CS381, Spring 2022

Lectures: 10:30 to 11:45 AM, on Tuesdays and Thursdays, in room 1066 of Honors College and Residences North.

Recitations: 6:00 to 6:50 PM, on either Tuesday or Thursday in WALC (see your registration for your exact schedule).1

Staff:

<table>
<thead>
<tr>
<th>Name</th>
<th>Email (at purdue.edu)</th>
<th>Office hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kent Quanrud</td>
<td>krq</td>
<td>Tue. 1–2PM at LWSN 1211</td>
</tr>
<tr>
<td>Md. Hassan Ameri</td>
<td>mameriek</td>
<td>Mon. 10–11AM at HAAS G50</td>
</tr>
<tr>
<td>Xiaoni Duan</td>
<td>duan79</td>
<td>Thur. 4:30–5:30PM at HAAS 175</td>
</tr>
<tr>
<td>Nitish Kumar Kumar</td>
<td>kumar410</td>
<td>Fri. 12:30–1:30PM at HAAS G50</td>
</tr>
<tr>
<td>Mohit Mayank</td>
<td>mayank</td>
<td>Wed. 2–3PM at HAAS G50</td>
</tr>
<tr>
<td>Yu Shi</td>
<td>shi1442</td>
<td>Thur. 1–2PM at LSWN 1190</td>
</tr>
<tr>
<td>Aniket Kumar Gupta</td>
<td>gupta532</td>
<td></td>
</tr>
<tr>
<td>Andrew Hou</td>
<td>hou116</td>
<td></td>
</tr>
<tr>
<td>Richard Li</td>
<td>li3426</td>
<td></td>
</tr>
<tr>
<td>Isha Prabesh</td>
<td>iprabesh</td>
<td></td>
</tr>
<tr>
<td>Pranjal Punekar</td>
<td>ppunekar</td>
<td></td>
</tr>
</tbody>
</table>

Please see the course syllabus (page 550) for additional information.

---

1Mohit will hold an additional recitation on Zoom at the address below, at 6:00PM on Thursdays. This is primarily for those who cannot make it in person.  
https://purdue-edu.zoom.us/j/96450499801?pwd=WNQsZ1h3Y3h3QVh3clh3ZTV3Y3h3ZTV3

2Recitation problems can be found in appendix F.
Class schedule

1. **January 11.** Searching and sorting, including binary search, merge sort, and lower bounds for sorting (chapter 1).

2. **January 13.** Boolean satisfiability, CNF satisfiability, and reduction from Boolean to CNF SAT (chapter 2). Homework 1 assigned (due January 24).

3. **January 18.** Subset sum (chapter 3).


5. **January 25.** Independent set, coloring, and other optimization problems on graphs and special cases of graphs (chapter 5).

6. **January 27.** Depth-first search and strongly connected components (chapter 6). Homework 3 assigned (due February 7).

7. **February 1.** Hamiltonian path on graphs and on DAG’s (chapter 7).

8. **February 3.** Breadth-first search and single-source shortest paths (chapter 8). Homework 4 assigned (due February 14).

9. **February 8.** Negative edge weights and all-pairs shortest paths (chapter 9).

10. **February 10.** Circuit satisfiability, NP-Completeness, and big-picture summary (chapter 10). (No homework assigned.)

(★) **February 16, 8–10PM:** Midterm 1. (See appendix H for practice problems.)


13. **March 1.** Spanning trees and Steiner trees (chapter 13).


---

3 All future dates should be regarded as tentative. The schedule is typically updated after the lecture to reflect what was covered.

4 In general, homework is assigned at the end of the week and due on the second Monday after that. (Some adjustments are made around midterms.)

5 On Zoom: https://purdue-edu.zoom.us/j/99736647778?pwd=dTBOMHp6WphUHdVa2ZMV2l6bEhxUT09.

   Homework 7 assigned. **Friday is the drop deadline.**

   (★) Spring Break


18. *March 24.* Matchings (chapter 18). (No homework assigned.)

   (★) *March 30, 8–10PM:* Midterm 2. (See appendix I for practice problems.)

19. *April 5.* Lazy data structures and amortized analysis (chapter 19).

20. *April 7.* The push-relabel algorithm for flow (as another example of amortized analysis, chapter 20). Homework 8 assigned.

21. *April 12.* Randomized searching and sorting, including quick-sort, quick-select, and randomized search trees (chapter 21).

22. *April 14.* Hashing and heavy hitters (chapter 22). Homework 9 assigned.

23. *April 19.* TBD (Some topic related to amortized analysis.).

24. *April 21.* TBD (Some topic related to randomized algorithms.) (No homework assigned).

25. *April 26.* TBD (Either review or “fun” related topic).

26. *April 28.* TBD (Either review or “fun” related topic). Last day of class. (No homework assigned.)

   (★) *Tuesday, May 3, 8–10AM, in LILY 1105:* Final. (See appendix J for practice problems.)
# Contents

Basic information .................................................. 1

Class schedule ...................................................... 2

I Lecture notes ...................................................... 12

1 Searching and sorting ............................................ 13
  1.1 Binary search ............................................... 13
  1.2 Sorting ...................................................... 15
  1.3 Lower bounds for sorting .................................. 23
  1.4 Further sorting algorithms ............................... 25
  1.5 Additional notes and references ......................... 28
  1.6 Exercises .................................................. 28

2 SAT: A different kind of search problem ..................... 30
  2.1 Boolean formulas ........................................... 30
  2.2 CNF ......................................................... 33
  2.3 Circuits ..................................................... 35
  2.4 Searching for answers ..................................... 39
  2.5 Additional results .......................................... 40
  2.6 Additional notes and references ......................... 42
  2.7 Exercises .................................................. 43
  2.A Proof of theorem 2.1 ...................................... 45

3 Subset-Sum, and the secret language of numbers ............ 48
  3.1 Brute force algorithms .................................... 48
  3.2 Packing in the complexity .................................. 54
  3.3 Perfect is the enemy of good ............................. 57
  3.4 Additional notes and references ......................... 58
  3.5 Exercises .................................................. 58
# Contents

4 Sequential Optimization 61
  4.1 TeX 61
  4.2 Edit distance 63
  4.3 Sequential matrix multiplication 69
  4.4 Additional notes and references 72
  4.5 Exercises 72

5 Optimization in graphs 77
  5.1 Independent set 77
  5.2 Graph coloring 87
  5.3 Dominating set 90
  5.4 Exercises 93

6 Searching and sorting graphs 99
  6.1 Directed acyclic graphs and topological sort 101
  6.2 Depth-first search 102
  6.3 Post-DFS Order 106
  6.4 Strongly-connected components 107
  6.5 Additional notes and materials 112
  6.6 Exercises 112

7 The longest path 115
  7.1 The longest path in a DAG 115
  7.2 The longest path in general graphs 117
  7.3 Exercises 120

8 Shortest paths 125
  8.1 Distances 125
  8.2 Distances in a DAG 126
  8.3 BFS for unweighted shortest paths 127
  8.4 Dijkstra’s algorithm for weighted shortest paths 129
  8.5 Exercises 133

9 Negative edge lengths and all-pairs shortest paths 136
  9.1 Negative edges, shortest walks, and shortest paths 137
  9.2 Bellman-Ford: distances with negative edges 138
  9.3 All pairs shortest paths and the doubling trick 140
  9.4 Floyd-Warshall 141
  9.5 The traveling salesman problem 143
  9.6 Additional notes and references 143
  9.7 Exercises 144
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.5</td>
<td>Exercises</td>
<td>309</td>
</tr>
<tr>
<td>20.A</td>
<td>Accelerating push-relabel with data structures</td>
<td>311</td>
</tr>
<tr>
<td>20.B</td>
<td>Directed minimum cut via one push-relabel</td>
<td>313</td>
</tr>
<tr>
<td>21</td>
<td>Random Sampling, Searching, and Sorting</td>
<td>316</td>
</tr>
<tr>
<td>21.1</td>
<td>Basic probability</td>
<td>316</td>
</tr>
<tr>
<td>21.2</td>
<td>Randomized minimum cut</td>
<td>319</td>
</tr>
<tr>
<td>21.3</td>
<td>Random variables</td>
<td>323</td>
</tr>
<tr>
<td>21.4</td>
<td>Averages</td>
<td>324</td>
</tr>
<tr>
<td>21.5</td>
<td>Randomized sorting</td>
<td>325</td>
</tr>
<tr>
<td>21.6</td>
<td>Randomized approximations for SAT</td>
<td>327</td>
</tr>
<tr>
<td>21.7</td>
<td>Exercises</td>
<td>328</td>
</tr>
<tr>
<td>22</td>
<td>Hashing and Heavy Hitters</td>
<td>333</td>
</tr>
<tr>
<td>22.1</td>
<td>Trends</td>
<td>333</td>
</tr>
<tr>
<td>22.2</td>
<td>Streaming</td>
<td>334</td>
</tr>
<tr>
<td>22.3</td>
<td>New Probabilistic Tools</td>
<td>336</td>
</tr>
<tr>
<td>22.4</td>
<td>Using hashing to approximate frequencies</td>
<td>338</td>
</tr>
<tr>
<td>22.5</td>
<td>Amplification</td>
<td>340</td>
</tr>
<tr>
<td>22.6</td>
<td>Extensions</td>
<td>341</td>
</tr>
<tr>
<td>22.7</td>
<td>Takeaways</td>
<td>342</td>
</tr>
<tr>
<td>22.8</td>
<td>Exercises</td>
<td>343</td>
</tr>
<tr>
<td>23</td>
<td>Hash tables and linear probing</td>
<td>345</td>
</tr>
<tr>
<td>23.1</td>
<td>Dictionaries</td>
<td>345</td>
</tr>
<tr>
<td>23.2</td>
<td>Hash tables with chaining</td>
<td>348</td>
</tr>
<tr>
<td>23.3</td>
<td>4-wise independence</td>
<td>350</td>
</tr>
<tr>
<td>23.4</td>
<td>Linear probing</td>
<td>354</td>
</tr>
<tr>
<td>23.5</td>
<td>Takeaways</td>
<td>357</td>
</tr>
<tr>
<td>23.6</td>
<td>Exercises</td>
<td>358</td>
</tr>
<tr>
<td>24</td>
<td>Self-adjusting trees</td>
<td>360</td>
</tr>
<tr>
<td>24.1</td>
<td>Rotations</td>
<td>360</td>
</tr>
<tr>
<td>24.2</td>
<td>Splay</td>
<td>362</td>
</tr>
<tr>
<td>25</td>
<td>Range Queries and Dynamic Trees</td>
<td>367</td>
</tr>
<tr>
<td>25.1</td>
<td>Range queries</td>
<td>367</td>
</tr>
<tr>
<td>25.2</td>
<td>Range queries on paths in a tree</td>
<td>368</td>
</tr>
<tr>
<td>25.3</td>
<td>Dynamic trees</td>
<td>369</td>
</tr>
<tr>
<td>25.4</td>
<td>Roots, re-rooting, and an application to dynamic disjoint union</td>
<td>375</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>26 Dynamic Connectivity and MST’s</td>
<td>377</td>
<td></td>
</tr>
<tr>
<td>26.1 Euler trees</td>
<td>377</td>
<td></td>
</tr>
<tr>
<td>26.2 Dynamic Connectivity</td>
<td>380</td>
<td></td>
</tr>
<tr>
<td>26.3 Dynamic MST</td>
<td>383</td>
<td></td>
</tr>
<tr>
<td>26.4 Exercises</td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>26.5 Additional notes and references</td>
<td>385</td>
<td></td>
</tr>
<tr>
<td>27 Convex minimization</td>
<td>386</td>
<td></td>
</tr>
<tr>
<td>27.1 Preliminaries</td>
<td>386</td>
<td></td>
</tr>
<tr>
<td>27.2 Convex functions</td>
<td>387</td>
<td></td>
</tr>
<tr>
<td>27.3 Gradient descent for smooth functions</td>
<td>390</td>
<td></td>
</tr>
<tr>
<td>27.4 Exercises</td>
<td>394</td>
<td></td>
</tr>
<tr>
<td>28 Greedy Approximations and LP Duality</td>
<td>396</td>
<td></td>
</tr>
<tr>
<td>28.1 Maximum coverage</td>
<td>396</td>
<td></td>
</tr>
<tr>
<td>28.2 Linear programs and LP duality</td>
<td>398</td>
<td></td>
</tr>
<tr>
<td>28.3 An algorithmic proof of LP duality</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>29 Randomized rounding</td>
<td>406</td>
<td></td>
</tr>
<tr>
<td>29.1 Max-flow Min-cut via LP duality</td>
<td>406</td>
<td></td>
</tr>
<tr>
<td>29.2 Set cover</td>
<td>409</td>
<td></td>
</tr>
<tr>
<td>29.3 Packing Multicommodity Paths</td>
<td>411</td>
<td></td>
</tr>
<tr>
<td>29.4 Exercises</td>
<td>414</td>
<td></td>
</tr>
<tr>
<td>30 Random Sums and Graphs</td>
<td>416</td>
<td></td>
</tr>
<tr>
<td>30.1 Random sums</td>
<td>416</td>
<td></td>
</tr>
<tr>
<td>30.2 Random graphs</td>
<td>418</td>
<td></td>
</tr>
<tr>
<td>30.3 A gap in component size</td>
<td>422</td>
<td></td>
</tr>
<tr>
<td>30.4 Galton-Watson branching processes</td>
<td>425</td>
<td></td>
</tr>
<tr>
<td>30.5 Exercises</td>
<td>427</td>
<td></td>
</tr>
<tr>
<td>A Additional exercises</td>
<td>428</td>
<td></td>
</tr>
<tr>
<td>B Basic data structures</td>
<td>429</td>
<td></td>
</tr>
<tr>
<td>B.1 Stacks</td>
<td>429</td>
<td></td>
</tr>
<tr>
<td>B.2 Heaps</td>
<td>430</td>
<td></td>
</tr>
<tr>
<td>C Mathematical Background</td>
<td>431</td>
<td></td>
</tr>
<tr>
<td>C.1 Induction</td>
<td>431</td>
<td></td>
</tr>
<tr>
<td>C.2 Estimating sums (quickly)</td>
<td>434</td>
<td></td>
</tr>
<tr>
<td>C.3 Additional notes and references</td>
<td>437</td>
<td></td>
</tr>
</tbody>
</table>
## Contents

**D Some NP-Complete Problems**  438  

**Bibliography**  441  

### II Course Materials  452  

**E Homework Assignments**  453  

- E.1 Homework 1  454  
- E.2 Homework 2  455  
- E.3 Homework 3  457  
- E.4 Homework 4  459  
- E.5 Homework 5  461  
- E.6 Homework 6  464  
- E.7 Homework 7  466  
- E.8 Homework 8  468  
- E.9 Homework 9  471  

**F Recitation Problems**  473  

- F.1 Recitation 1  474  
- F.2 Recitation 2  476  
- F.3 Recitation 3  477  
- F.4 Recitation 4  478  
- F.5 Recitation 5  479  
- F.6 Recitation 6  481  
- F.7 Recitation 7  482  
- F.8 Recitation 8  484  
- F.9 Recitation 9  485  
- F.10 Recitation 10  487  
- F.11 Recitation 11  489  
- F.12 Recitation 12  492  
- F.13 Recitation 13  493  

**G Selected solutions**  494  

**H Practice problems for midterm 1**  533  

- H.1 Mapping to other textbooks  533  
- H.2 Problems from Jeff's notes  535  

**I Practice problems for midterm 2**  537  

- I.1 Mapping to other textbooks  538  
- I.2 Problems from [Eri19]  539
Contents

J Practice problems for the final ........................................ 542
J.1 Mapping to other textbooks ..................................... 543
J.2 Problems from [Eri19] ............................................. 544

K Aggregate statistics ................................................... 547
K.1 Homework ........................................................ 547
K.2 Midterm 1 .......................................................... 548
K.3 Total ............................................................... 548
K.4 Additional remarks .............................................. 549

L Syllabus, policies, and procedures ............................... 550
L.1 Attendance ....................................................... 551
L.2 Textbooks ......................................................... 551
L.3 Correspondence .................................................. 552
L.4 Grading ........................................................... 553
L.5 Exams .............................................................. 553
L.6 Homework ....................................................... 554
L.7 Tips for success .................................................. 556
L.8 Registration ....................................................... 557
L.9 On the COVID-19 pandemic .................................. 557
L.10 Online exams ..................................................... 559
L.11 Academic integrity ............................................. 560
L.12 Posting Class Material ........................................ 561
L.13 Purdue’s Honor Pledge ....................................... 561
L.14 Grief Absence Policy ......................................... 561
L.15 Conduct and Courtesy ........................................ 561
L.16 Students with Disabilities ..................................... 562
L.17 Emergencies ...................................................... 562
L.18 Violent Behavior Policy ....................................... 563
L.19 Mental Health and Wellness ................................ 563
L.20 Health in general ............................................... 564
L.21 Nondiscrimination .............................................. 564
L.22 Privacy ........................................................... 565
L.23 Changes to the syllabus ....................................... 565
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 – at most, 10. To which Bill replies:

“You mean, $10^2$?!”

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

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:

\[
\begin{array}{c}
\text{||||||||||||} \\
= 23
\end{array}
\]

Clearly the right-hand side (RHS) is a much more efficient representation than the left-hand side (LHS). 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

\[\text{\textsuperscript{1}}\text{In fact, a previous version of this note mistakenly had 25 lines on the left, and nobody noticed.}\]
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.

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 one can use at most 100 guesses by looping through the numbers from 1 to 100; eventually you will guess the number. 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? If you keep 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

---

\(^2\)Why 6 instead of 7? Certainly 7 rounds are enough. But also we can see that 6 rounds are enough to reduce to (strictly) less than 2 choices (because \(\log_2(50) = \log_2(100) - 1 < 6\) - but the only integer less than 2 is 1! So 6 actually suffices. We should mention that normally we don’t care the difference of one more step when talking about computer algorithms – what’s one more step for a computer?
1. Searching and sorting

1.2 Sorting

number. This is our first algorithm; it is called **binary search**. Note that this algorithm isn’t “guessing” the secret number; it’s deducing it.

We have now introduced two mathematical functions that we have all implicitly understood 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

\[
\log_2(10^{80}) = 80 \log_2(10) \approx 265.75.
\]

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 an 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).

#### 1.2.1 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 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, 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 would be as follows.

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

So that’s how we naturally tend to sort. Let’s call this algorithm “human-sort”. 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 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 \( i \)th iteration (out of \( n \)), it identifies the \( i \)th largest number in increasing order. To identify 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

\[
(n \text{ iterations}) \times (\text{scanning } n \text{ items}) \approx n^2 \text{ operations over all.}
\]

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

---

3This algorithm is more commonly called insertion-sort selection-sort.
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 \( n \).

1.2.2 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 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 if 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

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

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 15), 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.

1.2.3 Merge-sort

The reader might have realized the following procedure to combine the two sorted lists into a single sorted list. Let us 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 in stark contrast to human-sort, where we had to scan all of the items to find the first item. Next, we need the second smallest number. 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 we have taken from the top of each list. Then in each 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. We needed a constant amount of time to produce each number in the output, so the overall running time on \( n \) numbers is \( O(n) \).

