through the heaps of our binomial heap until we don't have any conflicts. Because the heaps are in sorted order, we only look at the heaps we merge (plus one extra).

In the second scenario, we are merging two lists of heaps in sorted order. The first list is the overall list for the binomial heap; the second is the list of heaps from a previously separated heap. We "merge" the two lists from lowest rank to highest. Once we find two heaps of the same rank i, we remove both of them from their respective lists, merge them to create a heap of rank $i + 1$, and prepend this heap to the second list (the heaps we are trying to merge in). (At any point in time, the first list will always be a doubly linked list of heaps in strictly increasing order of rank, and the second list of heaps will be in nondecreasing order of rank.) In this scenario, the running time is proportional to $\log(n)$ plus the total number of individual merges.

To account for the merge, we define a potential Φ as the total number of heaps (including those that need to be merged in). In insertion, we create a new heap, increasing Φ by 1, and thereafter each individual merge is paid for by decreasing Φ. In `remove-min`, we generate at most $\log n$ heaps, increasing Φ by $\log n$, and all the merging thereafter is paid for by decreasing Φ. Thus insertion takes $O(1)$ amortized time and `remove-min` takes $O(\log n)$ amortized time.

[44] How do we store the heaps? How can we efficiently "merge" the heaps of the same rank? Etc.

[45] Note that one could also show that binomial heaps have $O(\log n)$-worst case time for both operations.

Exercise 27.5. The goal of this exercise is to show how to get constant time access for n keys with $O(n)$ space, using only universal hash functions. We will require the following fact that we ask you to prove.

1. Let $h : [n] \to [k]$ be a *universal* hash function. Show that for $k \geq n^2$, h has no collisions with probability $\geq 1/2$.

Now we describe the data structure. We first allocate an array $A[1..n]$ of size n. We have one universal hash function h_0 into $[n]$. If we have a set of (say) k collisions at an array cell $A[i]$, rather than making a linked list of length k, and we build another hash table, with a new universal hash function h_i, of size k^2, with no collisions (per part 1). (We may have to retry if there is a collision.) If the total size (summing the lengths of the first arrray and each of the second arrays) comes out to bigger than (say) $5n$, we try again.

2. For each $i = 1, \ldots, n$, let k_i be the number of keys that hash to the ith cell. We have

$$(\text{sum of array sizes of our data structure}) \leq n + \sum_{i=1}^{n} k_i^2.$$

   Show that[46]

$$\sum_{i=1}^{n} k_i^2 \leq n + O(\text{total \# of collisions (w/r/t } h_0)).$$

3. Show that

$$\mathbf{E}[\text{total \# of collisions (w/r/t } h_0)] \leq n/2.$$

4. Show that

$$\mathbf{P}[(\text{sum of all array sizes}) > Cn] \leq 1/2$$

   for some constant $C > 0$. ($C = 5$ is possible.)

Taken together, steps 1 to 3 above show that this approach will build a "perfect" hash table over the n keys in $O(n)$ space with probability of success at least $1/2$, using only universal hash functions. Even if it fails to work, we can then keep repeating the construction until it succeeds. This approach works better in *static settings*, when the set of keys is fixed.

Solution to exercise 27.5.

[46] Here a "collision" is an unordered pair of keys with the same hash. The $O(\cdots)$ means you can choose whatever constant you find convenient; 2 is possible.

Part 1. Any two items collides with probability at most $1/k \leq 1/n^2$. Taking the union bound over all $\binom{n}{2}$ pairs of items, the probability of having at least one collision is at most

$$\binom{n}{2}\frac{1}{n^2} = \frac{n(n-1)}{2n^2} \leq \frac{1}{2}.$$

Thus the probability of having no collisions is at most $1/2$.

Part 2. Suppose there are k_i elements in slot i. If $k_i \leq 1$, then there are no collisions. Otherwise, there are $\binom{k_i}{2} \geq (k_i^2 - 1)/2$ collisions among these elements. Thus

$$\sum_i k_i^2 \leq n + \sum_{i:k_i>1} \left(k_i^2 - 1\right) \leq n + \sum_{i:k_i>1} 2\binom{k_i}{2} = n + 2(\text{total \# of collisons}).$$

Part 3. Any two items collides with probability at most $1/n$. By linearity of expectation over all $\binom{n}{2}$ pairs of items, the expected number of collisions is

$$\binom{n}{2}\frac{1}{n} = \frac{n(n-1)}{2n} \leq \frac{n}{2}.$$

Part 4. We have

$$\mathbf{E}[\text{total size}] \leq n + \sum_i \mathbf{E}\left[k_i^2\right] \leq 2n + 2\,\mathbf{E}[\text{total \# of collisions with respect to } h_0] \leq 3n.$$

By Markov's inequality,

$$\mathbf{P}[\text{total size} > 6n] \leq 1/2,$$

as desired.

Rubric.
2 pts. Part 1
- **1 pt.** Single collision probability (invoking universality)
- **1 pt.** Union bound

3 pts. Part 2
- Important to distinguish $k_i = 1$ and $k_i > 1$.
- Needs to relate collisions at each slot to the number of collisions.

2 pts. Part 3
- **1 pt.** Analyze single item's collision.
- **1 pt.** Linearity of expectation of all pairs.

3 pts. Part 4
- **2 pts.** Compute $\mathbf{E}$[total size].
- **1 pt.** Apply Markov's inequality.

Exercise 27.8. In this exercise, we develop a refined analysis that can reduce the additive error substantially in (arguably realistic) settings when the stream is dominated by heavy hitters.

Let S denote the sum of frequency counts of all elements that are *not* ϵ-heavy hitters:

$$S = \sum_{e : p_e < \epsilon} f_e.$$

Note that $S \leq m$, and S might be much less than m when the stream is dominated by heavy hitters.

Show that, by increasing the parameter w (in the `count-min` data structure) by a constant factor, and still using universal hash functions, one can estimate the frequency of every element with additive error at most ϵS with high probability in $O(\log(n)/\epsilon)$ space.[47]

Solution to exercise 27.8. The high level idea is as follows. Fix an element e. Consider a single instance of the `hashed-counters` data structure with $w = \lceil 4/\epsilon \rceil$. We have additive error ϵS if the following holds:

1. e does not collide with any of the ϵ-heavy hitters.
2. The total noise from the remaining non-heavy hitters is at most ϵS.

We will show that each of the items above occur with probability at least $3/4$. If so, then by the union bound (with respect to the negated events), both occurs with probability $1/2$. That is, with probability at least $1/2$, we have additive error at most ϵS. We amplify to $1/\operatorname{poly}(n)$ probability of error by making $O(\log(n))$ copies.

We first analyze the probability that e does not collide with any of the heavy hitters. Let $D \subseteq [n]$ denote the set of heavy-hitters, we have $|D| \leq 1/\epsilon$. For a single heavy hitter $d \in D$, the probability that d collides with e is

$$\mathbf{P}[h(d) = h(e)] \leq \frac{1}{w}$$

since h is universal. Taking the union bound over all $\ell \leq 1/\epsilon$ heavy hitters,

$$\mathbf{P}[h(e) = h(d) \text{ for some } d \in D] \leq \frac{\ell}{w} \leq \frac{1}{4}.$$

Now we analyze the additive error from all non-heavy hitters. Let X denote the value of $A[h(e)]$ excluding the contributions from the heavy hitters. We want to show that $X - f_e \leq \epsilon S$ with probability at least $3/4$. By (a) linearity of expectation and (b) universality of h, we have

$$\mathbf{E}[X - f_e] \overset{(a)}{=} \sum_{\substack{c \neq e \\ c \notin D}} f_d \, \mathbf{P}[h(d) = h(e)] \overset{(b)}{\leq} \frac{1}{w} \sum_{\substack{c \neq e \\ c \notin D}} f_d = \frac{S}{w} \leq \frac{\epsilon S}{4}.$$

By Markov's inequality,

$$\mathbf{P}[X - f_e \geq \epsilon S] \leq \mathbf{P}[X - f_e \geq 4 \, \mathbf{E}[X - f_e]] \leq 1/4,$$

as desired.

[47] *Hint*: It might be helpful to think about the special case of $S = 0$.

Remark F.13. We analyze two events: not colliding with other ϵ-heavy hitters, and based on noise from non-heavy hitters. Above we analyzed them separately and them combined them via the union bound.

Since we are using limited randomness, you should be careful about how you are combining the two events. If you are using universal hash functions, then you cannot analyze one event conditional on the other – the other event "uses up" the randomness guarantee.

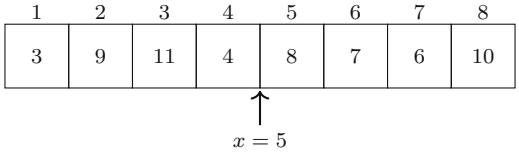
Rubric.
4 pts. Proving the element doesn't collide with any heavy hitter with probability (say) 1/4.
- **1 pt.** Observe that w is a constant factor bigger than the number of heavy hitters.
- **2 pts.** Analyze probability of colliding with a single heavy hitter (based on universality of hash function).
- **1 pt.** Union bound.

4 pts. Proving that the error from non-heavy hitters is ϵS with probability (say) 1/4.
- **3 pts.** Analyzing expected error from non-heavy hitters.
  - **1 pt.** Linearity of expectation.
  - **1 pt.** Universality of hash function.
  - **1 pt.** Getting the right bound. (Some constant factor less than ϵS).
- **1 pt.** Use Markov's inequality to bound the probability of large error.

1 pt. Combine the two events above with the union bound.
1 pt. Amplification.

F.14 Final

Exercise 1.14. Let $A[1..n]$ be an array of numbers, and x a number, such that $A[1] \leq x < A[n]$. (This also implies that $n > 1$.) We say that an index i "brackets" x if $A[i] \leq x < A[i+1]$. In the array below, the index $i = 4$ brackets $x = 5$ since $A[4] = 4 \leq 5 < 8 = A[5]$.



Consider the problem of computing either an index i that brackets x, or `nil` if no such index exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 1.14.

Recursive spec:

> `bracket(`$A[1..n]$`, `x`)`: Given an array of numbers $A[1..n]$ and a number x such that $A[1] \leq x < A[n]$, returns an index i that brackets x; i.e., $A[i] \leq x < A[i+1]$.

Recursive implementation.

> `bracket(`$A[1..n]$`, `x`)`
>
> 1. If $n = 2$ then return 1.
> 2. Let $i = \lceil \frac{n+1}{2} \rceil$.
> 3. If $A[i] > x$:
>   A. Return `bracket(`$A[1..i]$`, `x`)`
> 4. Else return $i - 1 +$ `bracket(`$A[i..n]$`,`x`)`

Solving the original problem. We call `bracket(`A`,`x`)`.

Running time. The running time is modeled by

$$T(n) = T(n/2) + O(1),$$

which gives $O(\log n)$.

Exercise 25.22. By now we have studied many algorithms that can find a path from s to t, and some variants of (s,t)-path problems that are NP-hard. But what if there's earthquakes?

Let $G = (V, E)$ be a directed graph and $s, t \in V$. We model earthquakes by assuming that each edge $e \in E$ gets destroyed independently with a fixed and known probability. For each edge $e \in E$, let p_e denote the probability of an edge e being destroyed in an earthquake.

We say that an (s,t)-walk *survives* an earthquake if all of its edges survive. The high-level goal is to compute the (s,t)-walk that is most likely to survive the earthquake.

1. (2 points) For an (s,t)-path with (distinct) edges $e_1, \dots, e_k$, what is the probability that the path survives an earthquake (as a function of the edge-wise probabilities p_e)?

2. (8 points) Consider the problem of computing the (s,t)-walk with the highest probability of surviving an earthquake. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 25.22.

Part 1. The path survives iff every edge along it survives, and edges fail independently. So the probability is

$$\prod_{i=1}^{k} (1 - p_{e_i}).$$

Part 2. We reduce to shortest paths, using the same logarithm trick as in the currency arbitrage problem (exercise 8.2).

Given $G = (V, E)$ with edge failure probabilities p_e for $e \in E$, we may discard any edge with $p_e = 1$ (which never survives) and assume that $p_e < 1$ for every edge. We also note that for any (s,t)-walk with a cycle, dropping the cycle leaves a walk with at least as high a probability of surviving, so it suffices to look for the best (s,t)-path.

Define edge lengths

$$\ell(e) = -\log(1 - p_e)$$

for each $e \in E$. Since $1 - p_e \in (0, 1]$, the edge lengths are nonnegative. We run Dijkstra's algorithm from s on G with edge lengths ℓ in $O(m + n \log n)$ time and return the shortest (s,t)-path.

We claim that for any (s,t)-path P,

$$\ell(P) = -\log \mathbf{P}[P \text{ survives}].$$

Equivalently, $\mathbf{P}[P \text{ survives}] = \exp(-\ell(P))$.

Let P be an (s,t)-path with edges $e_1, e_2, \dots, e_k$.

$$\ell(P) = \sum_{i=1}^{k} \ell(e_i) = \sum_{i=1}^{k} -\log(1 - p_{e_i}) = -\log \prod_{i=1}^{k} (1 - p_{e_i}) = -\log \mathbf{P}[P \text{ survives}].$$

Since $-\log$ is strictly decreasing on $(0, 1]$, minimizing $\ell(P)$ over (s,t)-paths is the same as maximizing $\mathbf{P}[P \text{ survives}]$. Combined with the earlier observation that it suffices to find the best (s,t)-path, the shortest (s,t)-path under ℓ is the (s,t)-walk most likely to survive an earthquake.

Exercise 14.2. Least romantic walk ever. You and your significant other have decided to go to a local nature preserve for a romantic walk. Together you picked a handful of trails that you would particularly like to traverse. But you're also a little pressed for the time because you need to practice exercises for the upcoming CS 381 final exam. Your goal is to plan for the shortest walk that covers all of your selected trails.

We model this with an undirected graph $G = (V, E)$ with positive edge lengths $\ell : E \to \mathbb{R}^+$, a starting vertex s, and a set of required edges $R \subseteq E$. Each edge models a trail, and R is the set of trails you want to visit. Call a walk *romantic* if it includes every edge in R. Consider the problem of computing the length of the shortest romantic walk starting and ending at s. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 14.2. We prove that the problem is NP-hard, by showing that a polynomial time algorithm for the shortest romantic walk gives a polynomial algorithm for undirected Hamiltonian cycle.

Consider an instance of undirected Hamiltonian cycle defined by an undirected graph $G = (V, E)$. We constructed an undirected, weighted graph $G' = (V', E')$ as follows. Starting from G,for each $v \in V$, add a new auxiliary vertex v' and an edge $\{v, v'\}$. Set $\ell(e) = 1$ for every edge G'. Pick any vertex $v_0 \in V$ as the starting vertex s. The required edges are the auxiliary edges $R = \{\{v, v'\} : v \in V\}$. (Coincidentally, the construction is not dissimilar from that of "almost independent set", exercise 13.4.)

We compute the shortest closed walk from s traversing every edge in R at least once, and declare that G has a Hamiltonian cycle iff this length is at most $3n$.

To prove correctness, we claim that G has a Hamiltonian cycle iff there is a romantic walk starting and ending at s with length at most $3n$.

Suppose G has a Hamiltonian cycle. We may assume it starts and ends at v_0: $v_0, v_1, \ldots, v_{n-1}, v_0$. Consider the walk in G':

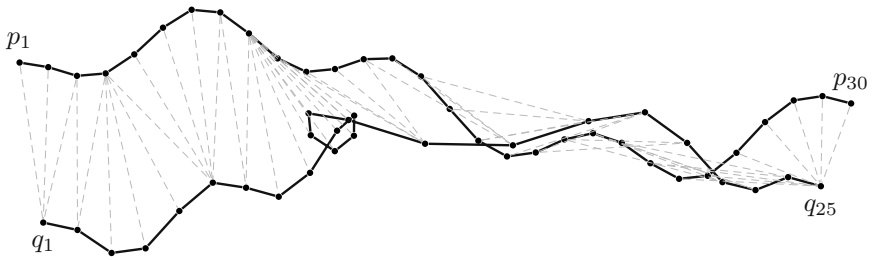
$$v_0, v_0', v_0, v_1, v_1', v_1, v_2, v_2', v_2, \ldots, v_{n-1}, v_{n-1}', v_{n-1}, v_0.$$

This walk takes the Hamiltonian cycle injects the cycle $v_i \to v_i' \to v_i$ at each vertex. It visits each required edge and has total length $= 2n + n = 3n$.