The above process is commonly called merging, and gives rise to a sorting algorithm called merge-sort. At a high-level, merge-sort is very simple: 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. 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 example, for an array of \( n \) comparable elements \( A[1..n] \), we define
\[
\text{merge-sort}(A[1..n]) = \text{an array consisting of the elements of } A \text{ sorted in increasing order.}
\]
Here the input and the output (and properties thereof) are made explicitly clear.

Now, a specification such as the above of course clarifies our intent when implementing. But also, by define such a commitment, it solidifies what we can assume from merge-sort when making recursive calls to smaller inputs. Indeed, the recursive specification will map directly to the inductive hypothesis when proving correctness.

Pseudocode for merge-sort is given in fig. 1.1. The task is now to analyze the algorithm. A full analysis of merge-sort should address two aspects.

1. Correctness: prove that merge-sort indeed 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 automatically 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. Note that \( \lfloor n/2 \rfloor \) and \( \lceil n/2 \rceil \) are both strictly less than \( n \) (because \( n \geq 2 \)), so by induction, both of these recursive calls will correctly sort half 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.

Having proven the algorithm is correct, it remains to analyze the running time. The first step is to model it, 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.

\[
T(0) = T(1) \leq C \\
T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + Cn.
\]

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 \( n \leq 1 \) then return \( A \).

/* In the general case, we divide the input and half and recursively sort each half. */

2. Let \( m = \lceil \frac{n}{2} \rceil \).

3. \( B_1[1..m] \leftarrow \text{merge-sort}(A[1..m]) \)

4. \( B_2[1..n-m] \leftarrow \text{merge-sort}(A[m+1..n]) \)

/* Now we merge the two sorted halves in linear time. */

5. Allocate a new array \( C[1..n] \), let \( i = 1 \), and let \( j = 1 \).

6. While \( i \leq m \) and \( j \leq n-m \)
   
   A. if \( B_1[i] \leq B_2[j] \)
      1. \( C[i+j-1] \leftarrow B_1[i] \) and \( i \leftarrow i + 1 \)
   
   B. else
      1. \( C[i+j-1] \leftarrow B_2[j] \) and \( j \leftarrow j + 1 \)

7. while \( i \leq m \)
   
   A. \( C[i+j-1] \leftarrow B_1[i] \) and \( i \leftarrow i + 1 \)

8. while \( j \leq m \)
   
   A. \( C[i+j-1] \leftarrow B_2[j] \) and \( j \leftarrow j + 1 \)

9. return \( C \)

Figure 1.1: The merge-sort algorithm.
Morally, when doing an asymptotic analysis in \( n \), we shouldn’t fret over \( \pm 1 \) 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 will also lead to a formally justified analysis for all \( n \).

1.2.4 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 example, the root of our tree corresponds to our initial call to `merge-sort(A[1..n])`. The root will have two children, corresponding to the subproblems for the recursive calls to `merge-sort(A[1..\lceil n/2 \rceil])` and `merge-sort(A[\lfloor n/2 \rfloor + 1..n])`. When doing an analysis like this, I will 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 will calculate the overall running time as follows. For \( i \in \mathbb{Z}_{\geq 0} \), let the \( i \)th level of the tree refer to the subproblems that are \( i \) generations away from the root. In our case, the root is the only node in level 0, its two children form level 1, its four grandchildren form level 2, and so forth. To upper bound the total computation, we divide it up 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. The overall running time is then bounded above by the product of the work-per-level and the total number of levels.

Fix a level \( i \). Let us label each subproblem with the size of the subproblem, and then the running time spent on that subproblem. In the \( i \)th 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 calls for recursive calls.

Next we identify the number of levels. Each level, the problem sizes divide by 2. Consequently, the leaves – corresponding to problems of size 1 – are at level (at most) \( O(\log n) \), and 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 special case where \( n \) is a power of two, for which the simpler recursion is correct. 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 \( T(n) \). Since \( n' \) is within a constant factor of \( n \),

---

\(^4\)Of course this seems true already just based on the merge-sort code, but it seems annoying to prove this formally.
1. Searching and sorting

1.3 Lower bounds for sorting

The function $T(n')$ 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

$$U(n) = T(n + 2)$$

$$\leq T(\lceil (n + 2)/2 \rceil) + T(\lceil (n + 2)/2 \rceil) + C(n + 2)$$

$$\leq 2T(n/2 + 2) + C(n + 2)$$

$$= 2U(n/2) + C'n$$

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, if we could have guessed that the running time was $O(n \log n)$, and then verified that it satisfies the recursion by induction.

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 $n$.

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 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

\[
\Omega(n)\text{, is the lower bound counterpart of } O(\cdot \cdot)\text{. Some quantity } x \text{ is } \Omega(f(n)) \text{ if there exists constants } C, N > 0 \text{ such that } x \geq Cf(n) \text{ for } n \geq N.
\]
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 $i$th comparison depends only on the outcomes of the preceding $i - 1$ comparisons. Ultimately, the algorithm outputs some reordered list as a function of only of 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

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

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 1.1.** 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: the 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 1.1 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. Let us point out that 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 1.4). 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.

### 1.4.1 Sorting with heaps

Recall that the *heap* data structure (cf. appendix 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.

\[
\text{heap-sort}(x_1, \ldots, x_n)
\]

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 1.2.** 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.

### 1.4.2 Quicksort

The following algorithm, called 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 also randomized way from merge-sort. 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 $O(n)$ items to do this.) We sort each set separately by recursive calls to quick-sort, and combine the lists at the end with $x$ in the middle.

```plaintext
quick-sort(x_1, ... , 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 $B = \{x_j : x_j > x_i\}$.

/* It takes $n - 1$ comparisons (between $x_i$ and each other element) to assemble $A$ and $B$. */

4. Recursively sort $A$ and $B$ (separately), and combine the sorted sublists with $x_i$ inserted in between. Return the combined list.

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** for 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 that quick-sort runs in $O(n \log n)$ time on average (for any list of $n$ items).

**Theorem 1.3.** 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 21.

Clearly the running time is proportional to the total number of comparisons. Let $y_i$ refer to the $i$th 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 $1/(j-i+1)$. Now, the key observation is that

by linearity of expectation (which is introduced in chapter 21), 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 $n$). This analysis requires additional probabilistic tools called concentration bounds.

### 1.4.3 Etcetera

There are many more possible sorting algorithms as well as heuristics 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.

---

*Here we use “heuristics” to refer to any techniques that (may be very useful but) do not theoretically improve the worst-case running times.*
1.5 Additional notes and references

The topics discussed here can also be found in [Eri19, §1.4–1.7], [DPV08, §2.3], [KT06, §5.1–5.2], and [Blu11b, Chap. 2 and 5]. Our discussion assumed some comfort with the subtle and important notion of induction. See [Che15] or [Eri19, Chapter 0] for more background on induction.

Lecture materials. Click on the links below for the following files:

- Handwritten notes prepared before the presentation.
- 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. 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 1.3. 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 \( i \)th 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 1.4.** 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 1.4?

**Exercise 1.5.** Consider coordinates in the plane; i.e., pairs $(a, b)$ where $a, b \in \mathbb{R}$. Recall that two coordinate pairs $(a, b)$ and $(c, d)$ 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 $n$ coordinate pairs $(a_1, b_1), \ldots, (a_n, b_n)$, consider the problem of computing the number of pairs of coordinate-pairs that are comparable. You may assume all pairs are distinct. Design and analyze an algorithm (as fast as possible) for counting the number of comparable pairs.

---

7We added this assumption on Wednesday, January 19, to make things a little simpler. Of course if you already solved the problem and can already handle duplicates then no adjustment is required.
Chapter 2

SAT: A different kind of search problem

2.1 Boolean formulas

A Boolean variable $x$ is one where one can take only one of two values, true and false. In computers, the Boolean values are represented by a single bit, with true = 1 and false = 0. We adopt the convention here as well. In Boolean algebra, we have 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 $\bar{x} \equiv \neg x$.

2. A **disjunction**, also called an or, is a binary operator denoted $x \lor y$ for two variables $x$ and $y$. We have $x \lor y = \text{true}$ if either $x$ or $y$ is true. If both $x$ and $y$ are false, then $x \lor y = \text{false}$.

3. A **conjunction**, also called an and, is a binary operator denoted $x \land y$. We have $x \land y = \text{true}$ if and only if both $x$ and $y$ are true; otherwise, $x \land y = \text{false}$.

These operations can be combined, using parentheses to notate the order of operations. For example, the formula

$$f(x_1, x_2) = (x_1 \land \bar{x}_2) \lor (\bar{x}_1 \land x_2)$$

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 \( x, y, z \in \{0, 1\} \),

\[
(x \lor y) \lor z = x \lor (y \lor z);
\]

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

\[
x \lor y \lor z
\]

without introducing any ambiguity. Conjunction \( \land \) is also associative. However, mixing \( \land \) and \( \lor \) is not associative: in general,

\[
(x \lor y) \land z \neq x \lor (y \land z).
\]

(Exercise 2.1 asks the reader for a counterexample.) Instead we have the following rules, called the **distributive property** of \( \lor \) and \( \land \).

\[
\begin{align*}
(x \lor (y \land z)) &= (x \lor y) \land (x \lor z), \\
(x \land y) \land z &= (x \land z) \lor (y \land z).
\end{align*}
\]

\( \neg \) interacts with \( \land \) and \( \lor \) by the following identity, called **De Morgan’s laws**.

\[
\begin{align*}
\neg(x \land y) &= \bar{x} \lor \bar{y}, \\
\neg(x \lor y) &= \bar{x} \land \bar{y}.
\end{align*}
\]

We leave it to the reader to verify these identities. Lastly we point out that disjunction and conjunction are **commutative**: \( x \land y = y \land x \), and \( x \lor y = y \lor x \).

Boolean formulas can have nested parentheses. The part of a formula corresponding to one pair of matching parentheses is called a **clause**. Nested parentheses describe 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

\[
f(x_1, x_2, x_3, x_4) = (x_1 \land x_2 \land x_3 \land \bar{x}_4) \\
\lor (\neg(x_1 \land x_2) \land ((x_1 \lor x_2) \land x_4) \lor (\neg(x_1 \lor x_2) \land \bar{x}_4))
\]

effectively implements 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 naturally try to program an 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 give an algorithm that also constructs the satisfying assignment (see exercise 2.2). 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 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 can 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:

\[
(y \land x) \lor (\overline{y} \land \overline{x}).
\]

Appending \( \land (y \land x) \lor (\overline{y} \land \overline{x}) \) 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

\[
(x \land y) \lor (x \land z).
\]

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?

2.2 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.

\[ f(x_1, x_2, x_3, x_4, x_5) = (\bar{x}_1 \lor x_2 \lor \bar{x}_3) \land (x_1 \lor x_4 \lor \bar{x}_5) \land (x_2 \lor \bar{x}_3 \lor x_4 \lor \bar{x}_5) \land (\bar{x}_2 \lor x_3 \lor x_4 \lor \bar{x}_5). \]

One example of a satisfying solution is the bit string 01011 satisfying the above. (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 \( (z \land x) \lor (\bar{z} \land \bar{x}) \), but this is not a CNF. To this end,

\[
(z \land x) \lor (\bar{z} \land \bar{x}) \overset{(d)}{=} (z \lor (\bar{z} \land \bar{x})) \land (x \lor (\bar{z} \land \bar{x})) \\
\overset{(b)}{=} (z \lor \bar{z}) \land (z \land \bar{x}) \land (x \lor \bar{z}) \land (x \lor \bar{x}) \\
\overset{(c)}{=} (z \lor \bar{x}) \land (x \lor \bar{z});
\]

a CNF, as desired. Here (a) and (b) are by the distributive property. (c) drops the tautologies \((x \lor \bar{x})\) and \((z \lor \bar{z})\).

2. **Conditional branching in CNF.** Suppose we want to model the Boolean statement “if \( x \) then \( y \) else \( z \)”. We have

\[
(x \land y) \lor (\bar{x} \land z) \overset{(d)}{=} (x \lor \bar{x}) \land (y \lor \bar{x}) \land (x \lor z) \land (y \lor z) \\
\overset{(e)}{=} (y \lor \bar{x}) \land (x \lor z) \land (y \lor z);
\]

a CNF, as desired. Here (d) is by the distributive property and (e) drops the tautology \((x \lor \bar{x})\).

For at least the above two cases, 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.
2. SAT: A different kind of search problem

2.3. Circuits

Theorem 2.1. Given any Boolean formula \( f(x_1, \ldots, x_n) \) of \( n \) variables and size \( m \), in polynomial time, one can construct a Boolean formula \( g(x_1, \ldots, x_n, x_{n+1}, \ldots, x_p) \) 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 2.4. (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 into a CNF formula \( g \) with 15 variables.

\[
g(x_1, \ldots, x_{15}) = (\bar{x}_{14} \lor x_1) \land (\bar{x}_{14} \lor x_2) \land (x_{14} \lor \bar{x}_1 \lor \bar{x}_2) \land (\bar{x}_{15} \lor x_3) \land (\bar{x}_{15} \lor x_4) \land (x_{15} \lor \bar{x}_3 \lor x_4) \land (x_{15} \lor \bar{x}_3 \lor x_4) \land (x_{15} \lor \bar{x}_1 \lor \bar{x}_2) \land (x_7 \lor x_1) \land (x_7 \lor \bar{x}_2) \land (x_7 \lor x_1 \lor x_2) \land (x_8 \lor x_1) \land (x_8 \lor \bar{x}_2) \land (x_8 \lor x_1 \lor \bar{x}_2) \land (x_9 \lor x_1 \lor \bar{x}_2) \land (x_9 \lor x_1 \lor x_2) \land (x_9 \lor \bar{x}_1 \lor \bar{x}_2) \land (x_9 \lor \bar{x}_1 \lor x_2) \land (x_10 \lor x_3 \lor x_4) \land (x_10 \lor \bar{x}_3 \lor x_4) \land (x_10 \lor x_3 \lor \bar{x}_4) \land (x_10 \lor \bar{x}_3 \lor \bar{x}_4) \land (x_10 \lor x_3 \lor x_4) \land (x_10 \lor \bar{x}_3 \lor x_4) \land (x_10 \lor x_3 \lor \bar{x}_4) \land (x_10 \lor \bar{x}_3 \lor \bar{x}_4)
\]

The formula \( g \) above has the following more precise 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 \( x_5, \ldots, x_{15} \in \{0,1\} \) such that \( g(x_1, \ldots, x_{15}) = 1 \).

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 2.2. Given any Boolean formula \( f(x_1, \ldots, x_n) \) of \( n \) variables and size \( m \), in polynomial time, one can construct a Boolean formula \( g(x_1, \ldots, x_n, x_{n+1}, \ldots, x_p) \) 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 3SAT implies a polynomial time algorithm for SAT.

The proof is left as exercise 2.5.

2.3 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.
2. SAT: A different kind of search problem

2.3. Circuits

Kent Quanrud
Spring 2022

Figure 2.1: A circuit computing the exclusive-or of two bits (at the top gate).

Figure 2.2: A circuit for adding two bits. Here, the gate at the top outputs true iff $x_1 + x_2 = x_3x_4$. 
2. SAT: A different kind of search problem

2.3. Circuits

Likely the reader has encountered Boolean circuits before and we briefly review the components. Fig. 10.2 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 ∧ 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 ∨ 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.

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 (x₁, x₂), (x₂, x₃), . . . , (xₖ, x₁) 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

¹One could consider evaluation over time, but this is a separate discussion.
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.

Let’s step back for a moment. Conceptually, this is really a question about directed acyclic graphs. Given a directed acyclic graph $G = (V, E)$, the goal is to order the vertices $V = \{v_1, \ldots, v_n\}$ so that every edge $(v_i, v_j) \in E$ is increasing in index; i.e., $i < j$. That is, we want to list the vertices so that all edges are directed from left to right. Such an ordering is called a topological order; the computational task is to topologically sort $V$.

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.

It would be convenient, then, to have a generic routine that takes as input a directed acyclic graph and outputs a list of its vertices in topological order. Consider, then, the following algorithm, which iteratively “peels” vertices from the graph. Below, recall that the in-degree of a vertex $v$ is the number of edges directed into $v$.

**Topological sort:**

1. While there is a vertex $v$ with in-degree 0:
   A. Append $v$ to the list.
   B. For every edge $(v, w)$ leaving $v$:
      1. Remove $(v, w)$ from the graph (decreasing the in-degree of $w$ by 1).
   C. Remove $v$ from the graph.
2. SAT: A different kind of search problem

2.4 Searching for answers

Lemma 2.3. Given a directed acyclic graph \( G = (V, E) \) with \( m \) edges and \( n \) vertices, the peeling algorithm above returns the list of vertices in topological order in \( O(m + n) \) time.

To make the routine run in linear time, as claimed, one needs to implement some obvious book-keeping like keeping the track of the in-degree of all the remaining vertices. We leave these details to the reader.

Conceptualizing topological sort, and commoditizing it as a linear time subroutine, gives us the following concise approach to evaluating a circuit.

*Topologically sort the gates of the circuit, and evaluate the circuits in topological order.*

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 encourage the reader to attempt the question themselves before proceeding.

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 2.6 to explain the construction explicitly. Such a construction implies the following theorem.

Theorem 2.4. There is a polynomial time algorithm for boolean SAT iff there is a polynomial time algorithm for circuit SAT.

2.4 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.2

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 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.

### 2.5 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.

#### 2.5.1 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

---

2This description oversimplifies the debate; there are other, more nuanced opinions and subtle related results.
2. SAT: A different kind of search problem

2.5. Additional results

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 \((x_1 \lor \overline{x}_2 \lor x_3)\). If \( x_1 = \text{true}, \) or \( x_2 = \text{false}, \) or \( x_3 = \text{true}, \) then the clause is satisfied. In fact the only way the clause is not satisfied is if \( x_1 = \text{false} \) and \( x_2 = \text{true} \) and \( x_3 = \text{false}. \)

That exact combination occurs with probability \( \frac{1}{8} \), 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 \( \frac{7}{8} \), 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 21.) But the point is that each clause contributes \( \frac{7}{8} \)th “satisfied clauses”, on average; consequently, if there are \( m \) total clauses, we expect to satisfy \( \frac{7m}{8} \) clauses. This implies, by the probabilistic method, that every 3SAT formula (automatically) has an assignment satisfying \( \frac{7}{8} \)-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 2.5.** 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 \( \frac{7}{8} \) 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 $\mu$. This means that either forcing $x_1 = \text{true}$ and randomly flipping the remaining $n - 1$ coins, or forcing $x_1 = \text{false}$ and randomly flipping the remaining $n - 1$ coins, has an average outcome of (at least) $\mu$. 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 2.6.** For all $k \in \mathbb{N}$, there is a deterministic $(1 - 1/2^k)$-approximation algorithm for $k$-SAT.

### 2.5.2 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 the proof in the Fall 2020 randomized algorithms course [Qua20b].

(More additional results to come.)

### 2.6 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 [AB09; Sip97; Wig19].

**Lecture materials and comments.** Click on the links below for the following files:

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

The lecture also had an interesting discussion about DNF, prompted by a question from the audience (paraphrasing): “If CNF is so hard, why not transform Boolean functions to DNF (disjunctive normal form), which is easier?” To further explore this question, we recommend exercise 2.13 (below) or exercise 3 in [Eri19, Chapter 12].

We did not have time to cover circuit SAT, which is not only interesting for its own sake and in connection to boolean SAT, but serves as a nice example to introduce topological sorting. We plan to weave some of these ideas back into later lectures; still, it would be good to read the corresponding section here.

2.7 Exercises

**Exercise 2.1.** Show that \( \land \) and \( \lor \) are not jointly associative by finding assignments \( x, y, z \in \{0,1\} \) such that \( (x \lor y) \land z \neq x \lor (y \land z) \).

**Exercise 2.2.** 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 2.3.** 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 \( \land \) and \( \lor \) would have two children. For example, the formula for XOR, \( f(x_1,x_2) = (\bar{x}_1 \land x_2) \lor (x_1 \land \bar{x}_2) \), is in this form.

**Exercise 2.4.** 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 \( O(m) \).

**Exercise 2.5.** 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.

**Exercise 2.6.** 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 \( O(m) \).

---

3 It might simplify things to first apply the preceding exercise.

4 As a second (appropriately vague) hint, how do you express “\( a = b \lor c \)” and “\( a = b \land c \)” in CNF?
Exercise 2.7. 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 $k$-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 2.8. 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 each of the following values of $k$, 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.\(^5\)

1. (8 points.) 5-occurrence-SAT (where each variable appears in at most 5 clauses).

2. (2 points.) 3-occurrence-SAT (where each variable appears in at most 3 clauses).

Exercise 2.9. 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 2.10. Recall that a palindrome is a string spelled the same forward or backwards, such as "racecar". Likewise we say that a bit string $x \in \{0,1\}^n$ is a palindrome 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 2.11. In the Max 2-SAT problem, one is given a CNF $f(x_1, \ldots, x_n)$ with exactly two variables per clause (i.e., a 2-CNF). The goal is to find an assignment that satisfies as many clauses as possible. (Note that we are not interested in knowing if $f$ is exactly satisfiable.) In the decision version of the problem, we

\(^5\)This problem was adjusted (again) on the morning of Jan 15. Previously "5"-occurrence-SAT was suggested as a hint; now it is just the main part of the problem. (Hopefully this makes it a little easier; those who solved it already in its previous, harder form, should not have much trouble adjusting to this version.)
are given a 2-CNF formula $f$ and an integer $k \in \mathbb{N}$, and the goal is to decide if there exists an assignment that satisfies at least $k$ clauses in $f$.

Either (a) show that a polynomial time algorithm for Max 2-SAT implies a polynomial time algorithm for boolean satisfiability (in general), or (b) design and analyze a polynomial time algorithm that, given a 2-CNF formula $f$, computes an assignment that maximizes the number of satisfied clauses.

**Exercise 2.12.** 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.

**Exercise 2.13.** A Boolean formula is in **disjunctive normal form (DNF)** if it is a disjunction of conjunctions. For example, for formula

$$f(x, y, z) = (x \land y) \lor (\bar{x} \land z),$$

which models “if $x$ then $y$ else $z$”, is in disjunctive normal form. Note that by the distributive property,

$$(a \lor b) \land (c \lor d) = (a \land c) \lor (a \land d) \lor (b \land c) \lor (b \land d),$$

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? (Why?)

**Exercise 2.14.** 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 $f = \varphi$.

### 2.A Proof of theorem 2.1

We left the proof of theorem 2.1 as exercise 2.4 (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 2.2 (which was left as exercise 2.5). (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 $f(x_1, \ldots, x_n)$ 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 $C_i$ in the tree by assigning its output to $y_i$.

To this end, observe that the expression “$x_1 = x_2 \land x_3$” can be expressed as the CNF

$$(\bar{x}_1 \lor x_2) \land (\bar{x}_1 \lor x_3) \land (x_1 \lor \bar{x}_2 \lor \bar{x}_3), \quad (2.1)$$

and the expression “$x_1 = x_2 \lor x_3$” can be expressed as the CNF

$$(x_1 \lor \bar{x}_2) \land (x_1 \lor \bar{x}_3) \land (\bar{x}_1 \lor x_2 \lor x_3). \quad (2.2)$$

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 (2.1) or (2.2) 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 (“$\land$’s”).
2. SAT: A different kind of search problem

2.A. Proof of theorem 2.1

\[
\begin{align*}
&x_{13} = \lor \\
&x_{14} = \land \\
&x_{15} = \land \\
&x_1 \quad x_2 \\
&x_{16} = \land \\
&x_3 \quad x_4 \\
&x_{17} = \lor \\
&x_1 \quad x_2 \\
&x_{18} = \land \\
&x_4 \\
&x_{19} = \lor \\
&x_1 \quad x_2 \\
&x_{20} = \land \\
&x_9 \\
\end{align*}
\]

\[
(\bar{x}_{14} \lor x_1) \land (\bar{x}_{14} \lor x_2) \land (x_{14} \lor \bar{x}_1 \lor \bar{x}_2) \\
(\bar{x}_{15} \lor x_3) \land (\bar{x}_{15} \lor x_4) \land (x_{15} \lor \bar{x}_3 \lor x_4) \\
(\bar{x}_6 \lor x_1) \land (\bar{x}_6 \lor x_2) \land (x_6 \lor \bar{x}_1 \lor \bar{x}_2) \\
(\bar{x}_7 \lor x_1) \land (\bar{x}_7 \lor x_2) \land (\bar{x}_7 \lor x_1 \lor x_2) \\
(\bar{x}_8 \lor x_4) \land (\bar{x}_8 \lor x_7) \land (x_8 \lor \bar{x}_4 \lor \bar{x}_7) \\
(\bar{x}_9 \lor x_1) \land (x_9 \lor \bar{x}_2) \land (\bar{x}_9 \lor x_1 \lor x_2) \\
(\bar{x}_{10} \lor \bar{x}_9) \land (\bar{x}_{10} \lor \bar{x}_4) \land (x_{10} \lor x_9 \lor x_4) \\
(x_{11} \lor \bar{x}_8) \land (x_{11} \lor \bar{x}_{10}) \land (\bar{x}_{11} \lor x_8 \lor x_{10}) \\
(\bar{x}_{12} \lor \bar{x}_6) \land (\bar{x}_{12} \lor x_{11}) \land (x_{12} \lor \bar{x}_6 \lor \bar{x}_{11}) \\
(x_{13} \lor \bar{x}_5) \land (x_{13} \lor \bar{x}_{12}) \land (\bar{x}_{13} \lor x_5 \lor x_{12}) \\
\]
Chapter 3

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 $i_1, i_2, \ldots, i_k \in [n]$. 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 $T$.

### 3.1 Brute force algorithms

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.

Perhaps the simplest approach is brute force: enumerate all subsets of the numbers, and return true if any of the subsets add up to $T$.

brute-force($\{x_1, \ldots, x_n\}, \ T$)

1. For all subsets $I \subset [n]$

   A. If $\sum_{i \in I} x_i = T$ then return true.

2. Return false.
This algorithm is (obviously) correct because it exhausts all possibilities. But it is also highly inefficient. The algorithm runs in roughly $O(n^2)$ time, broken down as looping over $2^n$ subsets of $[n]$, and for each summing up to $n$ numbers. As we have discussed before, $O(n^2)$ will never terminate even for modest values of $n$.

### 3.1.1 Recursion

Towards a more efficient algorithm, let us rewrite the brute force algorithm recursively. When designing a recursive algorithm, it is good practice to specify our function semantically. Such a spec implicitly establishes the induction hypothesis that ultimately justifies the recursion. In this case, such a description is fairly straightforward.

\[
\text{subset-sum}([x_1, \ldots, x_n], T) = \text{true} \text{ if there is a subset of } \{x_1, \ldots, x_n\} \text{ that sums to } T, \text{ and otherwise false.}
\]

Let us now implement what we have just defined.