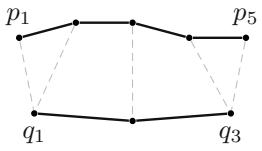
Suppose G' has a romantic walk W starting and ending at s of length $\leq 3n$. To traverse each required edge $\{v, v'\}$, where $v \in V'$, the walk must (a) visit every cycle, and (b) contain the cycle $v \to v' \to v$ for every vertex v. Removing these cycles leaves a walk in G, that start and ends at s, visits every vertex, and has at most $3n - 2n = n$ edges. In particular, the remaining walk is a Hamiltonian cycle.

Exercise 5.16. Walking the dog. You take your dog out for a walk, on a leash. You walk along a path with stops $p_1, p_2, \ldots, p_m$, while your dog walks along its own path with stops $q_1, q_2, \ldots, q_n$.

Each "step", either you advance to your next point p_i, the dog advances to its next point q_j, or you both advance to p_i and q_j, respectively. (Neither you nor the dog can go backwards.) Each step, the leash needs to be at least as long as the straight line segment between you (at some point p_i) and the dog (at some point q_j). The high-level goal is to minimize the length of the leash, coordinating your and your dog's steps as you traverse the paths.



In the example above, the line segment between you and the dog at each step is drawn as a dashed line. The leash length would be the maximum length over all these dashed lines. Specifically, you take the first two steps, then the dog takes a step, then you take a step, and then the dog takes the next four steps, and so forth. A simpler example is given on the right.



Consider the problem of computing the minimum possible leash length over all possible ways of stepping through the two paths. The input consists of a table of distances $D[1..m, 1..n]$, where $D[i, j]$ encodes the distance between p_i and q_j. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 5.16. We solve the problem with dynamic programming.

1. Recursive spec / induction hypothesis. $\mathsf{Leash}(i, j)$: For $1 \le i \le m$ and $1 \le j \le n$, $\mathsf{Leash}(i, j)$ returns the minimum leash length over walks that start with (p_i, q_j) and end with (p_m, q_n).

2. Recursive implementation.

$\underline{\mathsf{Leash}(i, j)}$

1. If $(i, j) = (m, n)$, return $D[m, n]$.
2. Otherwise, return $\max\{D[i, j], \min\{\mathsf{Leash}(i + 1, j + 1), \mathsf{Leash}(i + 1, j), \mathsf{Leash}(i, j + 1)\}\}$, where any $\mathsf{Leash}(i', j')$ with $i' > m$ or $j' > n$ is treated as $+\infty$.

3. Solving the original problem. We return $\mathsf{Leash}(1, 1)$.

4. Running time with caching / dynamic programming. We have $O(mn)$ subproblems and each subproblem takes $O(1)$ time, for a total of $O(mn)$.

Exercise 19.11. You work at Build-A-Graph, which is sort of like Build-A-Bear, but for graphs. Your customers come in with graphs and you find them subgraphs based on certain specifications about the vertex degrees.

Formally, the input consists of a directed graph $G = (V, E)$, and two lists $d_{\text{IN}}, d_{\text{OUT}} : V \to \mathbb{Z}_{\ge 0}$. The goal is to compute a subgraph $H = (V, F)$ where each vertex $v \in V$ has in-degree $d_{\text{IN}}(v)$ and out-degree $d_{\text{OUT}}(v)$, or declare that no such subgraph exists. (Here "subgraph" means that $F \subseteq E$. You can assume that $\sum_{v \in V} d_{\text{IN}}(v) = \sum_{v \in V} d_{\text{OUT}}(v)$.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution to exercise 19.11. We give a polynomial time algorithm with maximum flow. The problem is a generalization of the cycle cover problem, exercise 19.10.

If $\sum_{v \in V} d_{\text{IN}}(v) \neq \sum_{v \in V} d_{\text{OUT}}(v)$, declare "no subgraph exists", the in-degree and out-degree of any subgraph must have the same sum. Otherwise, let $T = \sum_v d_{\text{IN}}(v) = \sum_v d_{\text{OUT}}(v)$

We construct a flow network G' as follows.

1. Add a source $s^\star$ and sink $t^\star$.
2. For each $v \in V$, add two copies v_{OUT} and v_{IN} (no arc between them).
3. Add an arc $(s^\star, v_{\text{OUT}})$ with capacity $d_{\text{OUT}}(v)$, and an arc $(v_{\text{IN}}, t^\star)$ with capacity $d_{\text{IN}}(v)$.
4. For each arc $(u, v) \in E$, add an arc $(u_{\text{OUT}}, v_{\text{IN}})$ with capacity 1.

The source-arcs out of $s^\star$ form an $(s^\star, t^\star)$-cut of capacity T (as do the sink-arcs into $t^\star$), so the maximum flow value is at most T.

Compute a maximum integral $(s^\star, t^\star)$-flow f in G'. If $|f| < T$, declare "no subgraph exists". Otherwise $|f| = T$, and we form the output subgraph $H = (V, F)$ by taking F to be the edges of G whose corresponding arc in G' carries flow:

$$F = \{(u, v) \in E : f(u_{\text{OUT}}, v_{\text{IN}}) = 1\}.$$

Correctness. Suppose a valid $F \subseteq E$ exists. Set $f(u_{\text{OUT}}, v_{\text{IN}}) = 1$ iff $(u, v) \in F$, and saturate every source and sink arc. Conservation at v_{OUT} holds because v's out-degree in F equals $d_{\text{OUT}}(v)$, the source-arc capacity. Conservation at v_{IN} holds analogously. This is a feasible flow with value T, which by the cut bound above is the maximum.

Conversely, suppose the max flow has value T. By integrality, $f(u_{\text{OUT}}, v_{\text{IN}}) \in \{0, 1\}$ for every arc. Since the total source-arc capacity is exactly T, every source arc is saturated; similarly every sink arc is saturated. Conservation at v_{OUT} then forces v's out-degree in F to equal $d_{\text{OUT}}(v)$; conservation at v_{IN} forces v's in-degree to equal $d_{\text{IN}}(v)$. So F is a valid degree-constrained subgraph.

Running time. G' has $O(n)$ vertices and $O(n + m)$ arcs, all integer capacities, with max flow value $T \leq m$. By Ford-Fulkerson the running time is $O(Tm) = O\big(m^2\big)$.

Exercise 21.2. Let $G = (V, E)$ be an undirected graph with positive vertex weights $w : V \to \mathbb{R}_{>0}$. Recall that a *vertex cover* is a set $S \subseteq V$ such that every edge has at least one endpoint in S. Computing a minimum-weight vertex cover is NP-hard. Our goal is to produce a 2-approximation in polynomial time.

Consider the following *linear program* (LP):

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w(v)\, x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \text{ for each edge } \{u, v\} \in E, \\ & 0 \leq x_v \leq 1 \text{ for each vertex } v \in V. \end{aligned}$$

This LP relaxes the integrality of vertex covers: any vertex cover $C \subseteq V$ corresponds to the LP-feasible point $x_v = 1$ for $v \in C$, and $x_v = 0$ otherwise. However, an optimal LP solution may take fractional values. You may use as a black box that an optimal LP solution can be computed in polynomial time.

Let x^* denote an optimal solution to the LP, and define

$$S = \{v \in V : x^*_v \geq 1/2\}.$$

1. (4 points) Prove that S is a vertex cover.
2. (6 points) Prove that the total weight of S is at most twice the weight of any vertex cover. (I.e., S is a 2-approximation to the minimum-weight vertex cover.)

Solution to exercise 21.2.

Part 1. Let $\{u, v\} \in E$ be any edge. We have

$$\max\{x^*_U, x^*_v\} \geq \frac{1}{2}(x^*_u + x^*_v) \geq \frac{1}{2},$$

so either u or v is in S. It follows that S is a vertex cover.

Part 2. Since x^* solves a linear relaxation of vertex cover, we have

$$\sum_v w_v x^*_v \leq \mathrm{OPT}\,.$$

Meanwhile, each $v \in S$ has $x^*_v \geq 1/2$. We have

$$\sum_{v \in S} w(v) \leq \sum_{v \in S} 2x^*_v w(v) \leq 2 \sum_{v \in V} x^*_v w(v) \leq 2\,\mathrm{OPT}\,.$$

That is, S has weight at most twice the minimum-weight vertex cover.

Appendix G

Practice problems for exams

In my opinion, practice problems are the best way to increase your comfort level with the material. We recommend the exercises listed at the end of each chapter. All word problems from past exams are listed as exercises. We also recommend going back and re-solving problems we have done before in homework.

G.1 Mapping to other textbooks

The topics we covered are pretty standard for algorithms although the order of presentation and points of emphasis may be different.[1] Here I will list what appear to be the most relevant parts of two textbooks by Erickson [Eri19] and Kleinberg and Tardos [KT06], just based on their table of contents.[2] In general, exercises from these chapters might be good practice. These textbooks also have some examples that I have not presented and these examples might be helpful as well.

In terms of [Eri19], we covered

1. Sections 1.4, 1.6 – 1.9 on merge sort, recursion trees, and divide-and-conquer.
2. Chapter 3, on dynamic programming.
3. Chapter 5, basic graph algorithms.
4. Chapter 6, depth-first search.
5. Chapter 7, spanning trees.
6. Chapter 8, shortest paths.

[1] The textbooks also have different perspectives and organizational principles from one another.

[2] Of course I may miss some things and can update the list if people point things out. This list will also be updated for each exam.

7. Chapter 10, max-flow and min-cut.
8. Chapter 11, applications of flows and cuts.
9. Chapter 12, NP-hardness
10. Sections 9.6 and 9.8, on all pairs shortest paths via dynamic programming.

Erickson's "Extended dance remix" and "director's cut" (https://jeffe.cs.illinois.edu/teaching/algorithms/) has notes on additional topics, including:

1. Chapter A is on Fast Fourier Transforms.
2. Chapter 9 is on amortized analysis
3. Chapters 1, 2, 5, and 6 cover different topics in randomized algorithms that overlap with our discussions.

In terms of [KT06], we covered

1. Chapter 3 (on graphs).
2. Secton 4.4–4.6 (on Dijkstra's algorithm)[3]
3. Chapter 5 (on divide-and-conquer).
4. Chapter 6 (on dynamic programming).
5. Chapter 7 (on network flow)
6. Chapter 8 on NP and computational intractablity (partly, more comments below).
7. Chapter 13 is on randomized algorithms.

G.2 Problems from "Jeff's notes" [Eri19]

Here I am selecting some problems from [Eri19] that might be good practice. I've chosen them to be diverse and I am restricting the number of problems I select for each chapter. However you might want to look at other problems too which are fun[4] and you should be equipped to solve most of them.

I am going in order of chapters in [Eri19], which is a different order of topics from our lectures.

[3] [KT06] calls Dijkstra's algorithm a "greedy algorithm" but I have not.
[4] Maybe only in my screwed up sense of the word fun.

G.2.1 Chapter 1

6, 11, 14, 16, 22, 32.

G.2.2 Chapter 3

This chapter is on dynamic programming. All of the problems in this chapter are great, but here are just a few (chosen primarily to form a diverse set).

2, 5.a, 5.b, 6.b, 12, 32, 42, 47, 48.

When practicing, the most important thing is identifying the right recursive spec / inductive hypothesis. Then implement some recursive code to confirm that your inductive hypothesis can be made algorithmic. The remaining steps are relatively easy. You might be able to fly through a lot of problems just focusing on the first two steps in particular since not much writing is actually required to do this.

It is worthwhile to actually write the sentence for the recursive spec out in full and one paper. It is easy to deceive yourself into thinking you're correct when you haven't really forced yourself to completely express your ideas.

G.2.3 Chapter 5

11, 12, 13, 20, 25.

G.2.4 Chapter 6

3, 9, 15, 17, 21.

G.2.5 Chapter 7

2, 4, 5, 6

G.2.6 Chapter 8

14, 15, 20, 22, 25.

G.2.7 Chapter 10

1, 2, 3, 7, 8,

G.2.8 Chapter 11

4, 5, 6, 8, 10, 19,

G.2.9 Chapter 12

1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16.a, 16.e, 16.f, 20, 22, 24, 25, 28.a, 29, 30, 31, 32, 39.

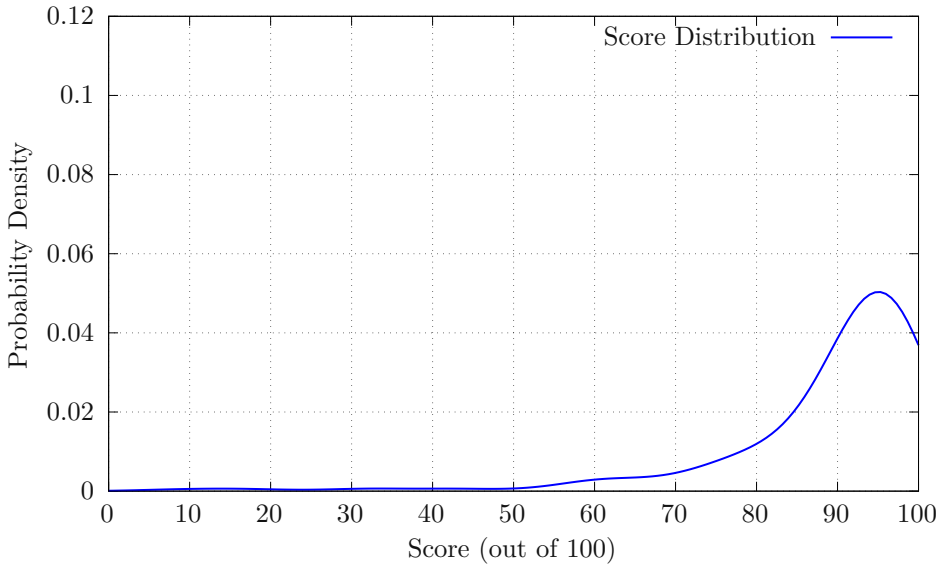
At least one problem requires the fact that one can compute an *Eulerian circuit* (if it exists) in polynomial time. An Eulerian circuit is a walk in a directed graph that starts and ends at the same vertex and traverses each edge exactly once. It is known that a strongly connected graph has an Eulerian circuit iff every vertex has in-degree equal to its out-degree. See also the Wikipedia article. (You may have seen Eulerian tours before in your discrete math class but I am not sure.)

Appendix H

Aggregate statistics

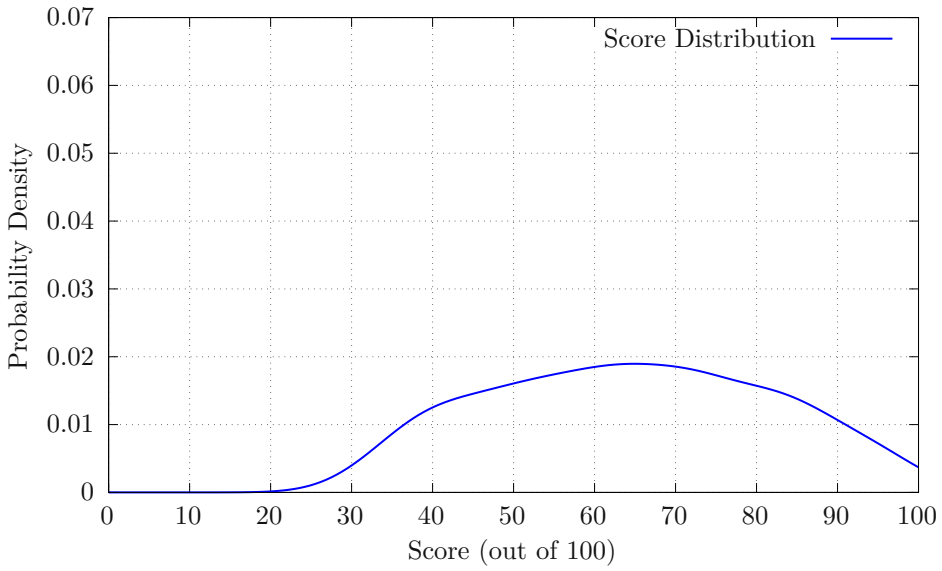
H.1 Homework

Below is a kernel density estimation (KDE) plot of homework scores based on word problems from HW0 through HW10 and MCQs from HW0 through HW10 (44 scores total). We drop the lowest $\lceil 44/5 \rceil = 9$ scores. The median is 93.5, the average is 89.3, and the standard deviation is 13.8 (rounding to the nearest .1).