```
subset-sum([x_1, \ldots, x_n \in \mathbb{N}], T)
1. If $T = 0$ then return true. // The empty set sums to 0
2. If $n = 0$ then return false. // The empty set sums to 0
3. If either one of
   A. subset-sum([x_2, \ldots, x_n], T) or
      // excludes $x_1$ from the subset sum
   B. subset-sum([x_2, \ldots, x_n], T - x_1)
      // includes $x_1$ in the subset sum

   returns true, then return true.
4. Otherwise return false.
```

It is easy to see the algorithm is correct by induction on $n^1$. The algorithm still enumerates all possible combinations, although this time it proceeds through the combinations one variable at a time. Consider the first time we call the function on the input $x_1, \ldots, x_n$ and $T$. The algorithm guesses whether or not to include $x_1$, as follows. It first checks to see if there is a feasible solution that doesn’t use $x_1$, by recursing on all the remaining numbers with the same target value $T$. In

---

^More precisely, it is easy to see that the pseudocode implements the specification for a given problem assuming the specification holds inductively for subproblems.
3. Subset-Sum, and the secret language of numbers

3.1. Brute force algorithms

the second recursive call, it checks to see if there is a feasible solution that uses $x_1$, by recursing on the remaining numbers with a new target value of $T - x_1$. By induction on $n$ these recursive calls will correctly decide if there is a subset sum for their respective subproblems. If either recursive call returns true, then there is a subset sum. If neither recursive call succeeds, then since any solution must either include or exclude $x_1$, we conclude that the subset sum problem is infeasible.

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

$$ T(n) = O(2^n). $$

This an improvement on the $O(n2^n)$ running time we started with, by a factor of $n$. While a factor of $n$ improvement is usually cause for celebration, here it is insignificant relative to $O(2^n)$. We are still stuck at exponential time algorithm. Can we do better?

3.1.2 Caching

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

$$ \text{subset-sum}(i, S) = \text{true if there is a subset of } \{x_i, \ldots, x_n\} \text{ that sums to } S, \text{ and false otherwise}. $$

The following pseudocode implements $\text{subset-sum}$ based on the new definition above, and is just a more compact version of our previous pseudocode.

$$ \text{subset-sum}(i, S) $$
1. if $S = 0$ then return true
2. if $S < 0$ then return false
3. if $i > n$ then return false
4. return $\text{subset-sum}(i + 1, S) \lor \text{subset-sum}(i + 1, S - x_1)$. 
To solve our initial problem, we call \( \text{subset-sum}(1, T) \).

Now, let us cache the answers to all our recursive calls so we never solve the same problem twice. This means that whenever we solve a subproblem for the first time, we also save the answer in some table or array. In every recursive call, we first check this table to see if we have already solved the subproblem. If so, then we can return the saved answer. If not, then we execute the pseudocode above, and also store the answer before returning it.

Note that we don’t have to cache the problems where \( S \leq 0 \) or \( i > n \) because these answers are fixed. We only save problems of the form \( \text{subset-sum}(i, S) \) where \( i \in [n] \) and \( S \in [T] \). The values can be stored in an \([n] \times [T]\) two-dimensional array (or in a hash table). The total size to store all the answers is \( O(nT) \). The pseudocode below gives an example of such an implementation.

```c
cached-subset-sum(i, S)

/* We assume there is an \( n \times T \) array denoted \( A[1..n, 1..T] \) (“A” for “Answer”) allocated in global memory, with all entries initially set to null. */