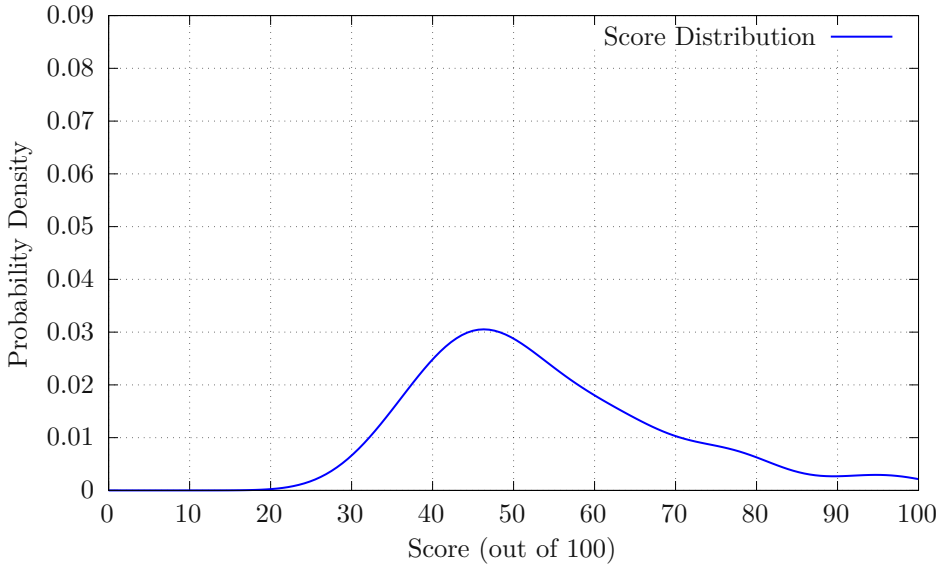
H.2 Midterm 1

284 students took the first midterm. Below is a KDE plot of all the scores (properly weighted, and scaled out of 100). The median is 64.1, the average is 64.4, and the standard deviation is 17.4.



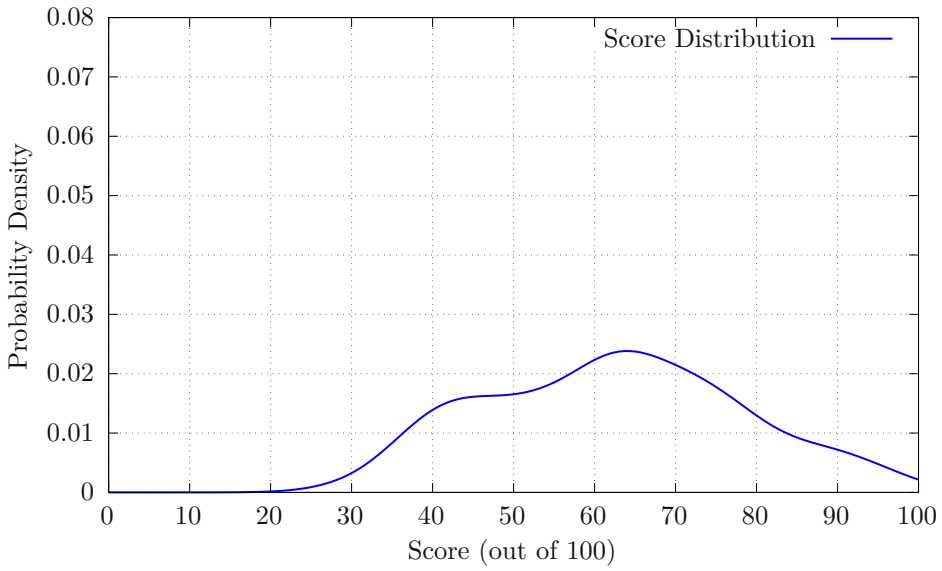
H.3 Midterm 2

275 students took the second midterm. Below is a KDE plot of all the scores (properly weighted, and scaled out of 100). The median is 50.8, the average is 54.1, and the standard deviation is 15.1.



H.4 Final

268 students took the final exam. Below is a KDE plot of all the scores (properly weighted, and scaled out of 100). The median is 62.6, the average is 62.3, and the standard deviation is 15.8.

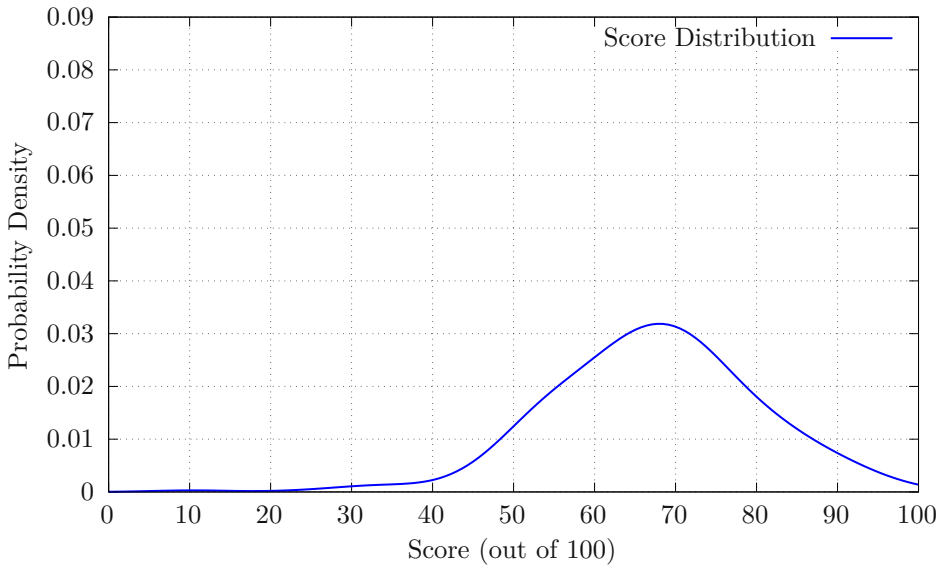


H.5 Total

Based on the syllabus, the total score is computed at the end of the semester as

$$.25(\text{homework}) + .20(\text{midterm 1}) + .25(\text{midterm 2}) + .30(\text{final}).$$

If a student missed an exam with an excused absence, that exam's weight was redistributed proportionally over the remaining exams. Below is a KDE plot of the total scores for the 272 students who completed the course. The median is 67.6, the average is 67.3, and the standard deviation is 12.6.



Appendix I

CS 381: Syllabus, Policies, and Procedures

Welcome to CS381, an undergraduate course on fundamental algorithms. Please see the schedule (Page 3) for a tentative list of topics. The class will be similar to the Spring 2025 undergraduate course (cf. https://fundamentalalgorithms.com/fas24). For lecture times and staff information, see on page ??.

Please take advantage of class time, PSOs, and office hours to ask questions or express any concerns. Please reserve email only for true emergencies, which I do not expect to arise.

The lectures are accompanied by lecture notes (usually one chapter per lecture) and you are expected to be informed of their contents. I (Kent) am planning to record the lectures and put them online (say, over the weekend, on Boilercast and on Youtube), as well as upload handwritten slides from the lectures. Links to these resources are provided at the end of each chapter. I caution that the recordings are meant to supplement in-class lectures and should not be regarded as a substitute.

If you are unable to register for the class because it is full, just wait. Historically, slots have always opened up. You can add yourself to gradescope and submit homework in the meantime.

I.1 Textbooks

No textbook is strictly required as lecture notes are provided. The following textbooks are all very good, and each have their own merits. At the end of each lecture/chapter, we give pointers to the relevant sections in these textbooks. (You will probably have to get to know all of them over the course of your computer science careers, anyway.) We will also provide pointers to other references on the web.

1. Jeff Erickson. *Algorithms*. 2019. URL: http://jeffe.cs.illinois.edu/teaching/algorithms/.

2. Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006. As a popular textbook, you may be able to borrow a copy from a friend.

3. Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008. Draft version available at https://book.huihoo.com/pdf/algorithms/.

4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms. Online access available via the Purdue Library.

There are several classes across the country that have put lecture videos online. You may find the following videos helpful.

1. Erik Demaine and Srini Devadas. *6.006: Introduction to Algorithms*. Fall 2011. Course website at http://courses.csail.mit.edu/6.006/fall11/staff.shtml. Video lectures posted at https://youtube.com/playlist?list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb.

2. Erik Demaine, Srini Devadas, and Nancy Lynch. *6.046/18.410: Design and Analysis of Algorithms*. Spring 2015. Course website at https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/. Video lectures posted at https://youtube.com/playlist?list=PLUl4u3cNGP6317WaSNfmCvGym2ucw3oGp.

I.2 Correspondence

The course website is

www.fundamentalalgorithms.com,

where this document is posted.

Email. Please reserve emails to the instructor for true emergencies. Please use class time or office hours to ask questions or express concerns.

EdStem. There is an EdStem for the course at the following address.

https://edstem.org/us/join/fsaryR

The point of Piazza is to foster discussion among the students. The TA's will monitor Piazza on weekdays (regularly but not continuously) during reasonable hours.

I.3 Grading

- 25% Homework
- 20% Midterm 1
- 25% Midterm 2
- 30% Final

We compute numerical scores based on the weighting above (as a fractional value between 0 and 1), and then we curve the grades.

I.4 Exams

The midterms and final are each one part multiple choice and one part word problems. We have reserved classrooms through the school to give students an extended period of time for the tests.

There are generally no makeup or alternative exams.

I.5 Homework

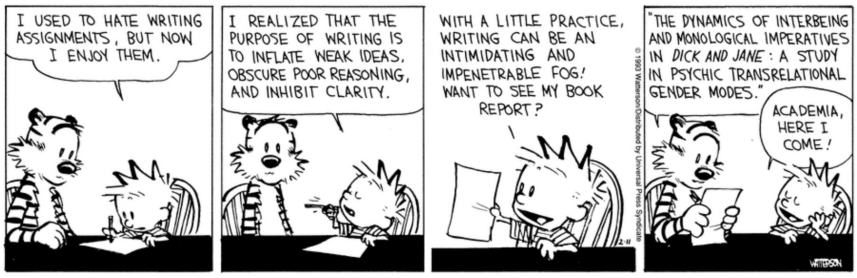
This course has regular homework, generally due every week. Homeworks will be due at 11:59PM on Thursday nights.

Typesetting. Homework submissions that are not typeset in LaTeX or equivalent will not be graded. Some tips on typesetting are listed below. A simple Overleaf template is set up at https://www.overleaf.com/read/fczzqbftywcp.

On writing. The onus is on the student to make the arguments in their solution clear, and points will be docked if the grader cannot easily verify that the solution is correct. The class is as much about *communicating complicated ideas* as solving problems and applying techniques. Particularly clear exposition may be selected as homework solutions which is rewarded with extra credit (see below).

Here are some articles about writing:

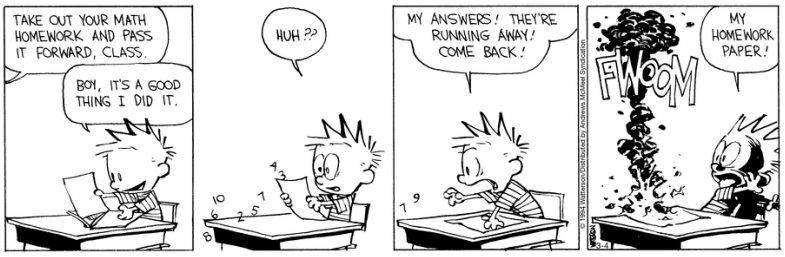
- Terry Tao: https://terrytao.wordpress.com/advice-on-writing-papers/.
- Cormac McCarthy: https://www.nature.com/articles/d41586-019-02918-5. (Accessible via school library.)



Gradescope. The word problems will be collected online at gradescope.com. The multiple choice questions will be posted on gradescope.com. as well. If you are registered for the course on BrightSpace, then you should have been automatically added to gradescope. Otherwise you can add yourself with the code 8DXYPZ.

Collaboration. Collaboration is allowed and interaction among students is encouraged. Currently we are allowing up to three students per submission. Please also indicate any other students (outside your group) that you may have worked on the problems with.

Dropping scores. In the overall homework grade, the bottom one-fifth of word problem scores and multiple choice assignments will be dropped. More precisely, if there are n total world problems and multiple choice assignment assigned in homework, then the $\lceil n/5 \rceil$ lowest scores will be dropped. (To be clear: each weekly multiple choice assignment is worth 10 points, just like a word problem, and counted just like a word problem.) This is largely to help cover the arbitrary exceptions that arise throughout a semester.



Late policy. For word problems, we have a simple late policy where you can submit up to 4 days late (on Monday), at a cost of 35% of the total points.

There are no exceptions to the late policy. We expect the $\lceil n/5 \rceil$-dropped scores to absorb most scenarios that arise; besides, 35% off is not the end of the world.

In rare circumstances accompanied by documentation by the Dean of Students we may instead give a 0/0 for all problems on that assignment.

Solutions. The staff will select exemplary submissions and publish them as solutions. If you have a strong preference to be excluded from consideration for a particular homework problem, please indicate it clearly and explicitly at the top of your submission (for each problem). For this reason, *please leave your student ID off of your submission.* If you have a strong preference to be anonymous if your homework is selected, please indicate that on your document.

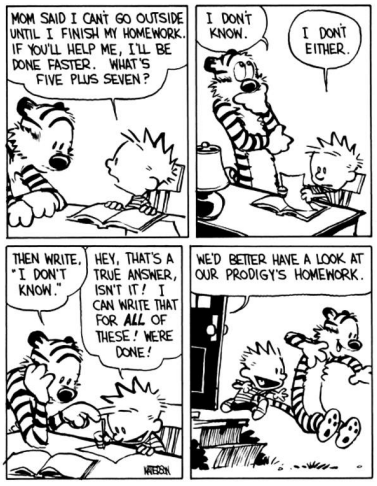
Selected solutions will get 10% extra credit.

We plan to put up the solutions quickly after the homework is collected.

We do *not* publish solutions to have PSOs.

Resubmitting homework. You might have noticed that there is both a late policy and a plan to post solutions very soon after the submission deadline. You can take advantage of this by comparing the answer key to your own submission, and possibly resubmit your homework late even with the benefit of the answer key. If you do use the posted answer key in a resubmission, we expect you to cite it accordingly, and still express the solution in your own words.

IDK. One may simply write "I don't know" or "IDK" and automatically get 25% of the possible points (for any problem or subproblem).



Regrades. Regrade request must be initiated within one week of the grades being returned.

Typesetting tips.

- The standard for typesetting mathematical and scientific articles is LaTeX. Even if you do not know LaTeX now, you probably have to learn it sooner or later (and certainly if you pursue graduate studies).

- The instructor uses `emacs` to write LaTeX, but any editor will do. There is also a website called `overleaf.com` for typesetting LaTeX.
- Alternatively, the open-source software `marktext` allows one to write LaTeX within a markdown document, which is particularly easy to use.
- `LyX` is another popular latex editor that is WYSIWYG.
- There are several apps for scanning documents (e.g., when inserting pictures) that are much better than taking a photo. The instructor uses `scanbot`, and other popular apps include `microsoft office lens`, `camscanner`, and `evernote scannable`.

I.6 Academic integrity

Behavior consistent with cheating, copying, and academic dishonesty is not tolerated. Depending on the severity, this may result in a zero score on the assignment or exam, and could result in a failing grade for the class or even expulsion. Purdue prohibits "dishonesty in connection with any University activity. Cheating, plagiarism, or knowingly furnishing false information to the University are examples of dishonesty." (Part 5, Section III-B-2-a, University Regulations) Furthermore, the University Senate has stipulated that "the commitment of acts of cheating, lying, and deceit in any of their diverse forms (such as the use of substitutes for taking examinations, the use of illegal cribs, plagiarism, and copying during examinations) is dishonest and must not be tolerated. Moreover, knowingly to aid and abet, directly or indirectly, other parties in committing dishonest acts is in itself dishonest." (University Senate Document 7218, December 15, 1972). You are expected to read both Purdue's guide to academic integrity (`http://www.purdue.edu/purdue/about/integrity_statement.html`) and Prof. Gene's Spafford's guide (`http://spaf.cerias.purdue.edu/integrity.html`) as well. You are responsible for understanding their contents and how it applies to this class.

I.7 Posting Class Material

Posting material associated with this class (e.g., solutions to homework sets or exams) without the written permission of the instructor is forbidden and may be a violation of copyright.

I.8 Purdue's Honor Pledge

As a boilermaker pursuing academic excellence, I pledge to be honest and true in all that I do. Accountable together - we are Purdue. https://www.purdue.edu/provost/teachinglearning/honor-pledge.html.

I.9 Grief Absence Policy

Purdue University recognizes that a time of bereavement is very difficult for a student. The University therefore provides the following rights to students facing the loss of a family member through the Grief Absence Policy for Students (GAPS). According to GAPS Policy, students will be excused for funeral leave and given the opportunity to earn equivalent credit and to demonstrate evidence of meeting the learning outcomes for missed assignments or assessments in the event of the death of a member of the student's family.

I.10 Conduct and Courtesy

Students are expected to maintain a professional and respectful classroom environment. This includes: silencing cellular phones, arriving on time for class, speaking respectfully to others and participating in class discussion. You may use non-disruptive personal electronics for the purpose class participation (e.g., taking notes).

I.11 Students with Disabilities

Purdue University is required to respond to the needs of the students with disabilities as outlined in both the Rehabilitation Act of 1973 and the Americans with Disabilities Act of 1990 through the provision of auxiliary aids and services that allow a student with a disability to fully access and participate in the programs, services, and activities at Purdue University. If you have a disability that requires special academic accommodation, please make an appointment to speak with the instructor within the first three (3) weeks of the semester in order to discuss any adjustments.

It is the student's responsibility to notify the Disability Resource Center (http://www.purdue.edu/drc) of an impairment/condition that may require accommodations and/or classroom modifications. We cannot arrange special accommodations without confirmation from the Disability Resource Center.

I.12 Emergencies

In the event of a major campus emergency, course requirements, deadlines and grading percentages are subject to changes that may be necessitated by a revised semester calendar or other circumstances beyond the instructor's control. Relevant changes to this course will be posted onto the course website and/or announced via email. You are expected to read your purdue.edu email on a frequent basis. Emergency Preparedness: Emergency notification procedures are based on a simple concept: If you hear an alarm inside, proceed outside. If you hear a siren outside, proceed inside. Indoor Fire Alarms are mean to stop class or research and immediately evacuate the building. Proceed to your Emergency Assembly Area away from building doors. Remain outside until police, fire, or other emergency response personnel provide additional guidance or tell you it is safe to leave. All Hazards Outdoor Emergency Warning sirens mean to immediately seek shelter (Shelter in Place) in a safe location within the closest building. "Shelter in place" means seeking immediate shelter inside a building or University residence. This course of action may need to be taken during a tornado, a civil disturbance including a shooting or release of hazardous materials in the outside air. Once safely inside, find out more details about the emergency. Remain in place until police, fire, or other emergency response personnel provide additional guidance or tell you it is safe to leave. In both cases, you should seek additional clarifying information by all means possible: Purdue Home page, email alert, TV, radio, etc. Review the Purdue Emergency Warning Notification System multi-communication layers at http://www.purdue.edu/ehps/emergencypreparedness/warning-system.html. Please review the Emergency Response Procedures at https://www.purdue.edu/emergencypreparedness/flipchart/index.html. Please review the evacuation routes, exit points, emergency assembly area and shelter in place procedures and locations for the building. Video resources include a 20-minute active shooter awareness video that illustrates what to look for and how to prepare and react to this type of incident. See http://www.purdue.edu/securepurdue/police/video/

I.13 Violent Behavior Policy

Purdue University is committed to providing a safe and secure campus environment for members of the university community. Purdue strives to create an educational environment for students and a work environment for employees that promote educational and career goals. Violent Behavior impedes such goals. Therefore, Violent Behavior is prohibited in or on any University Facility or while participating in any university activity.

I.14 Mental Health and Wellness

If you find yourself beginning to feel some stress, anxiety and/or feeling slightly overwhelmed, try Therapy Assistance Online (TAO) (https://www.purdue.edu/caps/students/resources/self-help/digital/tao.php), a new web and app-based mental health resource available courtesy of Purdue Counseling and Psychological Services (CAPS). TAO is available to students, faculty, and staff at any time.

If you need support and information about options and resources, please contact or see the Office of the Dean of Students (www.purdue.edu/odos). Call 765-494-1747. Hours of operation are M-F, 8 am- 5 pm.

If you find yourself struggling to find a healthy balance between academics, social life, stress, etc. sign up for free one-on-one virtual or in-person sessions with a Purdue Wellness Coach at RecWell (https://www.purdue.edu/recwell/fitness-wellness/wellness/one-on-one-coaching/wellness-coaching.php). Student coaches can help you navigate through barriers and challenges toward your goals throughout the semester. Sign up is completely free and can be done on BoilerConnect. If you have any questions, please contact Purdue Wellness at evans240@purdue.edu.

If you're struggling and need mental health services: Purdue University is committed to advancing the mental health and well-being of its students. If you or someone you know is feeling overwhelmed, depressed, and/or in need of mental health support, services are available. For help, such individuals should contact Counseling and Psychological Services (CAPS) (https://www.purdue.edu/caps/) at 765-494-6995 during and after hours, on weekends and holidays, or by going to the CAPS office on the second floor of the Purdue University Student Health Center (PUSH) during business hours.

Purdue University is committed to advancing the mental health and well-being of its students. If you or someone you know is feeling overwhelmed, depressed, and/or in need of support, services are available. For help, such individuals should contact Counseling and Psychological Services (CAPS) at (765) 494-6995 and http://www.purdue.edu/caps/ during and after hours, on weekends and holidays, or through its counselors physically located in the Purdue University Student Health Center (PUSH) during business hours.

I.15 Health in general

In general, if medical conditions prohibit you from participating in the class, please be proactive in seeking professional medical care. The link to the Purdue University Student Health Center (PUSH) is listed below:

https://www.purdue.edu/push/.

In cases falling under excused absence regulations, the student or the student's representative should contact or go to the Office of the Dean of Students (ODOS, https://www.purdue.edu/advocacy/students/absence-policies.html) website to complete appropriate forms for instructor notification. Under academic regulations, excused absences may be granted by ODOS for cases of grief/bereavement, military service, jury duty, parenting leave, or emergent or urgent care medical care.

No one on the teaching staff is qualified to make any kind of diagnosis, and we rely the dean of students (who are suppose to be able to handle medical situations) to document serious medical cases and provide us with instructions when applicable. We have recourse policies in place for documented illness.

I.16 Basic needs and security

Any student who faces challenges securing their food or housing and believes this may affect their performance in the course is urged to contact the Dean of Students for support. There is no appointment needed and Student Support Services is available to serve students 8 a.m.-5 p.m. Monday through Friday. Considering the significant disruptions caused by the current global crisis as it relates to COVID-19, students may submit requests for emergency assistance from the Critical Need Fund (https://www.purdue.edu/odos/resources/critical-need-fund.html).

I.17 Nondiscrimination

Purdue University is committed to maintaining a community which recognizes and values the inherent worth and dignity of every person; fosters tolerance, sensitivity, understanding, and mutual respect among its members; and encourages each individual to strive to reach his or her own potential. In pursuit of its goal of academic excellence, the University seeks to develop and nurture diversity. The University believes that diversity among its many members strengthens the institution, stimulates creativity, promotes the exchange of ideas, and enriches campus life. Purdue University prohibits discrimination against any member of the University community on the basis of race,

religion, color, sex, age, national origin or ancestry, marital status, parental status, sexual orientation, disability, or status as a veteran. The University will conduct its programs, services and activities consistent with applicable federal, state and local laws, regulations and orders and in conformance with the procedures and limitations as set forth in Executive Memorandum No. D-1, which provides specific contractual rights and remedies.

I.18 Privacy

The Federal Educational Records Privacy Act (FERPA) protects information about students, such as grades. If you apply for a job and wish to use the instructor as a reference, you should tell the instructor beforehand. Otherwise, the instructor cannot say anything about you to a prospective employer who might call. The instructor is happy to provide references and to write letters of recommendation for his students as needed.

I.19 Netiquette

We want to foster a safe online learning environment. All opinions and experiences, no matter how different or controversial they may be perceived, must be respected in the tolerant spirit of academic discourse. You are encouraged to comment, question, or critique an idea, but you may not attack an individual. Our differences, some of which are outlined in the University's nondiscrimination statement below, will add richness to this learning experience. Please consider that sarcasm and humor can be misconstrued in online interactions and generate unintended disruptions. Working as a community of learners, we can build a polite and respectful course ambience. Please read the Netiquette rules for this course:

- Monitor how much space/time you are taking up in any discussion. Give other students the opportunity to join in the discussion.
- Do not use offensive language. Present ideas appropriately.
- Be cautious in using Internet language. For example, do not capitalize all letters since this suggests shouting.
- Avoid using vernacular and/or slang language. This could lead to misinterpretation.
- Keep an "open-mind" and be willing to express even your minority opinion.

- Think and edit before you push the "Send" button.
- Seek and take in feedback from others; learning from other people is an important life skill.

I.20 Changes to the syllabus

This syllabus is subject to change and changes will be announced appropriately.

Appendix J

PSO

J.1 Week 0

Exercise 1.1. Use recursion trees to solve the following recurrences.

1. $T(n) = 3T(n/3) + 6n$ and $T(1) = 2$.
2. $T(n) = 2T(n/3) + 4n$ and $T(1) = 7$.
3. $T(n) = 4T(n/3) + n$ and $T(1) = 11$.

Exercise J.1. Problems 1 and 2 describe a search problem on a nonempty array $A[1..n]$ of comparable elements.[1] For each of these problems, design a recursive algorithm that solves the search problem as fast as possible.[2] Briefly analyze the running time of each algorithm.[3]

1. Let $A[1..n]$ be in a *rotated* sorted order. That is, there exists an index $i \in [n]$, called the *rotation index*, such that the concatenation of $A[i..n]$ and $A[1..i-1]$ is sorted in increasing order. (One can think of $A[1..n]$ as being sorted initially, and then rotated cyclically to the right by $i-1$ slots). The goal is to compute the unknown rotation index i. For simplicity, you may assume all the elements are distinct.
2. We say that $A[1..n]$ is a *mountain* if there exists an index $i \in [n]$, called the *peak*, such that $A[1..i]$ is sorted in increasing order and $A[i+1..n]$ is sorted in decreasing order. Given a mountain $A[1..n]$, find the peak. For simplicity, you may assume all elements are distinct.

Exercise 1.7. Let $A[1..n] \in \mathbb{R}^n$ be an array of n numbers. An *inversion* is a pair of numbers out of increasing order; more precisely, a pair of indices $i, j \in [n]$ such that $i < j$ and $A[i] > A[j]$. Design and analyze an algorithm (as fast as possible) for counting the number of inversions in A.

Exercise 1.8. Two points $(a, b), (c, d) \in \mathbb{R}^2$ are *comparable* if either (a) $a \leq c$ and $b \leq d$, or (b) $c \leq a$ and $d \leq b$.

Given a set of n points $(a_1, b_1), \ldots, (a_n, b_n) \in \mathbb{R}^2$, consider the problem of computing the number of pairs of points that are comparable. You may assume for simplicity that all points are distinct. Design and analyze an algorithm (as fast as possible) for counting the number of comparable pairs of points.[4]

[1] Given two elements x and y, you can query if $x < y$, $x > y$, or $x = y$.

[2] This includes a recursive spec, and pseudocode implementing the recursive spec.

[3] For the sake of brevity we do not ask for an explicit analysis of correctness. We will treat your recursive spec as the induction hypothesis of a proof by induction, and fill in the rest.

[4] *Hint:* As a warmup, try to instead compute the number of *incomparable* pairs of points. (A pair

Exercise 1.11 Let T be a tree on n vertices. A *centroid* of T is a vertex v whose removal breaks T into subtrees of at most $n/2$ vertices. A tree can have up to two centroids.

1. Give a linear time algorithm taking as input a tree and returning a centroid. Your proof of correctness doubles as a proof that every tree has a centroid.[5]

Centroids can be used as a sort of "pivot" in divide-and-conquer algorithms over trees. Design and analyze an algorithm for each of the following problems, leveraging centroids and recursion.[6]

2. Design and analyze an algorithm that computes, for a given $k \in \mathbb{N}$, the number of pairs of vertices that are k edges apart in T.

Exercise 2.2 Design a recursive algorithm implementing the specification. (No proof or analysis is needed.)[7]

1. `subset-sum`$(x_1, \ldots, x_n,\ T)$: Given n integers $x_1, \ldots, x_n \in \mathbb{Z}$, and an additional integer T, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = T$, and `false` otherwise.

2. `min-vertex-cover`$(G = (V, E))$: Given an undirected graph $G = (V, E)$, return the minimum size of any vertex cover of G.

   (A vertex cover is a set of vertices $S \subseteq V$ that includes at least one endpoint of every edge $e \in E$.)

of points is incomparable if is not comparable. E.g., (x_1, y_1) and (x_2, y_2) are incomparable if $x_1 < x_2$ and $y_1 > y_2$.) You may want to do exercise 1.7 before this one.

[5]For sake of the PSO, you can assume this as a black box and focus on the following problem.

[6]A couple of these problems can be solved by other means; for pedagogical reasons, we still encourage you to solve them via centroids.

[7]Your algorithms will not be very fast, but their correctness should follow from an induction argument using the recursive spec as the induction hypothesis.

You are welcome to describe your algorithm using high level steps as long as they are clearly easy to implement and would take linear time; e.g., 'let G_1 be the graph obtained by removing such and such vertices and such and such edges." See the sample solutions on page 60.

Some of these problems have well-known efficient algorithms. Efficiency is besides the point; the goal here is to rapidly get some practice thinking recursively.

Lastly, do not change the recursive spec.

Exercise $n + 1$. Let S be a set of n straight lines in the plane, partitioning the plane into a number of regions that is quadratic in n. For simplicity, we assume that the lines are in "general position": no three lines meet at a point, and that no two lines are parallel. A *two-coloring* colors each region one of two colors, such that every two "adjacent" regions that share an edge have different colors. Design and analyze an algorithm, as fast as possible, computing a 2-coloring of the regions of S.[8]

[8] *Hint:* Suppose you've two-colored all the regions induced by a subset of $n-1$ of the lines. The nth line splits some of these regions in half, and each of these "split regions" violates the coloring requirements... now what?

J.2 Week 1

Exercise J.2. For the following problems find the smallest values of a, b such that $S(n) = O(n^a \log^b(n))$.

$$S(n) = \sum_{i=1}^{n} (4/5)^i \tag{J.1}$$

$$S(n) = \sum_{i=1}^{n} \frac{3}{i} \tag{J.2}$$

$$S(n) = \sum_{i=1}^{n^2} \frac{7}{i^2} \tag{J.3}$$

$$S(n) = \sum_{i=1}^{\log_2(n)} (5/4)^i \tag{J.4}$$

$$S(n) = \sum_{i=1}^{n} \sum_{j=1}^{i^2} j \tag{J.5}$$

Exercise 1.7. Let $A[1..n] \in \mathbb{R}^n$ be an array of n numbers. An *inversion* is a pair of numbers out of increasing order; more precisely, a pair of indices $i, j \in [n]$ such that $i < j$ and $A[i] > A[j]$. Design and analyze an algorithm (as fast as possible) for counting the number of inversions in A.

Exercise 1.8. Two points $(a, b), (c, d) \in \mathbb{R}^2$ are *comparable* if either (a) $a \leq c$ and $b \leq d$, or (b) $c \leq a$ and $d \leq b$.

Given a set of n points $(a_1, b_1), \ldots, (a_n, b_n) \in \mathbb{R}^2$, consider the problem of computing the number of pairs of points that are comparable. You may assume for simplicity that all points are distinct. Design and analyze an algorithm (as fast as possible) for counting the number of comparable pairs of points.[9]

Exercise 2.2 Design a recursive algorithm implementing the specification. (No proof or analysis is needed.)[10]

[9] *Hint:* As a warmup, try to instead compute the number of *incomparable* pairs of points. (A pair of points is incomparable if is not comparable. E.g., (x_1, y_1) and (x_2, y_2) are incomparable if $x_1 < x_2$ and $y_1 > y_2$.) You may want to do exercise 1.7 before this one.

[10] Your algorithms will not be very fast, but their correctness should follow from an induction argument using the recursive spec as the induction hypothesis.

You are welcome to describe your algorithm using high level steps as long as they are clearly easy to implement and would take linear time; e.g., 'let G_1 be the graph obtained by removing such and such vertices and such and such edges." See the sample solutions on page 60.