1. If \( S = 0 \) then return true.
2. If \( S < 0 \) then return false.
3. If \( i > n \) then return false.
4. If \( A[i, S] \) is null:
   A. \( A[i, S] \leftarrow \text{subset-sum}(i + 1, S) \lor \text{subset-sum}(i + 1, S - x_i) \).
5. Return \( A[i, S] \).
```

After adding caching to the pseudocode, each problem takes only constant time (excluding recursive calls). Thus the total running time becomes

\[
(# \text{ subproblems}) \cdot \left( \frac{\text{time per subproblem excl. recursive calls}}{} \right) = nT \cdot O(1) = O(nT).
\]

This running time looks much better than the previous, exponential running time of \( O(2^n) \). (More on this in a moment.)

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 3.2. (SAT was similarly self-reducible.) A more efficient approach is as follows. In the algorithm above, each subproblem \( \text{subset-sum}(i, S) \) 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 choices through a sequence of subproblems (starting from \texttt{subset-sum(1, T)}) that gives the actual set of numbers that add up to $T$.

3.1.3 Caching: a meta-algorithm

The difference in running time comes from the caching. It is a general technique that can be extended to all sorts of recursions, and is more commonly known as \textbf{dynamic programming}. The recipe is simple. Recursion, plus a little bit of memory to avoid redundancy, makes for more efficient algorithms.

The steps taken to convert the recursion into a more efficient dynamic programming algorithm were fairly mechanical and problem independent. When applying it to a problem, rather than going through these routine details again and again, the following points suffice to understand what is going on. (In this class, they are all required.) The following steps are also useful in helping you structure your thoughts and apply dynamic programming effectively, and with some practice, effortlessly.

1. A concise specification of the recursive function you are trying to implement, such as our one sentence description of \texttt{subset-sum(i, S)}. It should be clear what is the input, and what is the desired output. \textit{This is the induction hypothesis that justifies the correctness of your algorithm.}

2. Some kind of pseudocode or formula that implements the specification, such as the four line code for \texttt{subset-sum(i, S)} above.

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. An analysis of the space usage, which is often just the number of subproblems.$^2$

$^2$On February 13, I decided this was besides the main point being made in this algorithmic framework, and we will no longer check for this explicitly when grading. However it is still good to be mindful of how many subproblems are generated by your dynamic programming algorithm. I will also note that in some rare cases one can actually use less space than the number of subproblems (by being very careful in a non-recursive implementation) but again this is besides the pedagogical point.
6. A brief description of the recursive call that produces the answer we are actually looking for. Above, this is \( \text{subset-sum}(1, T) \).

7. When needed, some justification or proof for the correctness of the algorithm.

The critical thinking largely occurs in the first step, where we specify the function we want to implement. With a clear specification, the implementation in the second step can be straightforward. This is because recursive algorithms leverage induction, and a clearer specification gives a more useful induction hypothesis. Conversely, if it is difficult to implement the code in step 2, this may indicate that the specification is not quite right.

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. If one of the later steps feels awkward or difficult, then this might be best addressed by clarifying the specification or the pseudocode.

Steps that are not required include:

1. Literally implementing the caching (which is mechanical and clutters the code).

2. Providing code that iterates through the subproblems manually. (Recursion handles this automatically.)

3. The space usage (which is automatically bounded above by the running time, and rarely more interesting than that).

At some level we even discourage the above steps as they distract from the perspectives of top-down design and problem solving by induction.

Our first example of dynamic programming, \( \text{subset-sum} \), was relatively simple. But in upcoming discussions we will see how versatile this tool is, and the large design space it encumbers.

### 3.1.4 A polynomial running time?

Let us now return to \( \text{subset-sum} \). 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? \( O(n^T) \) 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 \( O(n^T) = O\left(n^{2\log(T)}\right) \) 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 have not actually succeeded in obtaining a polynomial time algorithm.

### 3.2 Packing in the complexity

Subset sum doesn’t look complicated, but it has resisted a number of attempts to obtain a (truly) efficient algorithm. To alleviate our frustration, 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 amounts of 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 important than we realized.

The main theorem of this section asserts precisely this, as follows.

**Theorem 3.1.** 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 \( f(x_1, \ldots, x_n) \), 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. But it is indeed possible. 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 \( f(x_1, \ldots, x_n) \) with \( n \) variables and \( m \) disjunctive clauses, where each clause has three distinct variables, possibly negated. We let \( C_1, \ldots, C_m \) denote the \( m \) clauses, and treat them as subsets of the set of symbols \( X = \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\} \). For example, a clause \((x_1 \lor \bar{x}_2 \lor x_3)\) 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 constraint \( C_i \subset X, X \cap S \neq \emptyset \).
2. For each Boolean variable \( x_j \), \( S \) contains exactly one of \( x_j \) or \( \bar{x}_j \).

Above, the set \( S \subset X \) indicate 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 \( x_j = \text{true} \) and \( \bar{x}_j \in S \) as assigning \( x_j = \text{false} \). 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 constraint \( C_i \) (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 \( A, B, C, \ldots \). For each \( x_j \), we will create a number \( A_j \) as follows. Recall that we have reserved a digit for each constraint and a digit for each variable. We describe \( A_j \) in terms of the value of each digit (out of 4).

1. For each constraint \( C_i \), if \( x_j \in C_i \), we set the \( C_i \)th 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 digit corresponding to \( x_j \) 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 constraint \( C_i \), if \( x_j \in C_i \), we set the \( C_i \)th digit (i.e., the digit corresponding to \( C_i \)) of \( \bar{A}_j \) to 1. Otherwise we set that digit to 0.
2. We set the digit corresponding to \( x_j \) 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 \( S \subseteq X \) 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 \( x_j \)th digit of \( D \) is exactly 1.
2. For every constraint \( C_i \), the \( C_i \)th 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?

The following adjustment effectively implements padding. For each constraint \( C_i \), we create two copies of the same number, called \( B_i \). \( B_i \) has value 1 in the \( C_i \)th digit and value 0 in all the other digits. The role of \( B_i \) is simply to allow us to artificially increase the \( C_i \)th digit of \( D \). In particular, with two copies of \( B_i \), we can add up to two units to the \( C_i \)th 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 \( C_i \)th digit of \( T \) to 3. For each variable \( x_j \), we set the \( x_j \)th digit of \( T \) to 1.
Observe that the size of our subset sum problem, \( O((m + n)^2) \), 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 3.1.

*The Boolean formula \( f \) is satisfiable iff there is a subset of the \( A_j \)'s, \( \overline{A}_j \)'s, and (2 copies each of the) \( B_i \)'s (as defined above) that adds up to \( T \).*

### 3.3 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 \( \log x \) 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.

Let us also point out that 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 3.6.) We may discuss approximation algorithms later, but for the moment, it is good to be aware of their existence and 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 cryptographic extensions of subset sum. See (e.g.) the notes [Pei13] for more details.

3.4 Additional notes and references

This lecture overlaps with [Eri19, §3.8] and [KT06, §6.4 and §8.8].

Lecture materials and comments. Click on the links below for the following files:

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

3.5 Exercises

Exercise 3.1. Suppose instead that we had reduced from 1-in-3-SAT to subset sum. How might this have simplified our construction in section 3.2?

Exercise 3.2. At the end of section 3.1.2, 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 3.1.2. 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 \texttt{true} if there exists a feasible solution, and otherwise \texttt{false}. Show that one can use this decision algorithm as a subroutine to obtain an efficient constructive algorithm for subset sum.
Exercise 3.3. Consider the following special case of subset-sum called the partition problem. The input consists of \( n \) numbers \( x_1, \ldots, x_n \in \mathbb{N} \). The goal is to partition the \( n \) numbers into two parts so that the sums of each part are equal. 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 3.4. Consider the following variation of subset sum which allows for an additive error of 1. We'll call it \texttt{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 3.5. 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 3.6. (Approximating subset sum.) Let \( \epsilon \in (0, 1) \) be fixed. Here we treat \( \epsilon \) as a fixed constant (like \( \epsilon = .1 \), for 10% error); in particular, running times of the form \( O(n^{O(1/\epsilon)}) \) 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.

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).
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 \( \text{SS}(i, S) = \text{true} \) if there is a subset of \( x_i, \ldots, x_n \) that sums to \( S \), and \( \text{false} \) otherwise.

2. Solving the original problem. The solution to the original problem is given by \( \text{SS}(1, T) \).

3. Recursive implementation.

\[
\text{SS}(i, S) \\
1. \text{If } S = 0 \text{ then return } \text{true}.
2. \text{If } S < 0 \text{ then return } \text{false}.
3. \text{If } i \leq 0 \text{ then return } \text{false}.
4. \text{Return the } \lor (\text{or}) \text{ of the following:}
   A. \text{SS}(i + 1, S)
   B. \text{SS}(i + 1, S - x_i).
\]

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 \( \text{SS}(i, S) \) 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 \( i \), \( \text{SS}(i + 1, S) \) will return \text{true}. If it does, then by induction on \( i \), \( \text{SS}(i + 1, S - x_i) \) will return \text{false}.

If there is no feasible solution, then there is no feasible solution to the subset sum problems described by \( \text{SS}(i + 1, S) \) or \( \text{SS}(i + 1, S - x_i) \). By induction on \( i \) both sub-calls will return \text{false}, hence so will \( \text{SS}(i, S) \), as desired.
Chapter 4
Sequential Optimization

4.1 \TeX

These notes (and hopefully yours too) are typeset using a system called L\LaTeX, which is an extension (by Leslie Lamport) of an earlier system called \TeX, designed and largely written by Donald Knuth. This is why it looks so nice\(^1\). In the 1970’s, Donald Knuth was writing his famous (and still unfinished) series on The Art Of Computer Programming. At the same time, the printing industry was shifting from old-fashioned hot metal typesetting to computer driven approaches. Knuth was so unsatisfied with the galley proofs of his book under the new system that he set out to invent his own digital typesetting system. Thus \TeX was born, and he began using it to typeset his books. \TeX was soon widely used in academia, and then a set of user-friendly macros by Lamport was bundled with \TeX to form L\LaTeX\(^2\).

\TeX takes unstructured text files as input and compiles it into PDF’s and other print formats, similar to how programming code is compiled into binary executables. This is in contrast to WYSIWYG editors, such as Microsoft Word, that immediately update a visual document as you type, and make their typesetting decisions immediately. The compilation step of \TeX gives it a chance to optimize various global criteria when making its many decisions, and ultimately produce professional quality documents.

Here we will discuss how \TeX does line-breaking (simplified for the sake of discussion), via an algorithm first proposed by Knuth in a 1977 memo\(^2\) and later published by Knuth and Plass [KP81]. When typesetting text, a basic decision one has to make is where to wrap the line. These decisions make a big difference on the overall visual product. Having too many words in a line squeezes the

\(^1\)Sadly, beautiful typesetting cannot otherwise improve this author’s prose.
\(^2\)https://www.sailedart.org/TEXDR.AFT%5E5B1,DEK%5D
white space and makes it harder to delineate words. Too few words leads to too much white space between them, which is distracting and also a waste of paper. By English convention, we restrict ourselves to breaking at white spaces between words or particular points where we can introduce a hyphen, rather than at all points within words. This chunking makes line wrapping that much clunkier.

One way to approach line-breaking is by the following greedy heuristic. Choose a minimum width for white space between words (e.g., half a centimeter). Each time we consider adding the next word to a line, calculate the amount of white space that would be between each pair of words if the new word were included. If this calculated value is above the minimum, then add the word to the line. If the value is below the minimum, then start a new line.

The greedy approach observes very local information in making its decisions. The fact that the choice of words in one line affects the possible options in all the remaining lines is overlooked.

Towards a more comprehensive approach, let’s try to formulate the overall problem clearly. I think we can all agree that we want the “best overall set of line breaks, over all possible ways of breaking up the text.” But what does it mean to be “best”? Knuth proposed we use a scoring function to quantify it. He proposed a loss function \( \text{loss} \) with the following format. \( \text{loss} \) takes as input a sequence of words \( w_1, \ldots, w_k \), imagines these words as a line, and outputs a numerical score that measures how “bad” the line would look. Here we treat \( \text{loss}(w_1, \ldots, w_k) \) as a black box, and refer to [KP81] for implementation details. Our goal, then, is to take a long sequence of words \( w_1, \ldots, w_n \), and divide them up into (consecutive) sub-sequences \( (w_1, \ldots, w_{i-1}), (w_i, \ldots, w_{i+1}), \ldots, (w_k, \ldots, w_n) \) to minimize the total loss,

\[
\text{loss}(w_1, \ldots, w_{i-1}) + \text{loss}(w_i, \ldots, w_{i+1}) + \cdots + \text{loss}(w_k, \ldots, w_n).
\]

This establishes a formal definition of the optimal line breaking. Given text in the form of a sequence of words, it is the partition of the sequence of word into intervals that minimizes the sum, over all such partitions, of the loss of that interval. This is well-defined because one can enumerate the finite number of ways to partition the text into intervals and for each compute the corresponding loss; the minimum loss gives the optimum line-breaking (as defined). The only issue is whether it can be computed efficiently.

Let’s now define a recursive algorithm to compute the optimal line breaking. We start with a recursive spec. Fix a sequence of words \( w_1, \ldots, w_n \) that we want to partition. We want to compute the minimum loss of any partition of the words \( (w_1, \ldots, w_n) \) into intervals. For \( i \in \mathbb{N} \), we define \( \text{BLB}(i) \) (which stands for “best line-breaking”) as

\[
\text{BLB}(i) = \text{minimum loss of any partition of the sequence of words } (w_i, \ldots, w_n) \text{ into intervals.}
\]
4. Sequential Optimization

4.2. Edit distance

We are precisely interested in $BLB(1)$. Consider any subproblem of the form $BLB(i)$. Note that if $i > n$ then there are no words to partition and the minimum loss is automatically 0. So we write:

$$BLB(i) = 0 \quad \text{if } i > n. \quad (4.1)$$

Consider the more interesting case where $i \leq n$. Any partition of $(w_i, \ldots, w_n)$ into intervals must, in particular, have a first interval $(w_i, \ldots, w_j)$ that includes $w_i$, followed by a partition of all the words $(w_{j+1}, \ldots, w_n)$ that come after that first interval. For the remaining partition, we can use induction (on $n - i$) to obtain the optimum partition. We do not know, however, where to end the first interval. And after thinking about it, it is not clear how to strategically choose that first interval, at least without much more info about the inner mechanisms of loss. But we don’t have to be clever. We can just try all possible choices; one of them will be optimal. Thus we have

$$BLB(i) = \min_{j=i, \ldots, n} \{ \text{loss}(w_i, \ldots, w_j) + BLB(j+1) \} \quad \text{if } i \leq n \quad (4.2)$$

Between (4.1) and (4.2) we have a full recursive algorithm. Consider now the running time. Let $T(n)$ be the running time required to compute $BLB$ over $n$ words. We have

$$T(0) = 1,$$

$$T(n) = \sum_{i=0}^{n-1} T(i) + O(n).$$

This running time, if we worked it out, is enormous - much larger than $2^n$. But let us cut off the calculations and intervene with the computer scientist’s favorite (only?) trick: caching. Obviously, we are solving the same problem $BLB(i)$ many times even though the answer is always the same. So we apply dynamic programming: we store our answers in an $n$-dimensional array and never compute the same answer twice. Now the running time becomes

$$(\# \text{ subproblems}) \times (\text{time per subproblem}) = n \times O(n) = O\left(n^2\right).$$

The space usage is $O(n)$. And your notes look great.

4.2 Edit distance: An algorithmic metric for strings

We are constantly discussing bit strings in this course, and in programming at large, we are constantly parsing, manipulating, and outputting strings. (Here,
besides bit strings, we can speak more generally of strings that are sequences of characters from some fixed alphabet $\mathcal{A}$, like the ASCII characters). Naturally, then, many applications required a method to compare two strings, and some way to quantify the extent of their similarity. That is, we seek some notion of “distance” between strings. One possible notion is **Hamming distance**, which simply counts the number of differing characters in two strings of the same length. That is, for a fixed alphabet $\mathcal{A}$, and two strings $x, y \in \mathcal{A}^k$ of the same length, the Hamming distance is defined as the number of differing bits:

$$\text{Hamming}(x, y) = \left| \{i \in [k] : x_i \neq y_i \} \right|.$$ 

We note that in some applications, the Hamming distance is normalized to be in $[0, 1]$, by dividing the above definition by the length $k$. Note that the Hamming distance is symmetric and two strings have Hamming distance 0 iff they are equal. One can also verify that the Hamming distance satisfies the triangle inequality:

$$\text{Hamming}(x, y) \leq \text{Hamming}(x, z) + \text{Hamming}(z, y)$$

for any three strings $x, y, z \in \{0, 1\}^k$. (Why?) Thus Hamming distance satisfies the three technical requirements of a *metric*, just like in Euclidean geometry.

The Hamming distance is natural and suitable for many definitions (e.g., in coding theory). But it has some weaknesses as well. First, it is only defined when two strings have the same length. Sure, given two strings of different length, one can truncate the longer one to match the shorter one, or pad the shorter one to match the longer one, but it is not clear that the Hamming distance is as meaningful then. (Also, it would no longer be a metric - why?) Besides this issue of length, the Hamming distance may not always correspond to our intuition for similarity. Suppose we had a string $x$. Suppose we remove the first character from the beginning, and appended this character to the end, to form a new string $y$. On one hand, we made a very small change to take $x$ into $y$. On the other

---

3Richard 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.

4That is, $\text{Hamming}(x, y) = \text{Hamming}(y, x)$ for all $x$ and $y$. 

---

4. Sequential Optimization

4.2. Edit distance
hand, the Hamming distance and \( x \) and \( y \) will be tremendous, because all the characters shift by an index. Let us give two real life examples where something similar frequently occurs.

When we write software, we frequently insert or delete a line in the middle of a program. The rest of the program is the same, and overall we think of this as a relatively small change. But in terms of Hamming distance, the misalignment that we introduce gives the impression that the entire program has turned over. This is an important consideration in version control systems (such as \texttt{git})

A second example is from biology. We now know that we are described by a genetic code. For the sake of discussion, these are long strings from an alphabet of four letters, \{A,T,G,C\}, each standing for a different molecule. These genetic codes frequently undergo mutations, where one letter is swapped for another, or letters are dropped or inserted along the code. This too introduces some misalignment, but functionally the altered mutated code is not so different. The Hamming distance would suggest an entirely different species, but biologically this is just not so.

A more robust measure than Hamming distance, that better models the number of changes made to a string, is called \textbf{edit distance}. The edit distance between two strings \( s, t \in \mathcal{A}^* \) is the minimum number of \textit{insertions}, \textit{deletions}, and \textit{substitutions} required to convert one string to another. These operations are defined as follows.

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.

Let us explicitly point out that edit distance is now completely well-defined. The edit distance between two strings \( x \) and \( y \) is bounded below by 0, and bounded above by the length of \( x \) plus the length of \( y \). It is also computable, by brute force - e.g., trying all sequences of edits with length at most the aforementioned upper bound.

Let \( \text{edit}(x,y) \) denote the edit distance between \( x \) and \( y \). Note that the definition is symmetric, in the sense that the edit distance from \( x \) to \( y \) is equal to the edit distance from \( y \) to \( x \). (This is because any sequence of edits converting \( x \)
4. Sequential Optimization

4.2. Edit distance

Kent Quanrud
Spring 2022

Hamming distance

Hamming characters differing in $X$ and $y$

$X = $ ABCDEFG

$Y = $ ABCEFGD

$\Rightarrow$ Edit distance $= 2$

Figure 4.2: An example of edit distance

to $y$ can be reversed to give a sequence of edits converting $y$ to $x$). It also satisfies the triangle inequality. Indeed, let $x, y, z$, and suppose we want to show that

$$\text{edit}(x, y) \leq \text{edit}(x, z) + \text{edit}(z, y).$$

To this end, observe that any sequence of edits from $x$ to $z$, followed by a sequence of edits from $z$ to $y$, gives a sequence of edits from $x$ to $y$. Thus $\text{edit}(x, z) + \text{edit}(z, y)$, being the length of some edit sequence from $x$ to $z$, gives an upper bound on the minimum length sequence of edits from $x$ to $y$. Thus $\text{edit}(x, y)$ is also a metric between strings, like Hamming distance. Unlike Hamming distance, it gracefully handles strings of different, arbitrary lengths. For many applications, such as code versioning and genetic comparisons, it is a much more useful measure of distance between strings.

With the edit distance now well-defined, let us progress towards a more efficient algorithmic implication. It turns out that edit distance is frequently used for large data – genetic codes and source codes can both be very large – and so particularly efficient algorithm can be helpful.

Before, we lazily pointed out that edit distance can always be computed by brute force. Of course this is very slow and we won’t be able to get a better algorithm without at least getting our fingernails dirty. Following the pattern of subset sum, let us first try to develop some cleaner and more precise recursions. We start from the easy base cases. If both strings are empty, then the edit distance is 0. Let’s write that down.

$$\text{edit}(\emptyset, \emptyset) = 0,$$

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.

$$\text{edit}(x[1..k], \emptyset) = k$$

$$\text{edit}(\emptyset, y[1..\ell]) = \ell$$

In the interesting case, we have two strings $x$ and $y$ that are nonempty. Suppose first that the first characters $x[1] \neq y[1]$. Any edit sequence from $x$ to $y$, simply to make the first character match, must either (a) insert $y[1]$ before $x[1]$, (b)
delete $x[1]$, or (c) substitute $y[1]$ for $x[1]$. If not, then the first character at the
end would have to be $x[1]$, a contradiction. Out of the above three choices, it
is not obvious what is the best option. One might think that we should always
substitute, since it seems to address two letters ($x[1]$ and $y[1]$) at the cost of a
a mistake, and disrupting a “match” later on. In general, it’s not clear the effects
of the current decision on downstream decisions. Instead of analyzing further,
let us be lazy, and simply brute force over the three options. Thus, if $x[1] \neq y[1]$,
we have

$$
\text{edit}(x[1..k], y[1..\ell]) = 1 + \min \begin{cases} 
\text{edit}(x[2..k], y[1..\ell]) \\
\text{edit}(x[1..k], y[2..\ell]) \\
\text{edit}(x[2..k], y[2..\ell]) 
\end{cases}
$$

The first quantity 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.

The other case is where $x[1] = y[1]$. In this case, we can simply leave $x[1]
and $y[1]$ alone, and focus on editing $x[2..k]$ and $y[2..\ell]$. Thus, if $x[1] = y[1]$,
then

$$
\text{edit}(x[1..k], y[1..\ell]) \leq \text{edit}(x[2..k], y[2..\ell]).
$$

Note that we wrote an inequality, rather than an equality, because technically
speaking, we have only upper bounded $\text{edit}(x[1..k], y[1..\ell])$. It seems obvious
that one should match $x[1]$ to $y[1]$, and indeed one can prove it with some care.
But let us remain lazy and instead do the following. If (for whatever reason) we
don’t want to match $x[1]$ to $y[1]$, then we must either insert $y[1]$ or delete $x[1]$.
(There’s no point in substituting when $x[1] = y[1]$ already.) Thus, if $x[1] = y[1]$, we have

$$
\text{edit}(x[1..k], y[1..\ell]) = \min \begin{cases} 
\text{edit}(x[2..k], y[2..\ell]) \\
1 + \text{edit}(x[2..k], y[1..\ell]) \\
1 + \text{edit}(x[1..k], y[2..\ell]) 
\end{cases}
$$

This last formula finishes covering all cases, and so we have a recursive
definition for edit-distance. We gather the preceding discussion in a single
pseudocode below. As this code is more or less brute force, it will correctly
compute the edit distance; for the same reason, it will be incredibly inefficient.
4. Sequential Optimization

4.2. Edit distance

Efficiency is particularly important for edit distance because it is frequently applied to large data sets, such as source code and genetic sequences. Currently, we have more or less a brute-force, recursive algorithm. We saw this previously both with subset sum and TEX, so we might as well try the same trick: cache all our answers so we never solve the same problem twice; a.k.a. dynamic programming.

To make the data structure component simpler, let us be more concrete about the indexing as follows. Let the two strings $x[1..m]$ and $y[1..n]$, whose edit distance we want to compute, be fixed. We define

$$\text{edit}(i, j) = \text{the minimum number of edit operations to convert } x[i..m] \text{ into } y[j..\ell].$$

Then $\text{edit}(1, 1)$ gives the edit distance between $x$ and $y$. A recursive implementation of $\text{edit}(i, j)$ is as follows.

$$\text{edit}(i, j) = \begin{cases} 0 & \text{if } i > m \text{ and } j > n \\ 1 + \text{edit}(i, j + 1) & \text{if } i > m \\ 1 + \text{edit}(i + 1, j) & \text{if } j > n \\ 1 + \min \left\{ \begin{array}{l} \text{edit}(i + 1, j), \\ \text{edit}(i, j + 1), \\ \text{edit}(i + 1, j + 1) \end{array} \right\} & \text{if } x[i] \neq y[j] \\ \min \left\{ \begin{array}{l} 1 + \text{edit}(i, j + 1), \\ 1 + \text{edit}(i, j) \end{array} \right\} & \text{if } x[i] = y[j] \end{cases}.$$
4.3 Sequential matrix multiplication

Recall that given two matrices \( A \in \mathbb{R}^{n_1 \times n_2} \) and \( B \in \mathbb{R}^{n_2 \times n_3} \), the product \( AB \) is defined coordinatewise by

\[
(AB)_{i,j} = \sum_{k=1}^{n_2} A_{i,k} B_{k,j}.
\]

Note that this is a sum of \( n_2 \) terms. (I think of this as “zipping up” along the inner dimension.) It takes \( O(n_1 n_2 n_3) \) to compute all \( n_1 n_3 \) coordinates of \( AB \).

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

\[
A_1 A_2 A_3.
\]

While matrix multiplication is defined pairwise, it is also associative; that is,

\[
A_1 (A_2 A_3) = (A_1 A_2) A_3.
\]

Consequently, we can write \( A_1 A_2 A_3 \) unambiguously.

Computationally, however, there is a distinction between computing \( A_1 A_2 A_3 \) as the product \( A_1 (A_2 A_3) \) or as the product \( (A_1 A_2) A_3 \).

Consider first computing the product as \( A_1 (A_2 A_3) \). Computing \( (A_2 A_3) \) takes \( O(n_2 n_3 n_4) \) time and produces a \( n_2 \times n_4 \)-dimension matrix. Multiplying \( A_1 (A_2 A_3) \) then takes \( O(n_1 n_2 n_4) \) time. Overall we spent \( O(n_2 n_3 n_4 + n_1 n_2 n_4) = O((n_1 + n_3) n_2 n_4) \) time.

\footnote{We can verify this directly: for \( i_1 \in [n_1] \) and \( i_4 \in [n_4] \), we have

\[
(A_1 (A_2 A_3))_{i_1,i_4} = \sum_{j_2=1}^{n_2} A_{1,i_1,j_2} (A_2 A_3)_{j_2,i_4} = \sum_{j_2=1}^{n_2} A_{1,i_1,j_2} \sum_{j_3=1}^{n_3} A_{2,j_2,j_3} A_{3,j_3,i_4} = \sum_{j_2=1}^{n_2} \sum_{j_3=1}^{n_3} A_{1,i_1,j_2} A_{2,j_2,j_3} A_{3,j_3,i_4};
\]

similarly one can show that \( ((A_1 A_2) A_3)_{i_1,i_4} = \sum_{j_2=1}^{n_2} \sum_{j_3=1}^{n_3} A_{1,i_1,j_2} A_{2,j_2,j_3} A_{3,j_3,i_4} \) by expanding out the appropriate sums.}
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. So we spend in total
\[
O(n_1n_2n_3 + n_1n_3n_4) = O(n_1n_3(n_2 + n_4))
\]
total time.

So there is a difference in running time. If for example, \(n_1 = n_3 = 1\) and \(n_2 = n_4 = \ell\) for \(\ell\) very large, the first approach, \(A_1(A_2A_3)\) would take \(O(\ell^2)\) time, while the second approach would take \(O(\ell)\) time.

More generally, suppose we had a sequence of \(k\) matrices \(A_1, \ldots, A_k\) where each \(A_i\) is an \(n_i \times n_{i+1}\)-dimensional matrix. We want to compute the product \(A_1A_2 \cdots A_k\) and can only apply multiply two matrices at a time. Consider the optimization problem of identifying the sequence of pairwise matrix multiplications to produce the product \(A_1A_2 \cdots A_k\) with as few numerical multiplications as possible. (Once we find the best sequence, then we can follow the sequence to actually compute the matrix product.) One way to think of a solution is by grouping the pairwise products with parentheses; for example, the following lays out an approach for 8 matrices.
\[
((A_1A_2)((A_3A_4)A_5))((A_6A_7)A_8)
\]

We can also envision the multiplications as a tree:

```
          ((A_3A_4)A_5)((A_6A_7)A_8)
          |        |
(A_1A_2)  (A_6A_7)A_8
    |            |
A_1  A_2  A_6A_7
    |
A_3  A_4
```

With these preliminary observations we pause to let the reader to try to solve the problem for themselves.

Let us first mention two greedy rules that seem natural and are commonly suggested, but one can devise counterexamples that show they are not 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 \(n_1 n_i n_{k+1}\), and then recursively compute the two halves \((A_1 \cdots A_i)\) and \((A_{i+1} \cdots A_k)\) in the same fashion.

Let us instead take a more conservative, recursive approach. The first step, as usual, is to develop a recursive specification that also gives an algorithmic induction hypothesis. This is essentially the same task as identifying the write subproblems to break down the original problem. We will consider the subproblems induced by subsequences \(A_iA_{i+1} \cdots A_j\) of the input. We define:

\[ \text{Fastest}(i, j) \] as the number of multiplications of the fastest way to multiply \(A_i \cdots A_j\).  

We now implement the pseudocode as follows.

\[
\text{Fastest}(i, j) \\
1. \text{If } i \geq j \text{ then return } 0.
2. \text{Return the minimum, over all } k = i, \ldots, j - 1, \text{ of } \\
   \text{Fastest}(i, k) + \text{Fastest}(k + 1, j) + n_i n_{k+1} n_j.
\]

To justify the pseudocode, fix an instance \(\text{Fastest}(i, j)\). Inductively (on the number of matrices) we assume that the subcalls to \(\text{Fastest}\) with smaller problems are correct. In this case, for each \(k\),

\[
\text{Fastest}(i, k) + \text{Fastest}(k + 1, j) + n_i n_{k+1} n_j
\]
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. Which \(k\) gives the best initial split? \textit{A priori}, who knows? So instead we exhaustively test all choices of \(k\) and return the best one.

Of course \(\text{Fastest}(1, k)\) would be exponentially slow as is. But if we apply dynamic programming then the running time is greatly reduced, to

\[
(\# \text{ subproblems}) \times (\text{time per subproblem}) = k^2 \times O(k) = O(k^3).
\]

The space usage is \(O(k^2)\).

Note that \(\text{Fastest}(1, k)\) only reports the minimum number of numerical multiplications required to produce the product \(A_1 \cdots A_k\). Of course we also want the actually sequence of operations that produces it. Here we can make a second table that records, for each \(\text{Fastest}(i, j)\), the choice of index \(\ell\) the
lead to the value obtained in Fastest\((i, j)\). Starting from the root problem Fastest\((1, k)\), we can follow the pointer of choices to reconstruct the optimal sequence of multiplications. (See also a related discussion on page 51, for subset sum.)

### 4.4 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].

**Lecture materials and comments.** Click on the links below for the following files:

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

### 4.5 Exercises

In addition to some of the exercises below, there are many freely available exercises with similar themes as this chapter. See for example the exercises in [Eri19, Chapter 3].

#### Exercise 4.1.
Let \(A[1..n]\) be an array of \(n\) numbers. Consider the problem of computing the length of the longest subsequence of \(A\) that is strictly increasing. Here a sequence of numbers \(x_1, \ldots, x_k\) is strictly increasing if \(x_i < x_{i+1}\) for \(i = 1, \ldots, k - 1\). 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 4.2.
Recall that a **palindrome** is a string that reads the same forwards and backwards. For example, 

\[
{\text{mom, dad, racecar, and gohangasalamitalasagahnog}}
\]
are all palindromes. Consider the problem where you are given a string \(A[1..n]\) as input, and the goal is to find the (length of the) longest palindrome that is a
subsequence of $A$. 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 4.3.** Consider the line-breaking problem from section 4.1. 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 4.1 to give a faster running time (in terms of $n$ and $k$).

**Exercise 4.4.** Consider the edit distance problem from section 4.2. In section 4.2, 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 4.2 for these nonuniform costs. This model is useful in biological settings where $\alpha, \beta, \gamma$ reflect prior assumptions on the likelihood of each operation.

In section 4.2 as well as in the previous exercise, we considered edit distance with uniform or linear costs. The next two problems consider more involved cost functions.

**Exercise 4.5.** 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 at a time. We can factor this propensity into our edit distance model by a “insert-or-delete-at-bulk” cost. Let $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ 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. 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 4.6.** 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.

Exercise 4.7. Let $A[1..m]$ be a sequence of integers. Recalling that a subsequence is (strictly) increasing if each successive integer is strictly greater than the previous.

Here we consider two variants of the longest increasing subsequence problem. In the first problem, we want to find the longest increasing subsequence where the sum of integers in the subsequence is even. In the second problem, we want to find the longest increasing subsequence where the sum of integers in the subsequence is odd.

For both of these 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.

Exercise 4.8. Let $A[1..m]$ be an array of integers, and let $T$ be an additional number. The problem is to find the length of the longest 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 4.9. 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 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.

Exercise 4.10. Recall that previously we computed the length of the longest increasing subsequence of an input sequence of numbers. Here, instead of increasing numbers, we are interested in “wavy” sequences of numbers that oscillate up and down. More specifically, we say that a sequence of numbers $y_1, y_2, \ldots, y_k$ is:
1. **Up-down wavy** if \( y_i < y_{i+1} \) for all odd \( i \), and \( y_i > y_{i+1} \) for all even \( i \).

2. **Down-up wavy** if \( y_i > y_{i+1} \) for all odd \( i \), and \( y_i < y_{i+1} \) for all even \( i \).

3. **Wavy** if it is either up-down wavy or down-up wavy.

(“Up-down wavy” sequences go up, then down, at the beginning. “Down-up wavy” sequences go down, then up, at the beginning.) For example, the sequence 1, 9, 2, 8, 3, 7, 4, 6 is an up-down wavy sequence. The reverse sequence, 6, 4, 7, 3, 8, 2, 9, 1 is a down-up wavy sequence.

Given an array of integers \( X[1..n] \), the goal is to compute the length of the longest wavy subsequence of \( X \).\(^{10}\) 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 4.11.** Exercise 4.2 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.

\[
\begin{array}{cccccccc}
\text{x} & \text{r} & \text{y} & \text{a} & \text{c} & \text{z} & \text{e} & \text{c} \\
8 & 5 & 1 & 3 & 7 & 4 & 2 & 4 \\
\end{array}
\]

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 \( A[1..n] \). 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.

---

\(^{10}\) As usual, the subsequence does not have to be a consecutive subsequence of \( X \). It could be like \( X[1], X[4], X[9] \), and so forth, skipping over some entries.
Sample solution.

**Problem.** The following problem models the line breaking problem from section 4.1. The input consists of an array \( X[1..n] \) where each \( X[i] \) represents a word. We are also given a “loss function” \( \text{loss}(i,j) \) 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 \( \text{loss}(i,j) \) takes constant time to evaluate.

The high-level goal is to partition \( X \) into contiguous subsequences

\[
X[1..t_1], X[t_1+1..t_2], X[t_2+1..t_3], \ldots, X[i_k-1+1..n]
\]

minimizing the total loss:

\[
\text{loss}(1,t_1) + \text{loss}(t_1+1,t_2) + \text{loss}(t_2+1,t_3) + \cdots + \text{loss}(i_k-1+1,n).
\]

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

\[
\text{min-loss}(i) = \text{the minimum loss over all partitions of } X[i..n]. \quad \text{(For } i > n, X[i..n] \text{ indicates the empty sequence.)}
\]

2. **Recursive implementation.**

\[
\text{min-loss}(i)
\]

   1. If \( i > n \) then return 0.

   2. Otherwise return the minimum, over all \( j = i, i+1, \ldots, n \), of

   \[
   \text{loss}(i,j) + \text{min-loss}(j+1).
   \]

3. **Solving the original problem.** The solution to the problem is given by \( \text{min-loss}(1) \).

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 \( i = 1, \ldots, n \), \( \text{loss}(i) \) 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.
Chapter 5

Optimization in graphs

The two preceding discussions introduced some basic combinatorial problems, such as subset sum and edit distance, that lead to surprisingly different outcomes. Edit distance had a polynomial algorithm, while subset sum was revealed to be as general as SAT (up to polynomial time reductions), hence unlikely to yield a polynomial running time. What’s the difference between subset sum and edit distance? Subset sum is not obviously harder than edit distance on first appearance. But for edit distance we were able to identify subproblems that decompose the initial problem into polynomially many subproblems; for subset sum, similar attempts came up short.

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.

5.1 Independent set

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.

5.1.1 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 $S \subset V$. 

![Figure 5.1: An independent set of size 5.](image-url)
We first observe that such a set $S$ can be found by brute force. Here is a recursive algorithm.

$$\text{MWIS}(G = (V, E), \ w : V \to \mathbb{R}_{\geq 0})$$

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. $\text{MWIS}(G - v, w)$ where $G - v$ denotes the graph obtained by removing $v$ and all its incident edges from $G$.
   
   // all independent sets including $v$
   B. $w(v) + \text{MWIS}(G', w)$ where $G'$ obtained by removing $v$ and its neighbors\(^a\) from $G$.

\(^a\)Two vertices $u$ and $v$ are **neighbors** if they are connected by an edge.

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 5.1.** 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 $C = (x_1 \lor \bar{x}_2 \lor x_3)$, we create a triangle with three
5. Optimization in graphs

5.1. Independent set

Kent Quanrud  
Spring 2022

Figure 5.2: The gadget encoding disjunctions over three variables as triangles.

\[ (a \lor b \lor c) \land (b \lor c \lor \bar{d}) \land (a \lor c \lor \bar{d}) \land (a \lor \bar{b} \lor \bar{d}) \]

Figure 5.3: Encoding the 3-SAT formula \( f(a, b, c, d) = (a \lor b \lor c) \land (b \lor c \lor \bar{d}) \land (a \lor b \lor \bar{d}) \) as a graph, where the maximum independent set indicates whether or not \( f \) is satisfiable.

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. So far, for \( m \) clauses, we have created a graph of \( m \) disjoint triangles, with \( 3m \) vertices and \( 3m \) edges in all. Note already that any independent set in \( G \) has at most one vertex per clause/triangle.

Our graph does not yet encode the fact that we can only set each variable \( x_i \) to be exactly one of true or false, and not both. Currently, we can take an independent set that (for example) contains vertices for both \( x_1 \) and \( \bar{x}_1 \). To encode this constraint, we add an edge between every pair of vertices of the form \( x_i \) and \( \bar{x}_i \), for all \( i \). With these additional edges, an independent set can no longer include variables corresponding to both \( x_i \) and \( \bar{x}_i \).

This completes the constructions. In total the graph has \( O(m) \) vertices and \( O(m + n) \) edges. A minimal example is given in fig. 5.2. A more ornate example is given in fig. 5.3.

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 $x_i = 1$ if any of the $x_i$-vertices appear in $S$, or we set $x_i = 0$ 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 $x_i = 1$ in the satisfying assignment, and we only choose a vertex corresponding to the symbol $\bar{x}_i$ if $x_i = 0$. In particular, we can’t choose both. So $S$ will be independent over the second set of edges as well. Thus $S$ 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.

5.1.2 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 $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$ 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 $w : \mathcal{I} \rightarrow \mathbb{R}_{>0}$, we define

$$\text{MWIS-intervals}(\mathcal{I}, w) = \text{the weight of the maximum weight independent subset of intervals of } \mathcal{I}.$$ 

We now implement $\text{MWIS-intervals}$ as specified above.

$$\text{MWIS-intervals}(\mathcal{I}, w : \mathcal{I} \rightarrow \mathbb{R}_{>0})$$

1. If $\mathcal{I} = \emptyset$ then return 0
2. Let $I \in \mathcal{I}$ be any interval
3. Let $\mathcal{I}' = \{ J \in \mathcal{I} : J \cap I = \emptyset \}$  
   \[\text{set of intervals disjoint from } I\]
4. return the maximum of 

   /* The MWIS among those excluding $I$. */
   A. $\text{MWIS-intervals}(\mathcal{I} \setminus \{I\}, w)$

   /* The MWIS among those including $I$. */
   B. $w(I) + \text{MWIS-intervals}(\mathcal{I}', w)$

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.
5. Optimization in graphs

5.1. Independent set

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 $\{I_1, \ldots, I_n\} =$ 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($\{I_1, \ldots, I_n\}$, $w$)

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. This limits the number of combinations of intervals in our subproblems: there are only $n + 1$ suffixes (including the empty set). We have an ideal situation for caching.

Let us now give 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 this with the following pseudocode.

$$
MWIS\text{-intervals}(i) = \begin{cases} 
0 & \text{if } i > n, \\
\max \left\{ \begin{array}{l}
MWIS\text{-intervals}(i + 1), \\
w(I_i) + MWIS\text{-intervals}(j(i))
\end{array} \right\} & \text{otherwise},
\end{cases}
$$

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. 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 \( \text{MWIS-intervals}(1) \). 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!

### 5.1.3 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 \( S \subseteq V \).

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

\[
\text{MWIS}(v) = \text{the maximum weight of any independent set in the subtree rooted at } v.
\]

Let us now sketch out an implementation \( \text{MWIS}(v) \). 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.

\[
\text{MWIS}(v)
\]

1. Return the maximum of

   /* Don’t include \( v \) in the independent set */
   A. The sum of \( \text{MWIS}(x) \) over all children \( x \) of \( v \)

   /* Include \( v \) in the independent set */
   B. The weight of \( v \) plus the sum of \( \text{MWIS}(x) \) 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 \( x \). \( \text{MWIS}(x) \) simply doesn’t offer this. The easiest way to address this deficiency is to strengthen the definition \( \text{MWIS}(x) \). We introduce a boolean flag, called \( \text{exclude-root} \in \{0,1\} \), as an additional parameter, and define \( \text{MWIS}(v, \text{exclude-root}) \) as follows.

\[
\text{MWIS}(v, \text{exclude-root}): \text{If } \text{exclude-root} = \text{true}, \text{then the maximum weight independent set in the subtree rooted at } v \text{ among those excluding } v. \text{ If } \text{exclude-root} = \text{false}, \text{then the maximum weight independent set in the subtree rooted at } v \text{ (which may or may not include } v)\]

This stronger definition makes the implementation effortless.
5. Optimization in graphs

5.1. Independent set

Kent Quanrud
Spring 2022

MWIS\((v, \text{exclude-root})\)

1. If exclude-root

   /* Don’t include \(v\) in the independent set */
   A. Return the sum of MWIS\((x, \text{false})\) 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\((x, \text{false})\) over all children \(x\) of \(v\).

   /* Include \(v\) in the independent set */
   B. The weight of \(v\) plus the sum of MWIS\((x, \text{true})\) 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\((r, \text{false})\). But our algorithm is not yet efficient. To this end – as usual – we apply caching. Let us save the answers to each MWIS\((v, \text{true})\) and MWIS\((v, \text{false})\) in a table\(^1\) of size \(2n\). Then we never compute the same answer twice.

Consider the running time. We have

\[ (# \text{subproblems}) \times (\text{time per subproblem}) = n \times O(n) = O\left(n^2\right). \]

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

\[ (\text{time spent on } \text{MWIS}(v, \cdots)) = O(\text{# children of } v). \]

Summing over all \(v\), we have

\[ \sum_v (\text{time spent on } \text{MWIS}(v, \cdots)) = O\left(\sum_v \text{# children of } v\right) = O(n). \]

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!\(^1\)

\(^1\)We assume the vertices are number 1 through \(n\). Otherwise one can number them in a preprocessing step.
5.2 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 \( \pi : V \rightarrow [k] \), 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 \( \pi : V \rightarrow [k] \) for \( k \) as small as possible.

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 5.2** (Brook's theorem, [Bro41]). Let \( G \) be an undirected graph with maximum degree \( \Delta \). Then \( G \) can be colored in \( \Delta \) 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 \( \Delta \) can be edge-colored with at most \( \Delta \) 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

\(^2\)Two edges are adjacent if they have a common endpoint.
an exponential time algorithm by brute force. Can one do better? What about the special case of \( k = 3 \) – 3-coloring?

**Theorem 5.3.** 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 the graph \( \pi : V \to [k] \) such that for each \( k \), \( \pi(v) \in S_v \).

![Diagram of 3-coloring subset problem]

**Lemma 5.4.** 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 \( y_1, \ldots, y_k \) 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 \( i \)th 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 \( k = 3 \).

![Diagram of reduction to k-coloring]

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 \( \pi \) of \( G \), we can extend \( \pi \) to a \( k \)-coloring on \( G \) 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 \( i \). 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 5.3, it now suffices to show that subset 3-coloring can be used to solve SAT.

**Theorem 5.5.** A polynomial time algorithm subset 3-coloring implies a polynomial time algorithm for 3-SAT.

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 \( \{t, f\} \). 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 5.6.** Consider the following instance of subset 3-coloring with five vertices.

\[ \text{In any proper subset coloring, } c = \text{true only if } a \lor b = \text{true.} \]

**Proof.** By inspection. 

Now we can construct a gadget for 3-CNF clauses. Recall that \( (a \lor b \lor c) = ((a \lor b) \lor c) \). Likewise we can compose to OR gadgets as follows.

**Lemma 5.7.** Consider the following instance of subset 3-coloring with 8 vertices.

---

\(^3\)This is because we only care if two colors have the same labels, which is invariant to permutation of the labels.
Then in any proper subset coloring of the above, \(a \lor b \lor c = \text{true}\).

For each clause \(C_i\), we identify the three vertices corresponding to the literals in that clause. (For example, \(x_1 \lor \overline{x}_2 \lor x_3\) corresponds to the vertices \(A_1, \overline{A}_2, \text{and } A_3\).) We build the 3CNF gadget, per lemma 5.7 over these three vertices. Each clause generates 6 new variables.

Between the two vertices \(A_j\) and \(\overline{A}_j\) 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 \(A_j = \text{true}\) and \(\overline{A}_j = \text{false}\) if \(x_j = \text{true}\), and reverse the colors if \(x_j = \text{false}\). 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 5.1.

**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 5.3.

### 5.3 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 5.8. 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 $x_i = \text{true}$; $\bar{v}_i$ is associated with the symbol $\bar{x_i}$ and the assignment $x_i = \text{false}$.

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 $x_i = \text{true}$, then we include the corresponding vertex $v_i$. If the satisfying assignment has $x_i = \text{false}$, 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 $\{u_i, v_i\}$.

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 ($x_i = \text{true}$ or $x_i = \text{false}$) 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 5.4.

**Lecture materials and comments.** Click on the links below for the following files:

- Handwritten notes prepared before the presentation.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.
5. Optimization in graphs

5.4 Exercises

Exercise 5.1. Theorem 5.3 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 5.2. 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 5.3. 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.
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 5.4. Recall the dominating set problem from section 5.3. 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 \( \mathcal{J} \). The non-nested assumptions means that for any two intervals \( I, J \in \mathcal{J} \), we never have \( I \) contained in \( J \) or \( J \) contained in \( I \).

The goal is to compute the minimum weight subset \( S \subseteq \mathcal{J} \) of intervals such that every interval in \( \mathcal{J} \) is either in \( S \) or overlaps some interval in \( S \).

\(^4\)It might be helpful to focus on \( k = 4 \) to develop your ideas.
• (For 1 pt. extra credit) Extend your algorithm to general intervals.\(^5\)

2. The minimum weight dominating set problem in trees.

**Exercise 5.5.** Given a graph \(G\), a **clique** is a set of vertices \(S\) such that every pair of vertices in \(S\) is connected by an edge. This is sort of like opposite of an independent set.

1. Consider the problem of finding the largest weight clique from a set of weighted intervals. (Here a clique of intervals is a set of intervals where every two intervals are overlapping.) 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 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 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.\(^6\)

**Exercise 5.6.** 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 \(S\).\(^7\) 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 5.7.** 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.\(^8\)

----

\(^5\)Of course, anyone who has already solved the general case automatically solves the special case where no two intervals are nested.

\(^6\)Hint: Consider the “complement graph” \(G' = (V,E')\) where there is an edge \(\{u,v\}\) iff \(\{u,v\}\) is not an edge in \(G\).

\(^7\)Here a vertex \(v\) is a **neighbor** of \(S\) if it is connected by an edge.
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 algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 5.8. 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 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 $p_{i_1}, \ldots, p_{i_h}$ 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 5.9. 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 $\{1, \ldots, k\}$) 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 $i = 1, \ldots, k$, 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).* We consider the problem both for general graphs and for 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.

[ER19, Chapter 3] provides several more exercises for dynamic programming on trees.

*Technically this would be the “infimum” of all assignment costs of (valid) vertex $k$-colorings.
**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
   
   $\text{MWIS}(v) =$ the weight of the maximum weight independent set in the subtree rooted at $v$.

   $\text{MWIS-no-root}(v) =$ the weight of the maximum weight independent set in the subtree rooted at $v$ among those that exclude $v$.

2. **Recursive implementation.**

   $\text{MWIS}(v)$

   1. Return the maximum of:
      
      /* The maximum weight independent set containing $v$. */
      
      A. $\text{w}(v) + \sum \text{children } w \text{ of } v \text{ MWIS-no-root}(v)$
      
      /* The maximum weight independent set excluding $v$. */
      
      B. $\sum \text{children } w \text{ of } v \text{ MWIS}(v)$

   $\text{MWIS-no-root}(v)$

   1. return $\sum \text{children } w \text{ of } v \text{ MWIS}(w)$

3. **Solving the original problem.** We return $\text{MWIS}(r)$ 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. **Space.** $O(n)$ as this is the number of subproblems.
7. Brief proof / justification. Fix $v$ and consider $\text{MWIS}(v)$. 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 set excluding the root of that subtree. By induction on subtrees, for each subtree rooted by a child $w$, $\text{MWIS-no-root}(w)$ 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 $w$, $\text{MWIS}(w)$ 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 $\text{MWIS-no-root}(v)$, 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, $\text{MWIS}(w)$ returns the weight of the MWIS in this subtree. So the sum we return gives the MWIS overall.

Remark 5.9. In the lecture notes we added a flag to $\text{MWIS}$; 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 $i$th interval in increasing order of left end-point $a_i$.

1. Recursive spec. For $i = 1, \ldots, n + 1$, we declare

   $\text{MWIS}(i) = \text{the maximum weight independent set of intervals } I_i, \ldots, I_n.$

   ($i > n$ implies the empty collection of intervals.)

2. Recursive implementation.
\[ \text{MWIS}(i) \]

1. If \( i > n \) then return 0.

2. Return the maximum of

   \[
   \begin{align*}
   /* & \text{Include } I_i. \qquad */ \\
   & A. \quad w(I_i) + \text{MWIS}(j + 1) \text{ where } j \text{ is the last index of any interval } I_j \text{ overlapping } I_i. \\
   /* & \text{Don't include } I_i. \qquad */ \\
   & B. \quad \text{MWIS}(i + 1)
   \end{align*}
   \]

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 \( \text{MWIS}(1) \).

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. Space. We need \( O(n) \) space to store the solutions to the \( n \) subproblems (and the sorted list of intervals).

7. Proof by induction. Consider \( \text{MWIS}(i) \). 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 \( I_1, \ldots, I_n \). 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 \( (I_{j+1}, \ldots, I_{j-1}) \) also overlap \( I_j \). 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, \( \text{MWIS}(j + 1) \) returns the weight of the MWIS of \( I_{j+1}, \ldots, I_n \). So \( w(I_i) + \text{MWIS}(j + 1) \) 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 \( I_{j+1}, \ldots, I_n \). By induction on the number of intervals, \( \text{MWIS}(i + 1) \) returns this weight, hence so does \( \text{MWIS}(i) \).
Chapter 6

Searching and sorting graphs

The previous discussion was about some optimization problems in undirected graphs; here we focus on directed graphs. The difference here is that the edges are directed, 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 \(v\). A directed graph is allowed to have edges in both directions \((u, v)\) and \((v, u)\) between two pairs.

Seemingly anything can be modeled as a graph, but let us 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 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 (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 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

---

\(^1\)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, this is like randomly reindexing the vertices from 1 to \( O(n) \).
6. Searching and sorting graphs

6.1. Directed acyclic graphs and topological sort

Kent Quanrud
Spring 2022

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\(^2\)

We start with an important special case of directed graphs where there are no directed cycles. Such a graph is called a directed acyclic graph, abbreviated DAG.

![Directed Acyclic Graph Diagram](image)

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 \( v_3, v_4, v_5, \ldots \) where for each \( i \), \( v_i \) is a source in the graph \( G_{i-1} \) obtained by removing \( v_1, \ldots, v_{i-1} \) from \( G \). The following subroutine encodes this process.

\[
\text{source-peeling}(G = (V,E))
\]

\[
/* \ We \ assume \ G \ is \ a \ DAG. */
\]

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.

---

\(^2\)This section overlaps with section 2.3, 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.
Consider the vertices \( v_1, \ldots, v_n \) 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 \( v_i \). As this holds for all \( i < j \), we see that all edges go from \( v_i \) to \( v_j \).

The list output by the source-peeling algorithm is an example of a **topological order**. In general, an ordering \( v_1, \ldots, v_n \) of \( V \) is a **topological order** iff every directed edge in \( G \) goes from \( v_i \) to \( v_j \) for \( i < j \).

Topological orderings are extremely useful algorithmically. They give us a sequential structure of the graph on which we can apply the algorithmic techniques of chapter 4 (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 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 \( G = (V, E) \) with \( m \) edges and \( n \) vertices. A **walk** in a graph is a sequence of vertices along edges; i.e., a sequence \( v_0, \ldots, v_k \in V \) such that each \((v_i, v_{i+1})\) is a directed edge. A **path** is a walk that doesn’t repeat vertices. We say that a vertex \( s \) can **reach** a vertex \( t \) if

\(^3\)The (unweighted) in-degree of a vertex \( v \) is the number of edges directed into \( v \).
there is a path from \( s \) to \( t \) in the graph. To test these definitions, consider the following directed graph. Can \( s \) reach all of the vertices in this graph?

Identifying the set of reachable vertices 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 from 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.

\[
\text{recursive-search}(s,t) \\
1. \text{If } s = t \text{ then return } \text{true} \\
2. \text{For each edge } (s,u) \in E \\
   A. \text{If recursive-search}(u,t) \text{ returns true then return true} \\
3. \text{Return false}
\]

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 \text{true}. Otherwise, we return \text{false}.

There is a catch, however; more precisely, an \textit{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 the line, or on trees, where there are clear notions of “left and right”, or “top and bottom”, and we naturally steered clear of cycles.

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 - meaning, we already visited \(v\) - then we stop exploring \(v\) and return (to wherever we left off before \(v\)). Otherwise we continue to explore \(v\)’s edges. The following algorithm, called depth-first-search (and abbreviated DFS), augments recursive-search with marking.

\[
\text{DFS}(v)
\]

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 $\text{DFS}(v)$ returns, all vertices reachable from $v$ have been marked.

**Proof.** Consider the moment that $\text{DFS}(v)$ returns. Let

$$A = \{\text{marked vertices}\} \quad \text{and} \quad B = \{\text{unmarked vertices}\}.$$ 

Let $w$ be reachable from $v$, and suppose by contradiction that $w \in B$. Since $w$ is reachable from $v$, there is a path $p$ 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.** $\text{DFS}(v)$ 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 $\text{DFS}$.

**Theorem 6.6.** In $O(m + n)$ time, $\text{DFS}(v)$ 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 $\text{DFS}(v)$ 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 $\text{DFS}(v)$. To make our discussion more concrete, let us modify DFS to print out vertices as we finish searching them.

\[
\text{DFS}(v)
\]

1. If $v$ is unmarked:

   A. Mark $v$.
   B. For each edge $(v, w)$ from $v$:
      1. DFS($w$).
      \hfill /* The following line is the only difference from the first version of DFS. */
   C. Append $v$ to the output

In case one call to $\text{DFS}(v)$ 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 $v$.

\[
\text{DFS-driver}(G = (V, E))
\]

1. For each vertex $v \in V$
   A. DFS($v$)

Ultimately $\text{DFS-driver}$ 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 $\text{DFS-driver}$ a **post-DFS ordering** of $V$. (“post” because it comes at the end of DFS.) To understand this ordering, we first recall the basic invariant of DFS: $\text{DFS}(v)$ 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 $\text{DFS}(v)$ 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 (which again applies to all graphs).

**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 marks $v$ 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 $s$ and 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 (that is, 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 $V$. In the picture below, 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 $v$, which tells us for every
6. Searching and sorting graphs

6.4. Strongly-connected components

Figure 6.2: The strongly connected components of a graph.

vertex who they can reach. We can fill out a two-dimensional table with these 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 $\text{DFS}(v)$ 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 $\text{DFS}(u)$ from every vertex $u$ visited by $\text{DFS}(v)$, and take the subset of vertices $u$ that could reach $v$. This will work, but the running time is $O(n(m+n))$ which is not very impressive.

2. Run $\text{DFS}(u)$ only from the leaves $u$ of the search tree carved out by $\text{DFS}(v)$. 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 $\text{DFS}(v)$ 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 $\text{DFS}(v)$ routine to try to detect back-edges 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 $G_m$ in $O(m + n)$. 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.

Taking a step back, consider the graph $G_C$ obtained by contracting each strongly connected component to a single vertex.

![Diagram of contracted graph with no cycles]

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 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.
6. Searching and sorting graphs

6.4. Strongly-connected components

Kent Quanrud
Spring 2022

sink-first-SCC(G = (V, E))

1. until V = ∅

   /* 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 ← 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:

\[
\text{If } u \text{ can reach } v \text{ but } v \text{ cannot reach } u, \text{ then } v \text{ is listed before } u. \quad (\star)
\]

This property does not tell us about the relative order 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 (★) above, the last vertex is a vertex from a source component.

Proof. Let \( v_1, \ldots, v_n \) satisfy (★); 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 (★).

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 (★). 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.

\[
sink\text{-}first\text{-}SCC(G = (V,E))
\]

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, \( sink\text{-}first\text{-}SCC(G) \) 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], [CLRS09, Chapter 22], and [DD11, (video) lecture 14].

**Lecture materials and comments.** Click on the links below for the following files:

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

## 6.6 Exercises

Many more exercises related to the discussions in 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.

\[
\text{edge-search}(v)
\]

1. for each edge \( e = (v \rightarrow w) \) 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 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

\begin{align*}
\end{align*}
have to cycle through the American dream thereafter. We say that a vertex \( s \) can **patriotically reach** a vertex \( t \) if there is an American walk from \( s \) to \( t \).

Consider the problem of deciding whether a vertex \( s \) can patriotically reach \( t \). 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.\(^4\)

**Exercise 6.5.** Let \( G = (V, E) \) be 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: (red, white, blue), (white, blue, red), or (blue, red, white). We say that a vertex \( s \) can **un-patriotically reach** a vertex \( t \) if we can walk from \( s \) to \( t \) via an un-American walk.

Consider the problem of deciding whether a vertex \( s \) can un-patriotically reach \( t \). 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 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. 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 6.7.** 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

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

\(^4\)Hint: don’t change the algorithm; change the graph!\)
faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 6.8.** 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. Partial credit will be awarded to an algorithm that works in nearly linear time on DAGs.\(^5\)

---

\(^5\)Previously we mistakenly wrote linear time (without the “nearly”); but the solution we had in mind involves logarithmic factors.

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

The longest path

Last chapter introduced directed graphs, and focused on questions concerning connectivity between vertices. Among the structural ideas that emerged were topological orderings in acyclic graphs, strongly connected components, and the relations between them. Algorithmically we investigated depth-first search and its applications, highlighted by a linear time algorithm to decompose the entire graph into its strongly connected components.

We continue to focus on directed graphs, and 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 7.1.) An important special case of the decision question is where $k = n$. That is, does the graph contain a path traversing all the vertices? Such a path is called a Hamiltonian path; the “Hamiltonian path problem” is to decide 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. After that we will consider general graphs.

7.1 The longest path in a DAG

Let $G = (V, E)$ be a directed acyclic graph (DAG). Recall that DAG’s were first introduced when discussing circuit-SAT (section 2.3), and played an important role in our discussion on DFS and strongly connected components (chapter 6). 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 the repetition implies a cycle. Our problem, then, is the same as finding the longest walk in $G$; the constraint of avoiding repetition is irrelevant.

Let us develop a simple recursive algorithm. We want the length of the longest path/walk in $G$. (The actual longest path itself will follow easily) Where to start the path? Who knows – so let’s throw the starting vertex in the recursive specification as a parameter. Thus let us define:

$$\text{longest-walk}(v) = \text{the length of the longest walk starting from } v \text{ (as measured by the number of edges).}$$

Given $\text{longest-walk}(v)$ as defined above, we will return the maximum $\text{longest-walk}(v)$ over all $v \in V$.

Now, to implement longest-walk as specified, we have the following. (Recall that a sink vertex is a vertex with no outgoing edges.)

$$\text{longest-walk}(v) = \begin{cases} 0 & \text{if } v \text{ is a sink} \\ 1 + \max_{(v,w) \in \delta^+(v)} \text{longest-walk}(v) & \text{otherwise.} \end{cases}$$

As with any recursive algorithm, we should first justify that this algorithm terminates in finite time – more precisely, that there are no potential cyclic dependencies in the recursive calls. 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. Consequently the recursive calls from $\text{longest-walk}(v)$ only go down the topological order – and never back up create an infinite loop.

Now, directly running the recursive algorithm above has a worst case running time of $O(n!)$ (see exercise 7.2). But suppose we cache our answers – that is, we apply dynamic programming – and visit each vertex only once. Excluding recursive subcalls, the time to compute $\text{longest-walk}(v)$ is proportional to the out-degree of $v$ (denoted $\deg^+(v)$). Summing $\deg^+(v)$ over all $v$ counts every edge exactly once, so the total running time is

$$O\left( \sum_{v \in V} 1 + \deg^+(v) \right) = O(m + n).$$

So the longest path problem can be solved in linear time in a DAG.

Of course this only gives the maximum length of any path. We can extract the actual path as follows. For each $v$, we record the vertex $w$ following $v$ on the
longest walk from \( v \). We can then recover the longest walk from \( v \) by following these starting from \( v \). (This is essentially the same idea, used generically in other dynamic programming situations, where the optimal choice for each subproblem is stored in a secondary table, in order to reconstruct the object attaining the optimum value afterwards.)

Stepping back, this problem highlights a vital algorithmic feature of DAG’s, which is that we can apply the ideas of dynamic programming to DAG’s, using the topological order to guide the induction.