1. `subset-sum`$(x_1, \ldots, x_n,\ T)$: Given n integers $x_1, \ldots, x_n \in \mathbb{Z}$, and an additional integer T, returns `true` if there is a subset of indices $S \subseteq [n]$ such that $\sum_{i \in S} x_i = T$, and `false` otherwise.

2. `min-vertex-cover`$(G = (V, E))$: Given an undirected graph $G = (V, E)$, return the minimum size of any vertex cover of G.

   (A vertex cover is a set of vertices $S \subseteq V$ that includes at least one endpoint of every edge $e \in E$.)

Exercise J.3. Prove by induction that a tree with n vertices has $n - 1$ edges.

Exercise 2.10. Each year, medical students apply to residency programs, and programs decide which applicants to accept. Both sides have preferences: students rank programs, and programs rank applicants. The National Resident Matching Program (NRMP) has used a stable matching algorithm since the 1950s to assign students to programs in a way that prevents "blocking pairs"—a student and program who would both prefer each other over their assigned match.

Formally, consider two disjoint sets A and B with $|A| = |B| = n$. Each element of A has a strict preference ranking over all elements of B, and vice versa. A *perfect matching* is a bijection $M : A \to B$. A matching M is *unstable* if there exist $a \in A$ and $b \in B$ such that a prefers b to $M(a)$ and b prefers a to $M^{-1}(b)$; such a pair (a, b) is called a *blocking pair*. A matching with no blocking pair is *stable*.

The *Gale-Shapley algorithm* builds a perfect matching as follows. At a high level, each $a \in A$ applies to one $b \in B$ at a time, and each $b \in B$ tentatively accepts one of its applicants. It terminates when the tentatively accepted applications form a perfect matching.

In the first round, each $a \in A$ applies to its highest-ranked $b \in B$, and each $b \in B$ tentatively accepts its highest-ranked applicant, rejecting the rest. In each subsequent round, each rejected $a \in A$ applies to the best $b \in B$ it has not yet applied to. Each $b \in B$ tentatively accepts its most preferred among the new applicants and its current tentative match (if any), rejecting the rest.

1. Prove an upper bound on the number of rounds of the Gale-Shapley algorithm before terminating with a perfect matching.

2. Prove that the matching produced is stable.

Some of these problems have well-known efficient algorithms. Efficiency is besides the point; the goal here is to rapidly get some practice thinking recursively.

Lastly, do not change the recursive spec.

3. Prove that each element of A is matched to the best element of B that they could be matched to in any stable matching.

4. Prove that each element of B is matched to the *worst* element of A they could be matched to in any stable matching.

The last two points imply that, if there is a unique stable matching, the algorithm finds it regardless of which side proposes.

Exercise $n + 1$. Let S be a set of n straight lines in the plane, partitioning the plane into a number of regions that is quadratic in n. For simplicity, we assume that the lines are in "general position": no three lines meet at a point, and that no two lines are parallel. A *two-coloring* colors each region one of two colors, such that every two "adjacent" regions that share an edge have different colors. Design and analyze an algorithm, as fast as possible, computing a 2-coloring of the regions of S.[11]

Exercise J.4. Let $A[1..n]$ be an array of elements. The *frequency* of an element is the number of times it appears in $A[1..n]$. An element is a *majority element* if it has frequency at least $n/2$.

If the elements are comparable then we can identify the majority element by sorting all the elements and scanning the sorted list. Here we do *not* assume that the elements are pairwise comparable; we can only test if two elements $A[i]$ and $A[j]$ are equal.

1. Write a recursive spec for the majority element problem.

2. Design a divide-and-conquer algorithm (implementing your recursive spec) for the majority element problem that divides the input in half, and runs in $O(n \log n)$.[12]

3. Prove your algorithm is correct by induction, where the recursive spec is your induction hypothesis.

4. Analyze the running time of your algorithm.

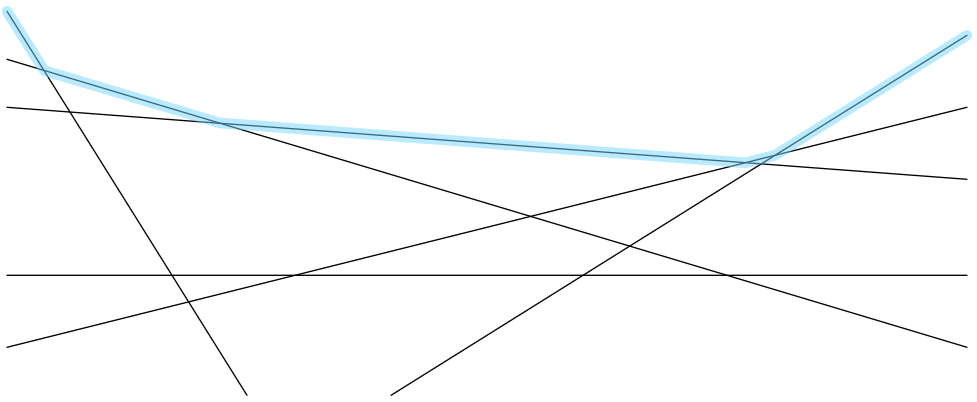
[11] *Hint:* Suppose you've two-colored all the regions induced by a subset of $n - 1$ of the lines. The nth line splits some of these regions in half, and each of these "split regions" violates the coloring requirements... now what?

[12] There is also a well-known $O(n)$ time algorithm which we are not asking for.

Exercise 3.6. Let L be a set of n lines of the plane, where the ith line is of the form $y_i(x) = a_i x + b_i$ for $a_i, b_i \in \mathbb{R}$. The *upper envelope* is the function

$$U_L(x) = \max_{i \in [n]} y_i(x).$$

Observe that U_L is piecewise-linear and convex. We represent the upper envelope as a list of *breakpoints* $(x_1, \ell_1), (x_2, \ell_2), \ldots, (x_m, \ell_m)$ where $x_1 < x_2 < \cdots < x_m$ are the x-coordinates where the topmost line changes, and ℓ_i is the line achieving the maximum for $x \in [x_i, x_{i+1})$.[13]



For example, given three lines $\ell_1 \colon y = -x + 4$, $\ell_2 \colon y = 2$, and $\ell_3 \colon y = x$, the upper envelope is: ℓ_1 for $x < 1$, then ℓ_2 for $x \in [1, 2]$, then ℓ_3 for $x > 2$.

Design and analyze a recursive algorithm, as fast as possible, computing the upper envelope of n lines in the plane.[14]

Remark J.1. The upper envelope appears in computational geometry (convex hulls, linear programming) and in the "convex hull trick" for optimizing certain dynamic programs.

Exercise J.5. Given a set P of n points in $\mathbb{R}^2$, the *convex hull* of P is the smallest convex polygon containing all points in P. Equivalently, it is the polygon formed by stretching a rubber band around the points and letting it snap tight.

We represent the convex hull as a list of its vertices in clockwise order, starting from the leftmost point (breaking ties by taking the lowest).

[13] You should also specify the line on top for $x < x_1$; handle this however you like.

[14] *Hint:* Of course n lines induces $\binom{n}{2}$ points of intersection, but how many breakpoints can the upper envelope have?

For example, given points $(0,0)$, $(1,2)$, $(2,1)$, $(1,0)$, and $(1,1)$, the convex hull is the quadrilateral with vertices $(0,0), (1,2), (2,1), (1,0)$ in clockwise order. (The point $(1,1)$ is in the interior.)

Design and analyze a divide-and-conquer algorithm, as fast as possible, computing the convex hull of n points in the plane.

For simplicity, you may assume the points are in general position: no three points are collinear, and no two points share the same x-coordinate.

Food for thought. Consider the bijection between lines, $y = ax + b$, and points $(a, -b)$. This is called geometric duality. Suppose you had a algorithm for convex hull. Leveraging duality, use the algorithm for the convex hull to design an algorithm for computing the upper envelope.

J.3 Week 2

Exercise 4.3. In Corollary 4.3, we saw that FFT can multiply two degree-n polynomials in $O(n \log n)$ time. Now consider the problem of multiplying n degree-1 polynomials. The input consists of n linear polynomials

$$a_1 + b_1 x, , \, a_2 + b_2 x, \, \ldots, \, a_n + b_n x,$$

and the output is the coefficients of their product, a degree n polynomial. Design and analyze an algorithm, as fast as possible, for this problem.

Exercise 4.6. For two polynomials $p(x) = a_0 + a_1 x + \cdots + a_n x^n$ and $q(x) = b_0 + b_1 x + \cdots + b_{n-1} x^n$, their product can be written as

$$p(x)q(x) = \sum_{i,j} a_i b_j x^{i+j}$$

where i, j both range from 0 through n. Suppose we instead wanted to compute the following "skew-product",

$$\sum_{i<j} a_i b_j x^{i+j},$$

where again i and j both range from 0 through n, but restricted to $i < j$. Design and analyze a fast algorithm for computing the skew-product of two polynomials.

Exercise 5.1. Below is a series of optimization problems that takes as input an array $A[1..n]$ of integers, and asks for optimal subsequences of A satisfying certain properties.[15] Design and analyze an algorithm for each of these problems, addressing Items 1–5 from Section 5.2.[16]

1. A sequence of numbers $x_1, \ldots, x_k$ is (strictly) increasing if $x_i < x_{i+1}$ for $i = 1, \ldots, k-1$. Compute the length of the longest increasing subsequence of A.

2. A sequence of numbers $x_1, \ldots, x_k$ is *convex* if $x_{i+1} - x_i \geq x_i - x_{i-1}$ for $i = 2, \ldots, k-1$. Compute the length of the longest convex subsequence of A.

3. Compute the maximum sum of any increasing subsequence of A.

[15] A "subsequence" of $x_1, \ldots, x_n$ is a sequence of the form $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

[16] In particular, no proof is required. We will interpret your recursive spec as the inductive hypothesis of a proof by induction, and fill in the the rest of the details ourselves. That said, some additional commentary may help the grader, especially in the event of partial credit.

4. Compute both:
   - the length of the longest increasing subsequences of A where the sum of integers is even,
   - the length of the longest increasing subsequence where the sum of integers is odd.

5. A sequence $x_1, \ldots, x_k$ is a *palindrome* if the reversed sequence is the same; i.e., $(x_1, \ldots, x_k) = (x_k, \ldots, x_1)$. For example,

   mom, *dad*, *racecar*, and *gohangasalamiimalasagnahog*

   are all palindromes. Compute the length of the longest palindrome subsequence of A.

6. Recall that the Fibonacci sequence

   1, 1, 2, 3, 5, 8, ...

   can be characterized as a sequence starting with 1, 1 and such that every successive number is the sum of the previous two numbers. ($1+1=2$, $1+2=3$, $2+3=5$, ...) Generalizing this notion, we say that a sequence $y_1, y_2, \ldots, y_k$ is *sub-Fibonacci* if $y_i \leq y_{i-2} + y_{i-1}$ for $i = 3, 4, \ldots, k$. That is, each successive number is at most the sum of the previous two.

   Let $A[1..n]$ be an array of numbers. Design and analyze an algorithm that computes the length of the longest increasing and sub-Fibonacci subsequence of A.

Exercise 5.2. Let $A[1..m]$ and $B[1..n]$ be two arrays. Each of the following problems asks for some aspect (e.g., the length) of a common subsequence of $A[1..n]$ and $B[1..n]$ satisfying or optimizing certain properties. (A common subsequence of A and B is a sequence that is a subsequence of both A and B.) Design and analyze an algorithm for each of these problems, addressing Items 1–5 from Section 5.2.[16]

1. Compute the length of the longest common subsequence of A and B.
2. Compute the length of the shortest common supersequence of A and B.
3. Compute the length of the longest common subsequence of A and B that is a palindrome.

Exercise 5.12. Word segmentation is the problem of dividing a string of written language into its component words. In English and many other languages using some form of the Latin alphabet, the space is a good approximation of a word divider (word delimiter), although this concept has limits because of the variability with which languages emically regard collocations and compounds. Many English compound nouns are variably written (for example, ice box = ice-box = icebox; pig sty = pig-sty = pigsty) with a corresponding variation in whether speakers think of them as noun phrases or single nouns; there are trends in how norms are set, such as that open compounds often tend eventually to solidify by widespread convention, but variation remains systemic. In contrast, German compound nouns show less orthographic variation, with solidification being a stronger norm. However, the equivalent to the word space character is not found in all written scripts, and without it word segmentation is a difficult problem.

Languages which do not have a trivial word segmentation process include Chinese, Japanese, where sentences but not words are delimited, Thai and Lao, where phrases and sentences but not words are delimited, and Vietnamese, where syllables but not words are delimited. In some writing systems however, such as the Ge'ez script used for Amharic and Tigrinya among other languages, words are explicitly delimited (at least historically) with a non-whitespace character.

The Unicode Consortium has published a Standard Annex on Text Segmentation,[1] exploring the issues of segmentation in multiscript texts.[17]

We model the word segmentation problem as follows. Let $A[1..n]$ be a sequence of letters from a fixed alphabet. A *word segmentation* is a partition of A into contiguous intervals $A[i_0 + 1..i_1], A[i_1 + 1..i_2], \ldots, A[i_\ell + 1..i_{\ell+1}]$; where $i_0 = 0$, $i_{\ell+1} = n$, and ℓ is any integer; such that each interval $A[i_j + 1..i_{j+1}]$ is a word. For example, the text

"youareawesome" has the word segmentation "you", "are", "awesome".

To determine which sequences form words, we assume access to a precomputed boolean array $\texttt{words}(1..n, 1..n)$, where for $i \leq j$, $\texttt{words}(i, j) = \texttt{true}$ iff $A[i..j]$ forms a word.

Given $A[1..n]$ and $\texttt{words}(1..n, 1..n)$, consider the problem of deciding if there is a word segmentation of A. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 5.11. In fair redistricting, the goal is to divide up a population into geographic regions so that all the local elections are as close as possible. This is like the opposite of gerrymandering; having close elections maximizes the influence of each constituent's vote.

[17] https://en.m.wikipedia.org/wiki/Text_segmentation#Word_segmentation

For simplicity we consider the following 1-dimensional formulation of the problem. Let $A[1..n]$ be a sequence of n (positive or negative) integers. Each index $i \in [n]$ represents a neighborhood block in a sequence of n blocks. We have two political parties, the Apples and the Oranges. For each index i, $A[i] \in \mathbb{Z}$ represents the number of people in block i that have registered for the Apple party, minus the number of people in block i that have registered for the Orange party; i.e.,

$$A[i] = \begin{pmatrix} \text{\# Apples} \\ \text{in block } i \end{pmatrix} - \begin{pmatrix} \text{\# Oranges} \\ \text{in block } i \end{pmatrix}.$$

A *district* is defined as a consecutive interval $[i..j]$ of blocks, where $1 \leq i \leq j \leq n$. The *disparity* of a district $[i..j]$ is the absolute difference between the number of Apples and number of Oranges in the district:

$$\text{(disparity of } [i..j]) = \left| \begin{pmatrix} \text{\# Apples} \\ \text{in blocks } i..j \end{pmatrix} - \begin{pmatrix} \text{\# Oranges} \\ \text{in blocks } i..j \end{pmatrix} \right| = \left| \sum_{k=i}^{j} A[k] \right|.$$

Given $A[1..n]$ and $L \in [n]$, the goal is compute the minimum total disparity,

$$\sum_{k=1}^{L} (\text{disparity of } A[i_k..j_k])$$

over all partitions of the sequence blocks $[1..n]$ into L (nonempty) districts $[i_1..j_1], \ldots, [i_L..j_L]$. ("Partition" means that the districts are disjoint and cover all of the n blocks.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

J.4 Week 3

Exercise 6.4. Let $G = (V, E)$ be a directed graph. Design and analyze an algorithm for each of the following problems.

1. Decide if there is a vertex x such that x can reach all other vertices.
2. Decide if there is a vertex x such that x can be reached by all other vertices.
3. Decide if there is a vertex x such that every vertex can either reach x or be reached from x.

Exercise 7.1 Each of the following problems describes a graph problem over an unweighted directed graph $G = (V, E)$ with m edges and n vertices. Each problem can be solved by constructing an auxiliary graph and calling a shortest path algorithm from this chapter as a black box. For each problem, (a) design an auxiliary graph and shortest path problem, (b) indicate which algorithm to apply and how, and (c) analyze the running time. No proof of correctness is required.