### 7.2 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 7.1.** 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 7.1.

We take as input a SAT formula \( f(x_1, \ldots, x_n) \) 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 \( x_j = \text{true} \) or \( x_j = \text{false} \)) 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 \( x_j = \text{true} \) and the other as setting \( x_j = \text{false} \).
We arrange these paths, one for each $x_i$, in a “ladder”-like structure, strung together by edges connecting the endpoints. In particular we will 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.

We have not yet encoded any clauses. To this end we have the following gadget. Let us (arbitrarily) fix, for each vertex $x_j$, the left to right direction as denoting $x_j = \text{true}$, and the right to left direction as denoting $x_j = \text{false}$. We want to design a gadget for each clause that is “free” when we travel along the direction of a vertex that corresponds to satisfying the clause. To be specific, 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 $x_j = \text{true}$), 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. Moving in the right to left direction (corresponding to $x_j = \text{false}$), we do something similar except in the opposite direction. We pick two consecutive unused nodes along $x_j$’s path. Moving in the right to left direction, 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 $m + 1$ additional vertices $d_{j,i}$ 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 through 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 $j = 1, \ldots, n$, we add all 4 combinations of directed edges from $s_j$ and $t_j$ to $s_{j+1}$ and $t_{j+1}$. For $j = n$, we add 4 directed edges from $v_{n,0}$ and $v_{n,m}$ to $v_{1,0}$ and $v_{1,m}$. 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 7.1.
Lecture materials and comments. Click on the links below for the following files:

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

7.3 Exercises

Exercise 7.1. Show how to use a polynomial time algorithm for the decision version of the longest path problem (“Is there a path of length \( k \)?”) gives a polynomial time algorithm for the search version (“Find the longest path in the graph.”).

Exercise 7.2. 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 7.3. A Hamiltonian cycle is a cycle that visits every vertex exactly once. 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.

Exercise 7.4. 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 \( O(n! \text{ poly}(n)) \) time. Can one do better? Design and analyze an algorithm that computes a Hamiltonian path in \( O(2^n \text{ poly}(n)) \) time. (The smaller the \( \text{poly}(n) \) term the better, but the key point is to reduce \( n! \) to \( 2^n \).)
Exercise 7.5. The following two 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. For both 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. Given a directed graph $G$, compute a walk that visits the greatest number of distinct vertices.

2. Given a directed graph $G$, compute a walk that traverses the greatest number of distinct edges.

Exercise 7.6. Let $G = (V, E)$ be a directed graph with vertex weights $w : V \rightarrow \mathbb{R}$. 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 7.7. Let $G = (V, E)$ be a directed graph with $m$ edges and $n$ vertices, where each vertex $\nu \in V$ is given an integer label $\ell(\nu) \in \mathbb{N}$. The goal is to find the length of the longest path\footnote{Recall that a path is a walk that does not repeat vertices.} in $G$ for which the labels of the vertices are strictly increasing.

1. (7 points) 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. (3 points) 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.

Exercise 7.8. Let $G = (V, E)$ be a directed graph where each edge is colored red, white, or blue. An American path is a path where the edge colors alternate red, white, blue. The American Hamiltonian path problem is to decide if there is an
American path that visits all the 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 7.9.** Consider a directed graph $G = (V, E)$ that is transitive 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 7.10.** 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 $r : E \rightarrow \mathbb{R}^>0$. 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. You may assume for simplicity that $G$ has no sinks.

1. Given an integer $k \leq n$, the goal is to compute a walk of length $k$ maximizing the sum of rewards along that walk.\(^2\) (If you repeat an edge, you get the same reward each time you repeat the edge.) 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. 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 $k$ maximizing the discounted total reward of the walk. 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.

\(^2\)There is always a $k$-edge walk under the assumption that $G$ has no sinks.
Exercise 7.11. 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 \( G \).\(^3\) 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 7.12. 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 \((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\) 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 \( \text{Length}(x, y, x', y') \) 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.

\(^3\)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.
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. We first observe that all walks are paths in a DAG, so it is equivalent to compute the length of the longest walk.

1. **Recursive spec.** For a vertex $v \in V$, we define
   \[ \text{LW}(v) = \text{the length of the longest walk in } G. \]

2. **Recursive implementation.**
   \[ \text{LW}(v) \]
   1. If $v$ is a sink then return $0$.
   2. Otherwise return the maximum of $1 + \text{LW}(w)$ over all edges $(v, w)$ leaving $v$.

3. **Solving the original problem.** We return the maximum of $\text{LW}(v)$ over all $v \in V$.

4. Say “caching” or “dynamic programming”. We cache the answers to the recursive algorithm above.

5. **Running time.** Each $\text{LW}(v)$ takes time proportional to the out-degree of $v$ (plus a constant). Summing over all $v$ counts each edge once, so we obtain a $O(m + n)$ running time overall.

6. **Space.** $O(n)$ as this is the number of subproblems.

7. **Brief proof / justification.** Fix $v$ and consider $\text{LW}(v)$. If $v$ is a sink then any walk starting at $w$ is just $w$, hence as length $0$, as returned by $\text{LW}(v)$.

   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 $V$, $\text{LW}(w)$ returns the length of the longest walk on $v$. So $1 + \text{LW}(w)$ is the length of the longest walk from $v$, and this value is returned in $\text{LW}(v)$. 

Sample solution.
Chapter 8

Shortest walks

We have previously discussed algorithms for reachability: whether or not there exists a path between two vertices. For reachability, a beautiful structure emerges from the strongly connected components and the DAG of contracted SCC’s, making the problem and many variations of it very tractable. We have also considered the problem of finding the longest path between vertices; this problem turns out to be very hard. 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.

8.1 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 \( \bar{\ell}(w) \).

\[
\bar{\ell}(w) \overset{\text{def}}{=} \sum_{e \in w} \ell(e)
\]

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 over the lengths of all walks from \( s \) to \( t \):

\[
d(s,t) = \inf \{ \bar{\ell}(w) : w \text{ is a walk from } s \text{ to } t \}.\]

By definition, this quantity is \( \infty \) 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. In this chapter we will focus on the unweighted and positive settings and postpone negative edge weights for 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 \( t \). Let us first point out 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.

This problem seems similar to the longest path problem previously discussed. We will approach the shortest path problem the same way. First we will consider DAG’s, which are simplest. Then we will consider general graphs.

### 8.2 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

\[
distance(v) \text{ as the length of the shortest path from } s \text{ to } v.
\]

\(^1\)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 \( \bar{\ell}(w) \leq L \).
We want to compute $\text{distance}(t)$. 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 $\text{distance}(t) = \text{distance}(v) + \ell(v, t)$. But which in-neighbor $v$? Who knows. Let us recursively compute $\text{distance}(v)$ over all incoming edges $(v, t)$ and chose the one minimizing $\text{distance}(v) + \ell(v, t)$. All put together we have the following recursive algorithm. (Note that we avoid infinite loops because $G$ is a DAG.)

$$
\text{distance}(t) = \begin{cases} 
0 & \text{if } t = s, \\
+\infty & \text{if } t \text{ has no incoming edges,} \\
\min_{(v, t) \in \partial^{-}(t)} \text{distance}(v) + \ell(v, t) & \text{otherwise.}
\end{cases}
$$

As usual with DAG’s, the initial recursive algorithm may not be fast, but we can make it efficient by saving all our answers. With dynamic programming, the running time becomes the sum, over all $t \in V$, of the number of incoming edges to $t$ (plus a constant). This gives a $O(m + n)$ running time in total.

If the algorithm seems similar to the longest path algorithm from section 7.1, that’s because they are essentially the same. One can extend the longest-walk algorithm for DAGs to real-valued edge lengths and moreover one can observe that the edge lengths do not have to be positive. Assuming this extension, then to find the shortest paths in a weighted DAG $G$, we could have negated all the edge lengths and found the longest path with respect to the negated edge lengths instead.

### 8.3 Unweighted shortest paths

We now consider shortest paths in general directed graphs. To ease into it we first focus on unweighted graphs: the length of a path is the number of edges.

When working with a DAG above, we could rely on the topological ordering to decide in which order of vertices to calculate the distances. General graphs do not have a topological order and we do not know \textit{a priori} in which order to calculate the vertex distances. However, we do know:

0. \textit{The vertices at distance 0}: This is just $\{s\}$.

1. \textit{The vertices at distance 1}: These are just the vertices $v$ with an arc $(s, v)$ from $s$.

2. \textit{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) don’t have an arc $(s, v)$. 
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) don't have any arcs \((x,v)\) from a vertex \( x \) at distance 1 or less.

Clearly a pattern has started to emerge. In general, for any integer \( k \),

\[ k \text{.  The vertices at distance } k: \text{ the vertices } v \text{ that (a) have an arc } (u,v) \text{ from some vertex } u \text{ at distance } k - 1, \text{ and (b) do not have any arcs } (x,v) \text{ from a vertex } x \text{ at distance } k - 2 \text{ or less.} \]

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 \( \text{distance}(s) = 0 \).

2. For \( k = 0,1,2,\ldots \), as long as there is at least one vertex \( u \) with \( \text{distance}(u) = k \):

   A. For all vertices \( u \) such that \( \text{distance}(u) = k \):
      
      1. For all arcs \((u,v)\) for which \( \text{distance}(v) \) is not yet set:
         a. set \( d(v) = k + 1 \).

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 \( u \) 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.

We remark that the algorithm above is implicitly very similar to our algorithm for DAG's. We are incrementally building out distances in increasing order from \( s \). To make the algorithm clearer, 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 as 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.
8. Shortest paths

8.4 Dijkstra’s algorithm for weighted shortest paths

\[ \text{BFS}(G = (V, E), s \in V) \]

// 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.

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 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.

8.4 Weighted shortest paths

We now move onto the weighted case. Let \( G \) have positive edge lengths \( \ell : E \rightarrow \mathbb{R}_{>0} \). 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 don 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 endpoint \( v_2 \) 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 opposite endpoint of the second smallest edge \( (s, x) \) leaving \( s \), but it does not have to be. It is possible that for an edge \((v_2, y)\) leaving \( v_2 \), the combined distance \( d(s, v_2) + \ell(v_2, y) \) is less than the length of any edge \((s, x)\) leaving \( x \). We will also consider this possibility and take the minimum of either scenario, as follows.
The third closest vertex \( v_3 \) is either (a) the endpoint \( x \) of the edge \( e = (s, x) \) of minimum length \( \ell(e) \) or (b) the endpoint \( x \) 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 \( d(s, v_2) + \ell(s, x) \).

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 \( s \):

1. To find the \( k \)th closest vertex \( v_k \) from \( s \), we consider all the closer vertices \( v_i \) (where \( i \in \{1, \ldots, k-1\} \)) and all edges \((v_i, x)\) where \( x \) is not yet labeled, and choose \( x \) that minimizes \( d(s, v_i) + \ell(v_k) \).

Similarly to unweighted distances, an inductive structure has emerged. \textit{A priori} it is not clear which vertex is the \( k \)th 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. (More concrete implementation details will be supplied momentarily). The algorithm is named after Edgar Dijkstra who published it in 1959.

\[
\text{Dijkstra}(G = (V, E), \ \ell : E \to \mathbb{R}_{\geq 0}, \ s \in V)
\]

\[
/* \text{ We assume for simplicity that all vertices are reachable from } s; \text{ other vertices would be ignored. } */
\]

1. Set \( \text{distance}(s) = 0 \).
2. Until all vertices (reachable from \( s \)) are assigned a distance:
   
   A. Let \( u, v \in V \) minimize \( \text{distance}(u) + \ell(u, v) \) over all vertices \( u \) that have been assigned a distance, and all edges \((u, v)\) leaving \( u \), restricting to endpoints \( v \) that have not yet been assigned a distance.
   
   B. Set \( \text{distance}(v) = \text{distance}(u) + \ell(u, v) \).
3. Return all the distance labels.

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
takens $O(m)$ time. So we have a $O(mn)$ time algorithm for computing distances from $s$.

So 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 $\text{distance}(u) + \ell(u,v)$ 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 $p$.

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. 8.1.
8. Shortest paths

8.4. Dijkstra’s algorithm for weighted shortest paths

Dijkstra\( (G = (V, E), \ell : E \to \mathbb{R}_{\geq 0}, s \in V)\)

\/* 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 priority queue / heap over \( V \) with priority decreasing in tentative\( (v) \).

3. While \( Q \) is not empty
   A. Remove the vertex \( v \) of minimum tentative\( (v) \) from \( Q \).
   B. Set distance\( (v) = \text{tentative}(v) \).
   C. For each outgoing edge \( (v, w) \):
      1. Set
         \[
         \text{tentative}(w) = \min\{\text{tentative}(w), \text{distance}(v) + \ell(v, w)\}
         \]
         and decrease \( w \)'s priority in \( Q \) accordingly.

4. Return the distance labels.

Figure 8.1: Accelerating Dijkstra’s algorithm with a priority queue.

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

\[
O(\log n) \times (# \text{decrease-key's}) + O(\log n) \times (# \text{remove-min's})
\]

\[
= O(m \log n) + O(n \log n) = O(m \log n).
\]

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 8.1.** 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.

132
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 8.1 as given.

If we use Fibonacci heaps, then the running time becomes

\[
O(1) \times (\# \text{ decrease-key's}) + O(\log n) \times (\# \text{ remove-min's})
= O(m + n \log n).
\]

In conclusion, we have the following remarkable running time for computing shortest paths with positive edge weights.

**Theorem 8.2.** Let \( G = (V, E) \) be a directed graph with positive edge weights \( \ell : E \rightarrow \mathbb{R}_{>0} \) 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.

**Lecture materials and comments.** Click on the links below for the following files:

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

### 8.5 Exercises

**Exercise 8.1.** Let \( G = (V, E) \) be an unweighted, directed graph, and 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. 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.
8. Shortest paths

8.5. Exercises

Kent Quanrud
Spring 2022

Exercise 8.2. 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 \(\ell : E \rightarrow \mathbb{R}_{>0}\), 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 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 8.3. The racetrack problem in Chapter 5 of [Eri19].

Exercise 8.4. 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. 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 8.5. Let \(G = (V, E)\) be a directed and unweighted graph, and \(s, t \in V\). Consider the problem of computing the number of shortest \((s, t)\)-paths.\(^3\) 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.

\(^3\)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 8.6.** Let $G = (V,E)$ be a directed graph with $m$ edges, $n$ vertices, positive edge lengths $\ell : E \to \mathbb{R}_{>0}$. Given $s,t \in V$, the goal is to compute the length of the shortest walk from $s$ to $t$ with an even number of edges (which may be $+\infty$ if there is no way to get from $s$ to $t$ with an even number of 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.
Chapter 9

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 DFS 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 below.

\[ a \quad 1 \quad 3 \quad b \quad 1 \quad d \quad 2 \quad c \]

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.

Let us state the problem formally. Let $G = (V,E)$ be a directed graph with real-valued (and possibly negative) edge lengths $\ell: E \rightarrow \mathbb{R}$. Fix $s,t \in V$. What is the length of the shortest walk from $s$ to $t$? We note that the length can possibly 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). Below the negative cycle between $s$ and $t$ makes the distance $-\infty$. 

136
In passing, let us test the reader with another closely question. Assuming the same input as before - a directed graph with real-valued edge weights and \( s, t \in V \). The goal is the compute the length of the shortest path from \( s \) to \( t \). Is there a polynomial time algorithm for this problem? How or why?

### 9.1 Negative edges, shortest walks, and shortest paths

Recall the definition of distances in a directed graph (page 126). 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 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 9.1.** 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 non negative, in which 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. ■

¹That is, there is an \((s,t)\)-path with total length less than or equal to any other \((s,t)\)-walk.
9.2 Shortest walks with negative edges

The problem is to find the shortest walk and there is no clear induction hypothesis. As noted above, negative edges means we can’t rely on distances to implicitly supply an induction argument, as we did in developing BFS and Dijkstra’s shortest path algorithms. Instead we will 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

\[
SW(s, t, k) = \text{the length of the shortest walk from } s \text{ to } t \text{ with at most } k \text{ edges (defined as } +\infty \text{ if there is none).}
\]

Here there is a clear induction argument on \( k \) – the \( k \) edge walks build on \( k - 1 \) edge walks by appending an extra edge.

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 \((v, t) \in \partial^-(t)\).

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 \( O(1 + \deg^-(t)) \) time. For fixed \( s \) and \( k \), it takes \( O(m + n) \) time to compute all of the subproblems for a fixed value of \( k \).

**Lemma 9.2.** 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 9.1, the \((s, t)\)-distance is attained by a path for every vertex \( t \).

\(^2\)Currently we are interested in the case where \( s \) is fixed, but later it will be convenient to have \( s \) as a parameter.
9. Negative edge lengths and all-pairs shortest paths

9.2. Bellman-Ford: distances with negative edges

**Corollary 9.3.** 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 $t \in V$.

**Stationary points and $-\infty$.** The remaining issue is $-\infty$. Intuitively, if $d(s, t) = -\infty$, then $SW(s, t, k)$ should decrease towards $-\infty$ as $k \to \infty$. That does not mean that $SW(s, t, k)$ decreases with every single $k$; perhaps we need many additional edges to go all the way around a negative cycle and actually decrease $SW(s, t, k)$. It’s also not clear how long to keep iterating on $SW(s, t, k)$ before we can conclude that there is no lower bound. So we have to analyze $SW(s, t, k)$ carefully.

For $k \in \mathbb{N}$, let

$$A_k = \{ v \in V \text{ where } SW(s, t, k) \neq SW(s, t, k - 1) \}$$

be the set of vertices $v$ where $SW(s, t, k)$ decreases compared to the previous iteration for $k - 1$. Recalling the definition of $SW(s, v, k)$, these are the vertices where the best $k$-edge walk from $s$ to $v$ is strictly shorter than the best $(k - 1)$-edge walk. Then for each $k$, we have

$$A_{k+1} \subseteq N^+(A_k),$$

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

$$A_{k+2} \subseteq N^+(N^+(A_k)), A_{k+3} \subseteq N^+(N^+(N^+(A_k))), A_{k+4} \subseteq N^+(N^+(N^+(N^+(A_k)))),$$

and so forth. In general, for any $\ell \geq k$, all the vertices in $A_{\ell}$ must be reachable from $A_k$. Conversely, for all vertices $t$ that are not reachable from $A_k$, we know that $SW(s, t, \ell)$ will never change. They are forever fixed point in this process. Consequently $d(s, t) = SW(s, t, k - 1)$ for any such $t$.

Now consider in particular $A_n$. 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 9.1, 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 $SW(s, v, \ell)$ would never change for $\ell \geq n$) we have $SW(s, v, n) = d(s, v)$. All together, we now have all the finite distances from $s$ as well as all the vertices with distance $-\infty$. Funny enough, we churn $SW(s, t, n - 1)$ for just one more iteration to get all the distances, finite or not.
Theorem 9.4. 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 vertex $s^*$ with an edge $(s^*, v)$ of length 0 for every $v \in V$. In particular $s^*$ can reach every vertex directly. If we apply theorem 9.4 from $s^*$ (that is, compute $SW(s^*, t, n + 1)$ 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 $-\infty$.

Corollary 9.5. Given a directed graph $G$ with real-valued edge weights, in $O(mn)$ time, one can identify all vertices reachable from a negative cycle.

9.3 All pairs shortest paths and the doubling trick

We now move on a related problem of computing the distances between all pairs of vertices, not just 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 9.5).

Recall our definition of $SW(s, t, k)$, which we repeat below:

$$SW(s, t, k) = \text{the length of the shortest walk from } s \text{ to } t \text{ with at most } k \text{ edges (defined as } +\infty \text{ if there is none).}$$

Before we had $s$ fixed and varied $t$; now we are interested in all choices of $s, t \in V$.

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 $SW(u, v, \ell)$ for any $u, v \in V$ and $\ell \leq k$. In particular, any

$^3$Don’t change the algorithm, change the graph.
(s, t)-path of at most 4 edges can be divided into two paths 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 powers of 2, consider the following alternative implementation for SW.

\[
\text{SW}(s, t, 2^i)
\]

1. If \(i = 0\), then return the minimum of:
   A. 0 when \(i = j\)
   B. \(\ell(s, t)\) when \((s, t) \in E\).
   C. +\infty.

2. Return the minimum of
   A. \(\text{SW}(s, t, 2^{i-1})\)
   B. \(\text{SW}(s, v, 2^{i-1}) + \text{SW}(v, t, 2^{i-1})\) for all \(v \in V\). // O(n) time.

The key difference between this implementation of SW(a, n, d) Bellman-Ford is in (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 9.6.** Let \(G = (V, E)\) be a directed graph with \(n\) vertices. In \(O(n^3i)\) time, one can compute \(\text{SW}(s, t, 2^j)\) for all \(s, t \in V\) and all \(j \leq i\).

If there is no negative cycles (as we have assumed) then \(\text{SW}(s, t, n - 1)\) (or any value greater than \(n - 1\)) gives the distance from \(s\) to \(t\). For \(k = \lceil \log n \rceil\), lemma 9.6 gives us the distances between all pairs of vertices.

**Corollary 9.7.** 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 \(n\).

### 9.4 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:

$$SP(i, j, k) = \text{the length of the shortest path from } v_i \text{ to } v_j \text{ using only } v_1, \ldots, v_k \text{ as intermediate vertices.}$$

We implement the above as follows.

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. $SP(i, j, k - 1)$
   - B. $SP(i, k, k - 1) + SP(k, j, k - 1)$

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 $SP(i, j, k)$ for all $i, j, k$.

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 $v_1$ 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 longest 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 algorithm is commonly called the **Floyd-Warshall** algorithm although there are other relevant publications from roughly the same time [Flo62; Ing62; Roy59; War62] (see the Wikipedia article).

**Theorem 9.8.** 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 to handle negative edges, called Johnson’s algorithm
[Job77], that we do not discuss.

9.5 The traveling salesman problem

Let \( G = (V, E) \) be a directed graph with positive edge weights\(^4\). A tour of \( G \) is
a walk, starting and ending at the same vertex\(^5\), that visits every vertex at least
once. The traveling salesman problem is to compute the minimum length tour.

The traveling salesman problem is extremely useful as a prototypical plan-
ning and routing problem (broadly construed). It has inspired a number of
techniques both in theory and in practice, and also captured the popular imagina-
tion. See [https://www.math.uwaterloo.ca/tsp/history/index.html](https://www.math.uwaterloo.ca/tsp/history/index.html) for more
on the history of this problem.

The problem turns out to be difficult, as it is a thinly disguised Hamiltonian
path problem. Indeed, let \( G = (V, E) \) be a directed graph in which we want to
find a Hamiltonian path. Let us add a new vertex \( s^* \) with edges \((s^*, v)\) and \((v, s^*)\) to
and from \( s^* \), for every \( v \in V \). Call this new graph \( G^* \).

Each Hamiltonian path map to a Hamiltonian cycle in \( G^* \) by connecting the
beginning and end through \( s^* \). The inverse mapping takes a Hamiltonian cycle
in \( G^* \) and produces a Hamiltonian path by splitting the cycle at \( s^* \).

Finally, we point out that a Hamiltonian cycle is a tour of \( G^* \) with \( n^* = n + 1 \)
edges. So a TSP solver can find a Hamiltonian cycle in \( G^* \), hence a Hamiltonian
path in \( G \).

**Theorem 9.9.** A polynomial time algorithm for the (directed) traveling salesman problem
implies a polynomial time algorithm for SAT.

9.6 Additional notes and references

The content in this chapter overlaps with [Eri19, chapter 9], [KTo6, sections
6.8–6.9], [DPVo8, chapter 4], [CLRS09, chapter 24], and [DD11, (video) lecture
19].

\(^4\)One could also consider negative edge weights; we restrict to positive edge weights for simplicity.
\(^5\)Such a walk is sometimes called a circuit, but should not be confused with logical circuits that
have been discussed elsewhere.
9.7 Exercises

Exercise 9.1. Recall that in foreign exchange, different currencies can be exchanged at 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 arbitrary 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.
Chapter 10

NP-Completeness

10.1 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 10.2 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 ∧ 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 ∨ 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.

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 $(x_1, x_2), (x_2, x_3), \ldots, (x_k, x_1)$ 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.\footnote{One could consider evaluation over time, but this is a separate discussion.}

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

---

Figure 10.1: A circuit computing the exclusive-or of two bits (at the top gate).

Figure 10.2: A circuit for adding two bits. Here, the gate at the top outputs true iff $x_1 + x_2 = x_3 x_4$. 
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 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 2.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 “\( z = x \land y \)” , “\( z = x \lor y \)” 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 “\( \land z \)” to the whole formula, where \( z \) represents the output gate, to require that the whole circuit is satisfied.

**Theorem 10.1.** There is a polynomial time algorithm for SAT iff there is a polynomial time algorithm for circuit SAT.
10.2 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.

![Diagram of a circuit over time]

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

$$O((\# \text{ time steps})^2).$$

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.

### 10.3 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 $f : \{0,1\}^n \rightarrow \{0,1\}$ 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 $f(x) = \text{true}$.

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 10.2.** A decision problem said to be in **nondeterministic polynomial time** (abbrev. **NP**) if there is a polynomial time algorithm $f : \{0,1\}^* \rightarrow \{0,1\}$, called the **verifier** or **prover**, such that an instance of the problem is satisfiable iff $f(x) = \text{true}$ 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 $f(x) = \text{true}$. 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 10.3** (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**.

### 10.4 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 [Karp’72] 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. appendix 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; Kar72; Lev73] 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; of which we have some, and we will continue see more. 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 \(\alpha\)-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 are 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 knowledge about computers.

10.5 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 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 obtaining a polynomial upper bound on the running time by multiplying the polynomial time per problem with the polynomial number of subproblems. We have seen cases where the subproblems are more easily discernible. For example, there is often hope when we can 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.
10.6 Additional notes and references

As mentioned previously, we refer to [AB09; Sip97; Wig19] for more on $P$ vs $NP$ question. Recently, to mark the 50th anniversary 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], [KTo6, Chapter 8], and [Sch, Chapter 6]. We also recommend the video lectures by Sipser on NP-Completeness and the Cook-Levin theorem [Sip20a; Sip20b] for those who are interested. Sipser’s proof of the Cook-Levin is more rigorous than the one presented here.

Lecture materials and comments. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.
Chapter 11

Faster algorithms by divide and conquer

The lectures before this generally emphasized the difference between polynomial and exponential time algorithms. This theme is strongly expressed in our many discussions on recursion with caching (a.k.a. dynamic programming), which at its core, was based on identifying induction hypotheses that break down problems into polynomially many subproblems.

This chapter is also about recursive algorithms that rely on induction for correctness. However, for the problems discussed here, it is already easy to see that they have 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. In particular, they are not simulating brute-force algorithms as was often the case in dynamic programming.

This paradigm is called divide-and-conquer. We have seen it before 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 will generally matching the running theme of “identifying subproblems”. The second “conquer” component, so to speak, will highlight an additional angle of “combining subproblem solutions”.

155
11.1 Multiplication

Our first example (excluding merge-sort) 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 first for the following reasons (besides the usual punt that “this is theory”). First, both the problem and the algorithm are (in our opinion) technically simpler than the other examples that follow. 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. Integer multiplication is so commonplace that we would seem to know everything about it, and one might not suspect any 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 even for simplicity (and otherwise pad both $x$ and $y$ with a leading 0). 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

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

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.

$$\text{multiply}(x[1..n], y[1..n]) = \text{the } (2n\text{-bit integer}) \text{ product of } x \text{ and } y$$

(as $n$-bit integers).

In fig. 11.1, we implement $\text{multiply}(x[1..n], y[1..n])$ 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.
11. Faster algorithms by divide and conquer

11.1. Multiplication

Figure 11.1: Karatsuba’s algorithm for integer multiplication.

11.1.1 Running time analysis

Let $T(n)$ denote the running time of $\text{multiply}(x[1..n], y[1..n])$ 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

$$T(n) \leq 3T(n/2 + 2) + O(n) \quad (11.1)$$

for $n$ larger than a constant (e.g., 5), and $O(1)$ time for $n$ below this constant. This is would be easy for a recursion tree were it not for the extra “+2” in the recursive call. Consider instead the cleaner recurrence

$$S(n) \leq 3S(n/2) + O(n) \quad (11.2)$$

$S(n)$ is clearly similar to $T(n)$; morally, we are interested in large values of $n$, and the omitted additive factor of 2 is diminutive compared to $n/2$ for large $n$. 

158
A clean recursion tree. Let us suppose for the moment that $S(n)$ does model Karatsuba’s algorithm. A quick sketch of a recursion tree for $S(n)$ shows that:

1. There are $3^k$ problems on the $k$th 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

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

(a) is because the sum of coefficients is a geometrically increasing sequence (cf. appendix C.2.3).

Justifying the clean recurrence. Now we have analyzed a “nice” recurrence for $S(n)$ and suggested without proof that this is representative of $T(n)$. Let us now furnish the details.

We define $S(n) = T(n + \alpha)$ for a constant $\alpha > 0$ TBD. We want to choose $\alpha$ so that $S(n) = T(n + \alpha)$ satisfies the nice recurrence (11.2). This can be done by choosing $\alpha = 4/3$ but let us explain how to deduce this value of $\alpha$.

On one hand, the given recurrence for $T$ gives

$$S(n) = T(n - \alpha) \leq 3T \left( \frac{n - \alpha + 2}{2} \right) + O(n).$$

Meanwhile substituting $S(n) = T(n + \alpha)$ into our target recurrence (11.2) gives

$$S(n) \leq 3S(n/2) + O(n) = 3T(n/2 + \alpha) + O(n).$$

To try to make the recurrences equal, let us choose $\alpha$ so that

$$\frac{n - \alpha + 2}{2} = \frac{n}{2} + \alpha;$$

This is solved by $\alpha = 4/3$. Now substituting $\alpha = 4/3$ into the given recurrence (11.1) gives

$$S(n) = T(n - 4/3) \leq 3T(n/2 - 4/3) + O(n) = 3S(n/2) + O(n).$$
as desired. We already know that for this recurrence, \( S(n) = O\left(n^{\log_2(3)}\right) \). To transfer this to \( T(n) \), we have

\[
T(n) = S(n + 4/3) = O\left(n^{\log_2(3)}\right)
\]
as desired.²

11.2 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?

We will develop a linear time algorithm for the more general problem of 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 \( k \)th largest number in \( A \). Again, we can solve this by sorting \( A \), but our goal is to do this faster.

We will develop a very fast, recursive algorithm for selection. To this end, it is necessary to define the intent of our recursive function, which also gives our induction hypothesis for correctness. We can do this even without having developed any algorithm details. We define, for an array \( A[1..n] \) of comparable elements and an integer \( i \in \{1,\ldots,n\} \):

\[
\text{select}(k, A[1..n]) = \text{the} \; \lceil k/2 \rceil \text{th smallest element out of } A[1..n].
\]

We now focus on implementing \( \text{select}(i) \) exactly as specified. As mentioned above, we will be more strategic in the implementation than we typically were with dynamic programming. There, in dynamic programming, the emphasis was on being exhaustive for the sake of correctness. Here, the emphasis is on aggressively cutting down the search space – very much like binary search.

Selection via medians. The strategy we pursue is based on the idea of a pivot. As a thought experiment to motivate this idea, suppose we have a subroutine \( \text{median}(A[1..n]) \) that can find (specifically) the median of \( A \) in linear time. To make this explicitly clear, we define

\[
\text{median}(A[1..n]) = \text{select}(\lfloor n/2 \rfloor, A[1..n])
\]

and assume we can use \( \text{median}(A[1..n]) \) as a \( O(n) \)-time subroutine. \( \text{median}(A[1..n]) \) is a special case of \( \text{select}(k, A[1..n]) \), and it is interesting to ask if the special can be leveraged to accelerate the general case.

²In practice, it is fine to just analyze the nicer recurrence \( S(n) \), perhaps with some comment such as, “we omit a constant additive factor in the recursive call which does not effect the runtime”.

160
11. Faster algorithms by divide and conquer

11.2. Selection

Kent Quanrud
Spring 2022

select(k, A[1..n])
/* select(k, A[1..n]) = the i-th smallest element in A[1..n]. We assume the elements in A are distinct for simplicity. (Otherwise, break ties arbitrarily.) */
/* We assume a linear time subroutine for apx-median(A[1..n]). */

1. If n ≤ 10 then find the k-th element by brute force.
2. Let p = median(A).
3. If k = ⌈n/2⌉ then return p.
4. If k < ⌈n/2⌉
   A. Let B[1..⌈n/2⌉ − 1] be an array of the ⌈n/2⌉ − 1 elements smaller than p.
   B. Return select(k, B)
5. If k > ⌈n/2⌉
   A. Let B[1..n − ⌈n/2⌉] be an array of the n − ⌈n/2⌉ elements larger than p.
   B. Return select(k − ⌈n/2⌉, B)

Figure 11.2: A (conceptual) divide-and-conquer algorithm for select(k, A[1..n]) using median(A[1..n]) to generate a pivot.

Consider an instance of select(k, A[1..n]). Let p be the median of A[1..n], produced by median. We call p the “pivot”. Of course if k = ⌈n/2⌉ then we return p. Otherwise, either k < ⌈n/2⌉ or k > ⌈n/2⌉. If k < ⌈n/2⌉, then we know the rank k element is among the ⌈n/2⌉ − 1 elements smaller than p. We assemble these elements in an array B[1..⌈n/2⌉ − 1], and call select(k, B). Similarly we recurse on the elements bigger than p if k > ⌈n/2⌉, though now we are looking for the (k − ⌈n/2⌉)-rank element remaining. By induction we assume that the recursive calls to select on smaller input correctly returns the desired element. See fig. 11.2 for pseudocode of this algorithm.

The more interesting part of select(k, A[1..n]) is in understanding the running time. Let T(n) denote the maximum running time of select(k, A[1..n]) on any input of size 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 ⌈n/2⌉ elements. So we model T
recursively as follows:

\[
T(1) = 1 \\
T(n) = T(\lfloor n/2 \rfloor) + O(n).
\]

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. appendix C.2.3), the geometrically decreasing coefficients sum to a constant, and we have

\[
T(n) = O(n).
\]

That is, assuming a linear time algorithm for finding the median, we can select any item in linear time.

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

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

That is, \( p \) is always in the middle \((4/5)\)ths of the input. (The choice of 1/10 was arbitrary; the reader may verify afterwards that any constant factor would lead to the same analysis.)

We can use \( \text{apx-median}(A[1..n]) \) to generate a pivot \( p \) in the exact same way as we used \( \text{median}(A[1..n]) \). One difference is that after receiving \( p \), we have to calculate its rank \( \ell \) among \( A \), which takes \( O(n) \) time. Thereafter we split the input by \( p \), and recurse on the appropriate subproblem. See fig. 11.3.

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. So this second version of \( \text{select}(k,A[1..n]) \), using a much weaker subroutine to select the pivot, is still a \( O(n) \)-time algorithm.
11. Faster algorithms by divide and conquer

11.2. Selection

Kent Quanrud
Spring 2022