1. For two vertices s and t, compute the length of the shortest walk from s to t with an even number of edges.[18]
2. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is a multiple of 5.
3. For two vertices s and t, compute the length of the shortest walk from s to t where the number of edges is either a multiple of 2, or a multiple of 3.

Exercise J.6. Red light, green light! The following model is inspired by streetlights which, at a given point in time, allow traffic to enter some streets and block traffic from entering others.

Let $G = (V, E)$ be a directed graph with positive, integer edge lengths $\ell(e)$ for $e \in E$, which represents the amount of time it takes to traverse that edge. (Here time will always be an integer.) Suppose each edge is also annotated as being either "even" or "odd". The model is as follows.

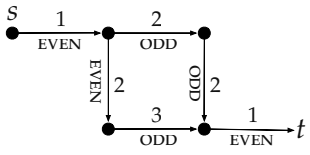
We start at a vertex s at time $x = 0$, and want to get to a vertex t as soon as possible. In general, taking an edge e takes $\ell(e)$ units of time, which is added to x. However, we can only take an even edge when the time x is even, and we can only

[18] For this and the other problems, if there is no such walk, then your algorithm should return $+\infty$.

take an odd edge when the time x is odd. We can also wait at a vertex for one unit of time during which x increases by 1.

Loosely speaking, one can imagine each vertex as being like a stop light. An even edge is like a one-way street that traffic can enter only when the time x is even. An odd edge is like a one-way street that traffic can enter only when the time x is odd.

In the example on the right, the (s,t)-path along the top takes $1+2+2+1_{(\text{WAIT})}+1=7$ units of time. (Here we annotated the 1's that correspond to waiting). The path along the bottom takes $1+1_{(\text{WAIT})}+2+1_{(\text{WAIT})}+3+1=9$ units of time.



For $s,t\in V$, consider the problem of computing the minimum amount of time needed to get from s to t in the time model described above. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 5.12. Word segmentation is the problem of dividing a string of written language into its component words. In English and many other languages using some form of the Latin alphabet, the space is a good approximation of a word divider (word delimiter), although this concept has limits because of the variability with which languages emically regard collocations and compounds. Many English compound nouns are variably written (for example, ice box = ice-box = icebox; pig sty = pig-sty = pigsty) with a corresponding variation in whether speakers think of them as noun phrases or single nouns; there are trends in how norms are set, such as that open compounds often tend eventually to solidify by widespread convention, but variation remains systemic. In contrast, German compound nouns show less orthographic variation, with solidification being a stronger norm. However, the equivalent to the word space character is not found in all written scripts, and without it word segmentation is a difficult problem.

Languages which do not have a trivial word segmentation process include Chinese, Japanese, where sentences but not words are delimited, Thai and Lao, where phrases and sentences but not words are delimited, and Vietnamese, where syllables but not words are delimited. In some writing systems however, such as the Ge'ez script used for Amharic and Tigrinya among other languages, words are explicitly delimited (at least historically) with a non-whitespace character.

The Unicode Consortium has published a Standard Annex on Text Segmentation,[1] exploring the issues of segmentation in multiscript texts.[19]

[19] https://en.m.wikipedia.org/wiki/Text_segmentation#Word_segmentation

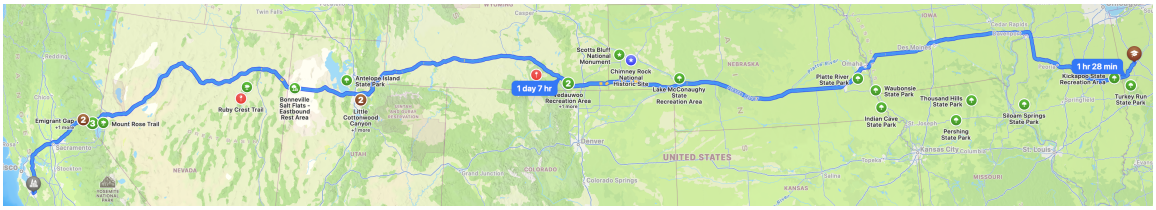
We model the word segmentation problem as follows. Let $A[1..n]$ be a sequence of letters from a fixed alphabet. A *word segmentation* is a partition of A into contiguous intervals $A[i_0 + 1..i_1], A[i_1 + 1..i_2], \ldots, A[i_\ell + 1..i_{\ell+1}]$; where $i_0 = 0$, $i_{\ell+1} = n$, and ℓ is any integer; such that each interval $A[i_j + 1..i_{j+1}]$ is a word. For example, the text

"youareawesome" has the word segmentation "you", "are", "awesome".

To determine which sequences form words, we assume access to a precomputed boolean array $\texttt{words}(1..n, 1..n)$, where for $i \leq j$, $\texttt{words}(i, j) = \texttt{true}$ iff $A[i..j]$ forms a word.

Given $A[1..n]$ and $\texttt{words}(1..n, 1..n)$, consider the problem of deciding if there is a word segmentation of A. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 5.10. Imagine you are planning a long road trip to San Jose, CA and you've already fixed your route (through Illinois, Iowa, Nebraska, Wyoming, Utah, Nevada, and then California via Tahoe).



You don't mind the drive, but you hate having to stop to get gas. What a pain! And some stations take longer than others. The car can drive at most 500 miles before needing to stop to get gas. The high-level goal is to plan your gas stops to minimize the amount of time spent pumping gas, while making sure you don't run out of gas between them.

We let n denote the total number of gas stations, and number the gas stations $1, \ldots, n$ in order along the route. The input consists of two arrays $D[1..n]$ and $T[1..n]$, and an additional value $Z > 0$. For each i, $D[i] \in \mathbb{R}_{\geq 0}$ is a nonnegative real number representing the distance along the route at which the gas station appears. (Again, the gas stations are ordered to be nondecreasing in $D[i]$.) $T[i]$ is a positive real number representing the amount of time it takes to pump gas at station i. Z represents the distance to the destination. You start with a full tank of gas.

Design and analyze an algorithm to compute the minimum amount of time one has to spend at gas stations in going from location 0 to location Z, while ensuring that you never drive more than 500 miles without visiting a gas station. (You should return $+\infty$ if it is impossible to do so.)

J.5 Week 4

Exercise 8.2. Recall that in foreign exchange, different currencies can be exchanged at various exchange rates. For example, at the time of writing, one can exchange 1 US dollar for about 0.89 Euro's. *Currency arbitrage* occurs when you start with some quantity of one currency, and make a sequence of exchanges through different currencies and end up with even more of the original currency than you started with.

Suppose we have n countries and for every (ordered) pair of countries X and Y we have an exchange rate $r_{X,Y}$; meaning, one unit of currency X can be exchanged for $r_{X,Y}$ units of currency Y. Consider the problem of identifying currency arbitrage starting from a fixed currency X, or deciding that no arbitrage is possible. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 9.8. Let $k \geq 3$. Recall that the k-SAT problem is the special case of SAT where $f(x_1, \ldots, x_n)$ is a CNF with exactly k distinct variables per clause.

1. Show that a polynomial time algorithm for k-SAT implies a polynomial time algorithm for $(k+1)$-SAT.

2. Show that a polynomial time algorithm for $(k+1)$-SAT implies a polynomial time algorithm for k-sat.

Exercise 9.11. In the *1-in-3-SAT* problem, one is given a 3-CNF $f(x_1, \ldots, x_n)$. The goal is to find a satisfying assignment such that every disjunctive clause (with three variables) is satisfied by *exactly* one of the variables.

Either (a) show that a polynomial time algorithm for 1-in-3-SAT implies a polynomial time algorithm for SAT, or (b) design and analyze a polynomial time algorithm for 1-in-3-SAT.

J.6 Week 5

Exercise 5.2. Let $A[1..m]$ and $B[1..n]$ be two arrays. Each of the following problems asks for some aspect (e.g., the length) of a common subsequence of $A[1..n]$ and $B[1..n]$ satisfying or optimizing certain properties. (A common subsequence of A and B is a sequence that is a subsequence of both A and B.) Design and analyze an algorithm for each of these problems, addressing Items 1–5 from Section 5.2. (No proof required.)

1. Suppose A and B consists of integers. Compute the length of the longest common increasing subsequence of A and B.

Exercise 5.12. Word segmentation is the problem of dividing a string of written language into its component words. In English and many other languages using some form of the Latin alphabet, the space is a good approximation of a word divider (word delimiter), although this concept has limits because of the variability with which languages emically regard collocations and compounds. Many English compound nouns are variably written (for example, ice box = ice-box = icebox; pig sty = pig-sty = pigsty) with a corresponding variation in whether speakers think of them as noun phrases or single nouns; there are trends in how norms are set, such as that open compounds often tend eventually to solidify by widespread convention, but variation remains systemic. In contrast, German compound nouns show less orthographic variation, with solidification being a stronger norm. However, the equivalent to the word space character is not found in all written scripts, and without it word segmentation is a difficult problem.

Languages which do not have a trivial word segmentation process include Chinese, Japanese, where sentences but not words are delimited, Thai and Lao, where phrases and sentences but not words are delimited, and Vietnamese, where syllables but not words are delimited. In some writing systems however, such as the Ge'ez script used for Amharic and Tigrinya among other languages, words are explicitly delimited (at least historically) with a non-whitespace character.

The Unicode Consortium has published a Standard Annex on Text Segmentation,[1] exploring the issues of segmentation in multiscript texts.[20]

We model the word segmentation problem as follows. Let $A[1..n]$ be a sequence of letters from a fixed alphabet. A *word segmentation* is a partition of A into contiguous intervals $A[i_0 + 1..i_1], A[i_1 + 1..i_2], \ldots, A[i_\ell + 1..i_{\ell+1}]$; where $i_0 = 0$, $i_{\ell+1} = n$, and ℓ is any integer; such that each interval $A[i_j + 1..i_{j+1}]$ is a word. For example, the text

"youareawesome" has the word segmentation "you", "are", "awesome".

[20] https://en.m.wikipedia.org/wiki/Text_segmentation#Word_segmentation

To determine which sequences form words, we assume access to a precomputed boolean array $\texttt{words}(1..n, 1..n)$, where for $i \leq j$, $\texttt{words}(i, j) = \texttt{true}$ iff $A[i..j]$ forms a word.

Given $A[1..n]$ and $\texttt{words}(1..n, 1..n)$, consider the problem of deciding if there is a word segmentation of A. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 3.2. Here we assume the same model as exercise J.4 – an array $A[1..n]$ of incomparable elements[21] – and generalize the notion of a majority element. For $k > 1$, an element is a $(1/k)$-*heavy-hitter* if it has frequency at least n/k. (e.g., majority elements are $(1/2)$-heavy hitters.) Note that there are at most k $(1/k)$-heavy-hitters in $A[1..n]$.

1. (2.5 pt.) Give a recursive spec for the $(1/k)$-heavy hitter problem.
2. (2.5 pt.) Give a divide-and-conquer algorithm for the $(1/k)$-heavy hitter problem implementing your recursive spec (the faster the better). (This algorithm should be similar to your majority element algorithm.)
3. (2.5 pt.) Analyze the running time of your algorithm.
4. (2.5 pt.) Prove your algorithm is correct by induction, where the recursive spec should act as your induction hypothesis.

Your solutions should generalize that of finding the majority elements. That is, plugging in $k = 2$ should recover the same results as for the majority problem.[22]

Exercise 7.6. Suppose you are a city planner, and you are worried that it takes too long for an ambulance to get from the university to the hospital. You've decided to build one new street to improve this particular route. You and your crack team have researched and gathered a list of potential streets that could be built; the task now is to choose one street that will improve this route the most.

To model this formally, let $G = (V, E)$ be a directed graph with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$, and let $s, t \in V$ be two vertices. Let E be partitioned into two nonempty disjoint sets E_0 and E_1. E_0 represents the edges/streets that are "already built"; E_1 represents the edges/streets that you can add to the graph. Let $G_0 = (V, E_0)$ denote the graph of already built edges.

Consider the problem of identifying the single edge $e \in E_1$ so that, when you add e to G_0, the distance from s to t in the augmented graph is as short as possible. For

[21] In addition to disallowing sorting, no hash tables are allowed as well.

[22] Your algorithm should be faster than $O(n^2)$ (which is the trivial running time) for, e.g., $k = n^{1/3}$.

this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

As a follow up: What if you can add two edges? For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

J.7 Week 6

Exercise 10.3. Recall that the exclusive-or (XOR) of two boolean variables x and y, denoted $x \otimes y$, is `true` iff exactly one of x or y is `true`.

In XOR-satisfiabilty, you are given a formula $f(x_1, \ldots, x_n)$ which is a conjuction of m (binary) XOR's from the n boolean variables and their negations; e.g.,

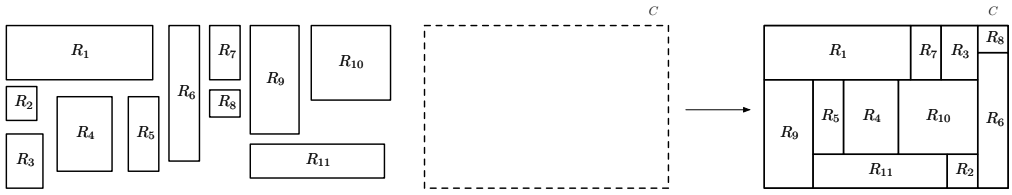
$$f(x_1, x_2, x_3, x_4) = (x_1 \otimes x_2) \wedge (x_2 \otimes \bar{x}_3) \wedge (x_3 \otimes x_4) \wedge (\bar{x}_1 \otimes \bar{x}_4);$$

and the goal is to compute a satisfying assignment for f. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 11.5. Let $x_1, \ldots, x_n \in \mathbb{N}$. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. The *partition problem* asks if one can partition $x_1, \ldots, x_n$ into two parts such that the sums of each part are equal.

Exercise 11.10. This problem is about rectangular *tiling*, where the input consists of a collection of (smaller) rectangular pieces, and a (larger) rectangular container. The high-level goal is to try to "tile" the rectangular container with the smaller rectangular pieces.



More formally, you have n rectangular pieces $R_1, ..., R_n$ where each R_i is specified by a height H_i and a width W_i. (You can assume the input is given as two arrays $H[1..n]$ and $W[1..n]$). You are also given a rectangular container C specified by a height H_C and a width W_C. A "tiling" of C by $R_1, \ldots, R_n$ is a placement of (some or all of) the rectangular pieces into the container so that there are no overlaps or gaps. (Like a jigsaw puzzle, or a tiled bathroom floor. To be clear, you can only use each piece once.) A "perfect tiling" of C by $R_1, \ldots, R_n$ is a tiling where all the rectangular pieces are used. For simplicity we assume that you cannot rotate the pieces.

1. (5 points) Consider the problem of computing a tiling of C by $R_1, \ldots, R_n$. (Not all the pieces have to be used.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. (5 points) Now consider the problem of computing a perfect tiling of C. (This time all the pieces have to be used.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

J.8 Week 7

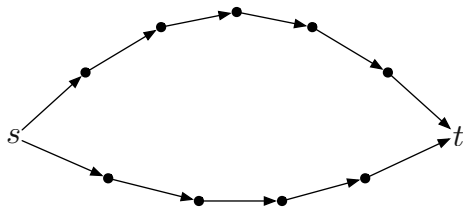
Exercise 12.6. Let $G = (V, E)$ be a directed graph with real-valued edge lengths $\ell : E \to \mathbb{R}$, and let $s, t \in V$. Consider the problem of computing the length of the shortest path from s to t. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 12.7. Here we consider an extension of shortest (s, t)-walks where one has to visit a family of vertices specified by the input. The input consists of a directed graph $G = (V, E)$ with positive edge lengths $\ell : E \to \mathbb{R}_{>0}$, as well as a list of vertices $x_1, \ldots, x_k \in V$. The high-level goal in both of the following problems is to compute the length of the shortest walk that visits all k vertices.

1. Suppose you are allowed to visit $x_1, \ldots, x_k$ in any order. Consider the problem of computing the length of the shortest walk visiting all of $x_1, \ldots, x_k$ in any order. (You may assume no vertices repeat in $x_1, \ldots, x_k$.) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

2. Suppose you have to visit $x_1, \ldots, x_k$ in the listed order. (Here vertices may repeat, but for simplicity you may assume that any two consecutive vertices v_i and v_{i+1} are distinct.) Consider the problem of computing the length of the shortest walk visiting $x_1, \ldots, x_k$ in order. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 12.19. Let $G = (V, E)$ be a directed graph and $s, t \in V$. We say that two (s, t)-paths $p, q : s \rightsquigarrow t$ are *internally disjoint* if they are vertex disjoint except at their endpoints s and t.



Consider the problem of computing the maximum number of edges of any two internally disjoint (s, t)-paths (for fixed s and t). For example, the picture above gives two internally disjoint (s, t)-paths with a total of 11 edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 13.2. Let k be fixed. Suppose you had access to a polynomial time algorithm for the decision version of k-coloring. Use this algorithm to obtain a polynomial time algorithm that obtains a k-coloring of a graph (if one exists).

Exercise 13.3. Recall that the k-coloring problem is to decide if there is a vertex coloring with at most k-colors. The subset k-coloring problem also specifies for each vertex v a subset $S_v \subseteq [k]$, and the k-coloring is restricted to have each vertex v colored by a color in S_v. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. The k-coloring problem in trees.
2. The k-coloring problem for intervals.
3. The subset k-coloring problem in trees, for fixed k.
4. The subset k-coloring problem for intervals, for fixed k.

J.9 Week 8

Exercise 16.4. Let $G = (V, E)$ be a undirected graph with distinct edge weights and let T be the MST. Suppose the weight of a single edge $e \in E$ is altered. The edge e may or may not be in T, and the weight may have been increased or decreased. Describe and analyze an algorithm that fixes the MST.[23]

Exercise 16.7. Of the multiple algorithms we have seen for MST, recall the parallel approach (Borůvka's algorithm) and the search approach (Prim's algorithm) with Fibonacci heaps. There is also a hybrid approach running in $O((m+n)\log\log n)$ time where one runs the parallel algorithm for some number of iterations (say, k iterations for a parameter k TBD) and contracts the selected edges, and then runs the search algorithm thereafter. (Such an algorithm is correct because ultimately it only takes good edges.) Describe and analyze this algorithm.[24]

Exercise 16.2. Consider the following algorithm that claims to compute the MST (in a connected graph).

`decremental-MST`

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in decreasing order of weight.
2. For $i = 1, 2, \ldots, n$:
   A. Unless $E - e_i$ is disconnected:
      1. $E \leftarrow E - e_i$.
3. Return E.

Either

1. Describe an input graph where `decremental-MST` fails to find the MST, or
2. Prove that `decremental-MST` always returns the minimum weight spanning tree.

For simplicity you may focus on the setting where the edge weights are distinct.

Justin's Irish problem. Suppose you are Irish. You are trying to host a large-scale party in your home in Dublin, Ireland. You have a whole host of activities (dunk-the-brit, drink-a-pint(s), lucky-charms-eating-competition, etc.). You had your attendees fill out a google form about which activities they prefer, and now you must schedule your events. However, your attendees are incredibly churlish, so you would rather

[23] Of course one could build an entirely new MST from scratch. Can one do better?
[24] It may be helpful to obtain a running time generically as a function of k, and then optimize k.

cancel the party if even a single attendee has two events they want to go to that are scheduled at the same time.

More formally, suppose there are n people $\{p_1, p_2, \ldots, p_n\}$, m time slots, k activities $A = \{a_1, \ldots, a_k\}$. Each person p_i is associated with a subset $L_i \subset A$ in which they would like to participate in. The problem is to decide if it is possible schedule all k activities in m time slots such that no person has 2 activities with a conflict, e.g., if person p_i wants to go to activity a_p and a_q, a valid solution should not schedule a_p and a_q in the same time-slot $r \in [m]$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

J.10 Week 9

Exercise 17.2. Let $G = (V, E)$ be a directed graph. For $s, t \in V$, let $\lambda(s, t)$ denote the size of the minimum (s, t)-cut. Prove the following "triangle inequality" for cuts:

$$\lambda(a, b) \geq \min\{\lambda(a, c), \lambda(c, b)\}$$

for any three vertices $a, b, c \in V$.

Exercise 18.3. Suppose you had an (s, t)-flow f. We know that there exists an (s, t)-path packing of the same size as f; here we are interested in algorithms that take f and compute such a path packing. Such a path packing is called a *flow decomposition* of f.

Design and analyze an algorithm that, in $O(m^2)$ time, computes a maximum path packing x of the same size as f, such that:

1. There are at most m distinct paths (with nonzero value) in x.
2. If f is integral, then x is also integral.

Exercise Slop-Maxxing. Suppose there is a $n \times n$ grid with the letters `S`, `L`, `O`, and `P`. Your goal is to form as many *disjoint* copies of the word `SLOP` as possible from the grid with the folowing rule: each letter of `SLOP` must be consecutive, i.e., you can start at any `S`, move to a neighboring `L`, move to its neighboring `O`, and so forth[25]. For example,

S	L	O	P	S
L	S	L	O	P
O	S	O	L	L
S	P	O	S	P
P	S	L	L	O

has solution

S	L	O	P	S
L	S	L	O	P
O	S	O	L	L
S	P	O	S	P
P	S	L	L	O

.

Give a polynomial time algorithm to acheive your goal of finding the most `SLOP` as possible.

Exercise 17.5. According to Wikipedia, a pinwheel is a simple child's toy made of a wheel of paper or plastic curls attached at its axle to a stick by a pin. It is designed to spin when blown upon by a person or by the wind.

A similar toy had developed independently in Polynesia (known as pekapeka or peʻapeʻa) using either coconut palm leaflets or strips of pandanus leaves;[26][27] in colder

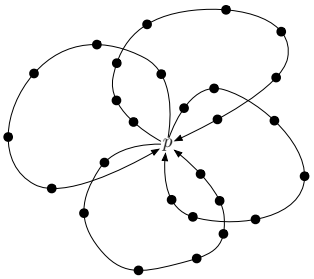
[25] Here, a letter Y is neighboring X if you can move up, down, left, or right once from X to get to Y, and vice versa.

[26] Koch, Gerd (1984). The Material Culture of Tuvalu. Suva: Institute of Pacific Studies, University of the South Pacific. p. 162. ISBN 9820202051.

[27] Te Rangi Hīroa (1930). Samoan Material Culture. Bernice P. Bishop Museum. p. 552.

climates like that of New Zealand (the toy also called pepepe in Māori), phormium leaves are used.[28]

Let $G = (V, E)$ be a directed graph. A *pinwheel* (in G) is defined as a collection of directed cycles all containing a common vertex p and vertex disjoint otherwise. p is called the *pin* of the pinwheel, and each cycle is called a *curl*. Consider the problem of computing the maximum number of curls in any pinwheel. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.



[28]Beattie, Herries (1994). Traditional Lifeways of the Southern Maori: The Otago University Museum Ethnological Project, 1920. University of Otago Press. p. 68. ISBN 978-0-908569-79-3.

J.11 Week 10

Exercise 9.12. A Boolean formula is in *disjunctive normal form (DNF)* if it is a disjunction of conjunctions. For example, the formula

$$f(x, y, z) = (x \wedge y) \vee (\bar{x} \wedge z),$$

which models "if x then y else z", is in disjunctive normal form. Note that by the distributive property,

$$(a \vee b) \wedge (c \vee d) = (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d),$$

so any CNF formula can be converted to a DNF formula. Let us define *DNF-SAT* as the problem of deciding whether a boolean formula in DNF is satisfiable.

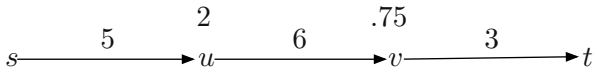
1. Is there a polynomial time algorithm for DNF SAT? If so, describe and analyze such an algorithm.

2. Would a polynomial time algorithm for DNF SAT imply a polynomial time algorithm for CNF SAT? Explain your reasoning.

Exercise 12.21. In various games like Mario Kart, not only are you trying to get from start to finish as fast as possible, but there are various items along the way that speed you up or slow you down. These items inspire the "multipliers" in the following variation of shortest paths.

Let $G = (V, E)$ be a DAG and $s, t \in V$. Let the edges have positive labels $\ell : E \to \mathbb{R}_{>0}$ representing the amount of time it takes to traverse the edge at unit speed. For each vertex $v \in V$, let $\alpha(v) > 0$ be a "speed multiplier". For example, if a vertex v has multiplier $\alpha(v) = 2$, then your speed doubles as you pass through v; if $\alpha(v) = 1/2$, then your speed decreases by a factor of 2 as you pass through v.

You start at s in a vehicle at unit speed. Your goal is to get to t as fast as possible. Along the way your speed increases or decreases multiplicatively based on the speed multipliers $\alpha(v)$ of each vertex you traverse. (For simplicity we assume $\alpha(s) = \alpha(t) = 1$.)



For example, in the path from s to t above, the edges are labeled by the time at unit speed, and the vertices are labeled by the speed multipliers. Going from s to t, the first edge is traversed at unit speed and takes 5 units of time, the second edge is

traversed at 2x speed and takes 3 units of time, and the last edge is traversed at 1.5x speed (since $2 \times .75 = 1.5$) and takes 2 units of time, for 10 units of time total.

Consider the problem of computing the minimum amount of time needed to get from s to t, taking into account the speed multipliers of each vertex. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

(Extra credit / food for thought: what if G was a general graph?)

Exercise 12.8. The following problems consider variations for the longest path problem that instead ask for longest walks (with varying definitions of "longest"). Here we recall that a walk may repeat vertices and edges, whereas a path cannot. Let $G = (V, E)$ be a directed graph with m edges and n vertices. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

1. Compute the length of the longest walk in G.
2. Compute the maximum number of distinct vertices of any walk in G.
3. Compute the maximum number of distinct edges of any walk in G.

Exercise 19.9. This problem models the following scenario. To prepare for the final, you and your classmate have decided to try to solve all n of the exercises in the textbook. Due to time constraints, you've decided to split up the problems. To try to make the total effort even, each of you went through and skimmed all the problems, and for each problem, estimated the number of minutes it would take to solve the problem. (Thus producing two lists of estimates, one for each person.) The goal is to assign each exercise to you or your classmate so as to minimize the maximum time that either of you would spend solving their assigned problems. We call this problem the "optimum study allocation problem".

Formally, we let $\{1, \ldots, n\}$ represent the n exercises. The input consists of two lists of natural numbers $A[1..n]$ and $B[1..n]$. For $i = 1, \ldots, n$, $A[i]$ represents your estimated time for problem i and $B[i]$ represents your classmate's estimated time for problem i. The goal is to divide $\{1, \ldots, n\}$ into two sets, $S \subseteq \{1, \ldots, n\}$ and $T = \{1, \ldots, n\} \setminus S$, to minimize the maximum of

$$\sum_{i \in S} A[i] \text{ and } \sum_{j \in T} B[j].$$

Here $\sum_{i \in S} A[i]$ represents the total time you estimate to solve all the problems in S and $\sum_{j \in T} B[j]$ represents the total estimated time for your partner to solve all the problems in T.

For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.7. Evacuate the building! Before Purdue can begin construction on its next glorious computer science building, at least two times the size of Lawson and with at most half as many restrooms, the architectural plan has to pass a series of safety regulations. One of the most important safety requirements is having an *evacuation plan* where in the event of an emergency, one can empty all the rooms to safe locations without overcrowding any of the hallways and other pathways, and risking a stampede. Here we model the problem of computing a safe evacuation plan (or declaring that none exists) as a graph problem, as follows.

The input consists of a directed graph $G = (V, E)$, two disjoint subsets of vertices $X, Y \subseteq V$, and integer edge labels $z : E \to \mathbb{N}$. X represents the rooms that would need to be evacuated, and Y represents the safe locations that the rooms in X must be evacuated to. (One safe location can be the destination of any number of evacuation routes.) The edges represent hallways/pathways through the building, and each edge e is annotated with an integer label $z(e)$ that signifies the number of evacuation paths it can accommodate. We define an *evacuation plan* as a set of paths from rooms in X to safety locations in Y, with one path starting from each room in X. We say that an evacuation plan is *safe* if no hallway/pathway is used more than its declared limit $z(e)$.

The problem is to either compute a safe evacuation plan or declare that no safe evacuation plan exists. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 19.8. Recall that in chess, a *rook* (♖) is a piece that can move horizontally or vertically as far as it wants. (It cannot move diagonally.) We say that a rook on a particular square is *attacking* another chess piece at another square if the rook can be moved horizontally or vertically from its square to the square of that chess piece.

Consider the following problem which we call the "n-rooks problem". At a high level, there is an $n \times n$ board where some of the squares have been marked off as "unavailable". The remaining squares are "available". The goal is to place n rooks in the available squares so that none of the rooks are attacking each other. We call this a "non-attacking placement" of the n rooks. (While rooks cannot be placed on unavailable squares, they can still move over unavailable squares to attack other pieces on the other side of unavailable squares.)

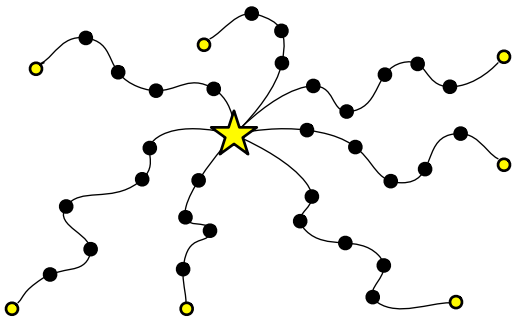
Below we draw a non-attacking placement of 4 rooks on a 4×4 chess board.

Unavailable squares are marked by X. Rooks are marked by ♖.

X	♖		X
♖	X	X	X
	X	♖	
X		X	♖

Formally, the input consists of an $n \times n$ bit-array $A[1..n, 1..n] \in \{\texttt{true}, \texttt{false}\}^{n \times n}$, where $A[i, j] = \texttt{true}$ indicates that square (i, j) is available, and $A[i, j] = \texttt{false}$ indicates that square (i, j) is unavailable. The goal is to decide if there is a non-attacking placement of n rooks in the available squares. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 17.4. A *star* is defined as a tree where at most one vertex has degree > 2. This special vertex is called the *center* of the star. (A star can have no particular center when it is just a path.)



Let $G = (V, E)$ be an undirected graph and $L \subseteq V$ a fixed set of vertices. Here we are interested in stars (as subgraphs of G, and centered anywhere) with as many leaves in L as possible. Consider the problem of computing the maximum number of leaves in L that can be obtained by any star in G. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

J.12 Week 11

Exercise 19.8. Recall that in chess, a *rook* (♖) is a piece that can move horizontally or vertically as far as it wants. (It cannot move diagonally.) We say that a rook on a particular square is *attacking* another chess piece at another square if the rook can be moved horizontally or vertically from its square to the square of that chess piece.

Consider the following problem which we call the "n-rooks problem". At a high level, there is an $n \times n$ board where some of the squares have been marked off as "unavailable". The remaining squares are "available". The goal is to place n rooks in the available squares so that none of the rooks are attacking each other. We call this a "non-attacking placement" of the n rooks. (While rooks cannot be placed on unavailable squares, they can still move over unavailable squares to attack other pieces on the other side of unavailable squares.)

Below we draw a non-attacking placement of 4 rooks on a 4×4 chess board. Unavailable squares are marked by `X`. Rooks are marked by ♖.

X	♖		X
♖	X	X	X
	X	♖	
X		X	♖

Formally, the input consists of an $n \times n$ bit-array $A[1..n, 1..n] \in \{\texttt{true}, \texttt{false}\}^{n \times n}$, where $A[i, j] = \texttt{true}$ indicates that square (i, j) is available, and $A[i, j] = \texttt{false}$ indicates that square (i, j) is unavailable. The goal is to decide if there is a non-attacking placement of n rooks in the available squares. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Life Imitates Art, TAs Grade Homework. Suppose you are a TA for `CS381`[29]. You and your fellow TAs (there are $n \in \mathbb{N}$ TAs total) are very stressed because there is a backlog of $k \in \mathbb{N}$ homeworks that need to be graded. To grade these k homeworks, Professor Kent has split the TAs up into k groups to handle each homework. Crucially, each group has one leader, who handles the most work (delegation, rubric construction, etc.).

Unfortunately, he used ChatGPT, so some TAs are in more than one group. He decided that the best we could do was to enforce a 'fairness requirement', where no single person leads too many groups.

[29] For us, this is very easy.