```c
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.) */
/* Here we implement select(k, A[1..n]) assuming a linear time subroutine for median(A[1..n]). */
1. If $n \leq 10$ then find the $k$th element by brute force.
2. Let $p = \text{apx-median}(A)$.
3. Compare $p$ to each element in $A[1..n]$ to identify its rank $\ell$.
4. If $k = \ell$ then return $p$.
5. If $k < \ell$ then return $\text{select}(k, B)$, where $B[1..\ell-1]$ is an array of the $[\ell] - 1$ elements smaller than $p$.
6. If $k > \ell$ then return $\text{select}(k - \ell, B)$, where $B[1..n - \ell]$ is an array of the $n - \ell$ elements larger than $p$.
```

Figure 11.3: A (second, conceptual) divide-and-conquer algorithm for $\text{select}(k, A[1..n])$ using $\text{apx-median}(A[1..n])$ 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 such a method, called the median-of-medians, invented by Blum, Floyd, Pratt, Rivest, and Tarjan [BFP+72]. The median-of-medians 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

$$ T(n) = T(7n/10) + T(n/5) + O(n). \quad (11.3) $$

Applying the recursion tree method to this recurrence gives $T(n) = O(n)$ – (why?) = linear time!

---
3 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!
11. Faster algorithms by divide and conquer

11.2. Selection

Kent Quanrud
Spring 2022

Figure 11.4: Sketch of the median-of-medians algorithm