Formally, there are k homeworks and n TAs. Homework i for $i \in [k]$ is graded by TA group $S_i \subseteq [n]$ with some leader $j \in S_i$. The fairness requirement is that any TA $j \in [n]$ who is in groups $S_{j_1}, \ldots, S_{j_m}$, does not lead more than $\left\lceil \frac{1}{|S_{j_1}|} + \frac{1}{|S_{j_2}|} + \cdots + \frac{1}{|S_{j_m}|} \right\rceil$ of those groups. Given the k groups $S_1, S_2, \ldots, S_k \subseteq [n]$ as input, give a polynomial time algorithm (the faster the better) that *always* finds a leader assignment following the fairness requirement, or show that a poly-time algorithm for this problem implies a poly-time algorithm for SAT.

Exercise 22.1. Recall the array-backed lists from Section 22.3, where we showed how to support `append` in $O(1)$ amortized time via the "doubling trick". As discussed at the end of Section 22.3, one may also want to include a `remove` operation that removes the last item in the list. To implement this, one could just unallocate the corresponding spot in memory, and decrease the counter tracking the size of our list. However, after many such removals, we may end up with an array that is mostly empty and much larger then the number of items in the list. It would be desirable to maintain the array to be within a constant factor. Here we will develop an analog of the doubling trick to facilitate deletions.

1. A first approach to addressing deletions might be as follows.

   > *After removing the element from the array and decreasing the size counter, if the size of the list is now less than half of the capacity of the array, allocate a new array of half the capacity and copy the elements of the list over.*

   Unfortunately, this approach will not run in $O(1)$ amortized time. To prove this, devise a sequence of n `append` / `remove` operations (for n arbitrarily larger) on which the data structure would take $\Omega(n^2)$ time.

2. While the approach above didn't quite work, a slight modification of it will. Implement `remove` so that `append` and `remove` both take $O(1)$ amortized time. (Analyze these operations as well.) Your implementation should still follow the general format of allocating a new array and copying all the elements at certain points in time.

J.13 Week 12

Exercise 22.1. Recall the array-backed lists from Section 22.3, where we showed how to support `append` in $O(1)$ amortized time via the "doubling trick". As discussed at the end of Section 22.3, one may also want to include a `remove` operation that removes the last item in the list. To implement this, one could just unallocate the corresponding spot in memory, and decrease the counter tracking the size of our list. However, after many such removals, we may end up with an array that is mostly empty and much larger then the number of items in the list. It would be desirable to maintain the array to be within a constant factor. Here we will develop an analog of the doubling trick to facilitate deletions.

1. A first approach to addressing deletions might be as follows.

   > *After removing the element from the array and decreasing the size counter, if the size of the list is now less than half of the capacity of the array, allocate a new array of half the capacity and copy the elements of the list over.*

   Unfortunately, this approach will not run in $O(1)$ amortized time. To prove this, devise a sequence of n `append` / `remove` operations (for n arbitrarily larger) on which the data structure would take $\Omega(n^2)$ time.

2. While the approach above didn't quite work, a slight modification of it will. Implement `remove` so that `append` and `remove` both take $O(1)$ amortized time. (Analyze these operations as well.) Your implementation should still follow the general format of allocating a new array and copying all the elements at certain points in time.

Exercise J.7. Suppose you repeatedly flip a coin that is heads with fixed probability $p \in (0, 1)$.

1. What is the expected number of coin flips until you obtain one heads?[30] Prove your answer.

2. What is the expected number of coin flips until you obtain two heads? Prove your answer.

[30] If the first toss is heads, that counts as one coin flip. If the first toss is tails and the second toss is heads, that counts as two coin tosses. Etc. It may be helpful to first think about a fair coin, where $p = 1/2$.

3. For general $k \in \mathbb{N}$, what is the expected number of coin tosses until you obtain k heads? Prove your answer.

Exercise J.8. Last week you decided to clean up your toolshed, but you compulsively made the following silly mistake. Originally you had n matching pairs of nuts and bolts of different sizes, but you decided to organize the nuts and bolts separately into a box full of nuts and a box full of bolts, splitting up all the pairs of nuts and bolts in the process. So now you have n nuts, and n bolts, such that each nut fits a distinct bolt and vice-versa, but you don't know which nut goes with which bolt.



The bolts all look pretty similar, so you can't compare two bolts directly with each other and tell which one is bigger. Likewise you cannot compare the nuts to each other. However, you can try to fit one nut to one bolt, and see if either:

1. The nut fits the bolt.
2. The nut is too big for the bolt.
3. The nut is too small for the bolt.

We will treat one of these nut-to-bolt tests as a single operation. Your goal is to match up all nuts and bolts. Of course you can compare every pair of nut and bolt in $O(n^2)$ time, but can you do better?

Design and analyze an algorithm, as fast as possible, to recover all matching pairs of nuts and bolts.

Lazy grading. The midterm results are back, and the 381 GTAs need to grade a whopping n student submissions! They want to spend as little time as possible grading (they have better things to do!). They can choose between two methods for grading any given submission:

- **Real grading:** This always takes a couple of painstaking minutes per submission, but it guarantees an accurate grade. The solution type will also be cached into the GTAs brain.
- **Skimming:** This method takes negligible time. However, the grader will skim so quickly that they only have a 50% chance of giving an accurate grade! That is, of course, unless they've seen the solution type before (in which case it is guaranteed to be accurate).

Unfortunately, the students had many different solution types. However, a preliminary ChatGPT analysis showed that every solution type had at least k instances among the n submissions.

Give a randomized algorithm to minimize the total grading time such that the probability of grading a submission inaccurately is at most $1/n$ (the GTAs can handle a couple regrade requests, after all). What is the asymptotic upper bound on the total grading time in terms of the number of "real grading" submissions required by your algorithm? Express your bound in terms of n and k.

J.14 Week 13

Exercise 25.3. Let A and B be two events. Prove that the following three identities are all equivalent:

$$\mathbf{P}[A \mid B] = \mathbf{P}[A], \qquad \mathbf{P}[B \mid A] = \mathbf{P}[B], \qquad \mathbf{P}[A, B] = \mathbf{P}[A]\,\mathbf{P}[B].$$

(That is, if A and B satisfies any one of the identities above, then it automatically satisfies the other two.)

Exercise 25.5. Prove linearity of expectation (Theorem 25.6).

Exercise 25.8. Suppose you only have access to a coin that flips heads with a known probability p, and tails with (remaining) probability $1 - p$. Describe and analyze a protocol that uses a limited number of tosses of this biased coin in expectation (the smaller the better) to simulate 1 coin toss of a fair coin. (The expected number of biased coin tosses you make may depend on p.)

Exercise 27.9. Consider the streaming model where we have elements $e_1, e_2, \ldots$ presented one at a time by a stream. A natural task is to sample a fixed number of elements uniformly at random from the stream. Usually, sampling (say) 1 item from a set of m elements is easy: randomly generate a number k between 1 and m, and return the kth element form your set. Sampling in streaming is trickier because we cannot hold the entire stream in memory, and don't know the length of the stream.

1. Consider the following randomized streaming algorithm that selects one element s from the stream:

   `sample-one`

   `/*` *m counts the number of elements in the stream so far, and s is the "sample" of 1 element from the stream.* `*/`

   1. $m \leftarrow 0$, $s \leftarrow$ `nil`.
   2. For each element e presented by the stream:
      A. $m \leftarrow m + 1$.
      B. With probability $1/m$:
         1. $s \leftarrow e$.

   For $i \in \mathbb{N}$, let e_i denote the ith element in the stream. For $m \in \mathbb{N}$ let s_m denote the value of s after the mth iteration. Show that for all i and m,

$$\mathbf{P}[s_m = e_i] = \begin{cases} 0 & \text{if } m < i \\ 1/m & \text{if } m \geq i. \end{cases}$$

That is, for each m, s_m is a uniformly random element out of $\{e_1, \ldots, e_m\}$.[31]

2. Now let $k \in \mathbb{N}$ be a fixed parameter. (e.g., $k = 3$.) Suppose you want to sample a set of k elements from the stream without replacement. Design and analyze an algorithm generalizing `sample-one` that maintains a sample S of k elements drawn uniformly at random from the stream. That is, for $m \geq k$, your algorithm should have a set S of k elements, where any particular set of k elements is equally likely (i.e., with probability $1/\binom{m}{k}$). For $k = 1$, your algorithm should coincide with `sample-one` above.[32]

Exercise 27.6. Let $x_1, \ldots, x_n$ be n numbers drawn independently and uniformly at random from the range $[0, 1]$. Obviously they can be sorted in $O(n \log n)$ time with `merge-sort` or `quick-sort`. Design and analyze an algorithm sorting these random numbers in linear time in expectation.

Exercise 25.15. Let P be a set of n distinct points in the plane. Consider the problem of computing the minimum distance $\|p - q\|$ between any two distinct points $p, q \in P$. (If $|P| \leq 1$, the minimum distance is defined as $+\infty$.) For simplicity we assume the points are in general position, so that all pairwise distances are unique. You may have learned a deterministic divide-and-conquer algorithm that runs in $O(n \log n)$ time. Here we will analyze the running time of a different randomized algorithm. To help build intuition, first solve the following problem.



1. (3 points) Let $a_1, \ldots, a_n \in \mathbb{R}$ be n distinct numbers presented in a uniformly random order. Imagine tracking the minimum as you examine the numbers one by one in (the randomized) order. What is the expected number of times the minimum changes?

[31] Fix i. For $m < i$ the probability 0 since e_i hasn't event appeared in the stream. Now, what about $m = i$? What about $m = i + 1$?

[32] One way to frame your analysis is as follows. For $m \geq k$, let S_m denote the (randomized) sample S after m iterations. Prove the following statement by induction on $m - k$:

For all $m \geq k$, and all sets $X \subseteq \{e_1, \ldots, e_m\}$ of k elements,

$$\mathbf{P}[S_m = X] = \frac{1}{\binom{m}{k}}.$$

In our argument, you may have two cases depending on whether or not $e_m \in X$.

We assume black box access to a data structure for the following "threshold" version of the minimum distance problem. Fix a threshold $\rho > 0$. The data structure takes $O(1)$ time to initialize, and starts with an empty point set. It supports the following operation:

> `insert(`x`)`: Insert a point $x \in \mathbb{R}^2$ into the underlying point set. If the minimum distance in the underlying point set is (strictly) less than ρ, then return `true`; otherwise return `false`. This operation takes $O(1)$ expected time.[33]

This subroutine, combined with part 1, suggests the following algorithm. Here we assume that one can do math operations (add, subtract, square, etc.) in $O(1)$ time, and that we can sample a random permutation of n items in $O(n)$ time.

1. Let $p_1, p_2, \ldots, p_n$ permute P uniformly at random. *// $O(n)$ time.*
2. Let $r_1 = +\infty$ and $r_2 = \|p_1 - p_2\|$.

/ We maintain the invariant that r_i is the minimum distance over $p_1, \ldots, p_i$. */*

3. Start a new instance of the threshold data structure with threshold r_2. Insert p_1 and p_2.
4. For $i = 3, \ldots, n$:
   A. Insert p_i into the threshold data structure. If the data structure returns true:
      1. Let $r_i = \min\{\|p_i - p_j\| : j < i\}$.
      2. Replace the threshold data structure with a new one with radius r_i. Insert $p_1, \ldots, p_i$ into the data structure.
   B. Otherwise set $r_i = r_{i-1}$.
5. Return r_n.

Now analyze this algorithm via the following steps.

2. (3 points) For each index $i > 2$, analyze the exact probability that $r_i < r_{i-1}$.

3. (4 points) Finally, bound the expected running time of the algorithm.

[33] We can implement this data structure by building a "hash table over grid cells". Initially we have an empty dictionary for an empty point set. Given a point $x = (x_1, x_2)$, compute the key $\texttt{key}(x) = (\lfloor x_1/(\rho/\sqrt{2}) \rfloor, \lfloor x_2/(\rho/\sqrt{2}) \rfloor$ in $O(1)$ time. (Rounding is not a standard operation in all models of computation, but here we assume it takes $O(1)$ time.) If this key is already occupied by a point $y \in P$, then $\|x - y\| \leq \rho$, and we return `true`. Otherwise consider the 8 "neighboring" keys/cells where we add or subtract at most one or zero to each of the key's coordinates. Assuming the minimum distance was $\geq \rho$ before inserting x, each of these 8 cells can have at most 1 point. If the distance between x and any of the $O(1)$ points in neighboring cells is $< \rho$ from x, return `true`.

J.15 Week 14

Exercise 5.14. Who needs friends to play Set? The card game *Set* has 81 different cards varying in four features across three possibilities for each kind of feature. The four features are:

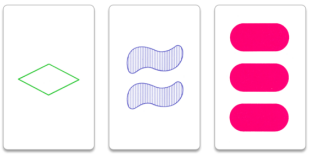
1. The shape, which is either a diamond, a squiggle, or an oval.
2. The number of shapes, which is either 1, 2, or 3.
3. The color of the shape, which is either red, green, purple.
4. The shading of the shape, which is either solid, striped, or open.

There are multiple variations of Set but they all are based on trying to form particular "sets" of three cards. A set consists of three cards satisfying *all* of these conditions:

1. They all have the same number or have three different numbers.
2. They all have the same shape or have three different shapes.
3. They all have the same shading or have three different shadings.
4. They all have the same color or have three different colors.

Loosely speaking, the rules of Set are summarized by: If you can sort a group of three cards into "two of _____ and one of _____", then it is not a set.

For example, the three cards on the right form a set. We have one card of each color, one of each shape, one of each shading, and one of each number. (In case this is printed in black and white, we mention that the first card is green, the second is purple, and the third is red.)



We will study a solitaire version of Set. Solitaire Set starts with 3 decks of cards. Each round you choose one of the following options.

1. If the three cards on top of the deck form a set, you may remove all three from the top of their decks, and set them aside.

2. You may discard one card from the top of one of the decks, revealing a new card underneath (unless that deck is empty).

(Even if the top three cards form a set, you can choose to discard one of them instead.) The goal is to form as many sets as possible by the time you empty all the decks.

We will consider the game as an optimization problem where we also know the full sequence of cards in the deck. The three decks are represented by three arrays $A[1..m]$, $B[1..n]$, and $C[1..p]$.[34] The cards are listed from top to bottom in each deck. *For simplicity, we assume access to a constant-time subroutine* `IsItASet(`i`,`j`,`k`)` *that*

[34]The cards in these decks are not distinct; a particular card may have multiple copies. So, for example, there may be many more than 81 cards total.

takes as input three indices i, j, k and outputs `True` *if the cards $(A[i], B[j], C[k])$ form a set, and* `False` *if they do not.*

The problem is to compute the maximum number of sets that can be obtained in a game of Solitaire Set over the decks $A[1..m]$, $B[1..n]$, and $C[1..p]$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 25.17. Let $G = (V, E)$ be an undirected graph with m edges and n vertices, vertex weights $w : V \to \mathbb{R}_{>0}$, and maximum (unweighted) vertex degree Δ. We let $W = \sum_{v \in V} w(v)$ denote the total weight of all the vertices.

Recall than an *independent set* is a set of vertices $S \subseteq V$ such that no two vertices in S are connected by an edge. Computing the maximum cardinality independent set is NP-Hard. In fact, for all $\epsilon > 0$, it is NP-Hard to get a $1/n^{1-\epsilon}$-approximation to the maximum cardinality independent set. (Of course, computing the maximum *weight* independent set is no easier.)

Consider the following algorithm that always returns a *maximal* independent set:

`randomized-greedy(`$G = (V, E)$`,`w`)`

1. Order the vertices $v_1, \ldots, v_n$ uniformly at random.
2. $S \leftarrow \emptyset$.
3. For $i = 1, \ldots, n$:
   A. If $S + v_i$ is independent, then set $S \leftarrow S + v_i$.
4. Return S.

We will analyze this algorithm by proving two claims. For both proofs, explain the reasoning for each step clearly.

1. (5 points) Consider a vertex $v \in V$ with degree d_v. Prove that $v \in S$ with probability $1/(d_v + 1)$.
2. (5 points) Prove that the expected weight of the returned independent set S is at least $W/(\Delta + 1)$. (Here you can assume part 1 as a fact even if you can't prove it.)

Set cover. In *set cover*, we are given m points $[m] = \{1, \ldots, m\}$, and n sets $S_1, \ldots, S_n \subseteq [m]$. The goal is to

find the minimum number of sets $S_{i_1}, \ldots, S_{i_k}$ such that $S_{i_1} \cup \cdots \cup S_{i_k} = [m]$.

For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Exercise 12.16. Let $G = (V, E)$ be a directed graph with no parallel edges. We say that a walk w in G has *distinct edges* if no edge appears twice in w. Consider the problem of computing the length of the longest walk in G with distinct edges. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.