```
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(B, \lceil m/2 \rceil, A[1..n])
```

Figure 11.5: The median-of-medians subroutine for approximating the median.

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


(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 (11.3) above. Pseudocode is given in fig. 11.5. Figure 11.7 on page 166 implements select(\(k, A[1..n]\)) with median-of-medians(\(A[1..n]\)).

We now formally analyze median-of-medians(\(A[1..n]\)).
Lemma 11.1. median-of-medians(A[1..n]) returns an element \( p \) with rank \( \ell \) in the range

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

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 greater 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 greater 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 at least as big as one element (namely, itself) from its set. So instead we get \( \ell \geq 3n/10 - 2 \).

Similarly one can show that \( p \) is less than or equal to at least \( 3n/10 - 2 \) elements.

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 \( k \)th level is

\[ (1/5 + 7/10)^k n = (9/10)^k n. \]

Thus the total running time is bounded above by

\[ O\left( \sum_{k=0}^{\infty} (9/10)^k n \right) = O(n). \]
11. Faster algorithms by divide and conquer

11.2. Selection

Kent Quanrud
Spring 2022

```plaintext
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.) */
/* Here we implement select(k, A[1..n]) assuming a linear time subroutine for median(A[1..n]). */
```

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

Figure 11.7: The linear time divide-and-conquer algorithm for \( \text{select}(k, A[1..n]) \) by [BFP+72] using the median-of-medians as a pivot.

**Theorem 11.2.** [BFP+72] Given an array of \( n \) comparable elements \( A[1..n] \) and an index \( k \in [n] \), the \( k \)th 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.

11.3 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.\(^4\) 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.

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

The goal is to implement $\text{min-distance}(P)$ 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

\(^4\)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 algorithm clean and focus on the high-level ideas.
not seem to generalize to general \( y \)-coordinates. In \( \mathbb{R}^2 \), the minimum distance is not necessarily attained by a a consecutive pair of points when listed in increasing order of \( x \).

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 11.2). 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 \( \delta \) be the minimum of the two values returned. \( \delta \) 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 \( \delta \).

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 \( \delta \) 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 \( \delta \) of \( a \). Now consider the following geometric facts.

1. \( P_3 \) lies in a narrow, vertical band of width \( 2\delta \).

2. Any two points in \( P_1 \cap P_3 \) have distance at least \( \delta \), and any two points in \( P_2 \cap P_3 \) have distance at least \( \delta \). This generally forces \( P_3 \) to be spread apart.

Combining the two facts leads to the following observation (formalized in lemma 11.3):

For any point \( p \in P_3 \), there are at most a constant number of other points with \( y \)-coordinate within \( \delta \) of \( p \).
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 \( \delta_1 = \text{min-distance}(P_1) \), \( \delta_2 = \text{min-distance}(P_2) \), 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 \( \delta_3 \) 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 11.8: Computing the minimum distance in \( \mathbb{R}^2 \) by divide-and-conquer along the horizontal axis.

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 11.3, 10 is a sufficiently large constant.)

Pseudocode of the divide-and-conquer algorithm described above is given in fig. 11.8. 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. 11.8.

**Lemma 11.3.** 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.
11. Faster algorithms by divide and conquer

11.3. Minimum distance in $\mathbb{R}^2$

We split the rectangle into two squares

$$L = [a - \delta, a] \times [b - \delta, b]$$
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 $\delta$. 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 $\delta$. 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. 11.8 correctly implements $\text{min-distance}(P)$ 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 $\text{min-distance}(P_1)$ and $\text{min-distance}(P_2)$ 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 $\delta$. Within $P_3$, by lemma 11.3, 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

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

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

### 11.4 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

$$x = a + ib,$$

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 $b$. $i$ denotes “imaginary $i$”, defined by $i^2 = -1$. We have

$$(a + ib)(c + id) = ac + (bc + ad)i + bdi^2 \overset{(a)}{=} ac + (bc + ad)i - bd;$$

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

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$$

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:

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

From this we have Euler’s identity,

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

as well as the identity

$$e^{2\pi i} = 1.$$
Roots of unity. Now, let $n \in \mathbb{N}$ be fixed. An $n$th root of unity is any (complex) number $\omega$ satisfying
\[ \omega^n = 1. \]
For example, 1 is an $n$th root of unity for any $n$, and $-1$ is an $n$th root of unity for any even $n$. $i$ is a 4th root of unity. More generally,
\[ \omega_n \overset{\text{def}}{=} e^{2\pi i/n} \]
is an $n$th root of unity, as well as $\omega_j^n$ for any $j \in \mathbb{Z}$. In fact, the $n$ distinct powers of $\omega_n$,
\[ 1 = \omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1} \]
give all the $n$th 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.$^5$ Here, identifying $p$ with its coefficients $a \in \mathbb{C}^n$, we write
\[ F_n a \overset{\text{def}}{=} \left( p(1), p(\omega_n), p(\omega_n^2), \ldots, p(\omega_n^{n-1}) \right). \]
The conjugate (nth) discrete Fourier transform, denoted $F_n^* a$, instead lists the values for the powers of $\omega_n^{-1}$:
\[ F_n^* a \overset{\text{def}}{=} \left( p(1), p(\omega_n^{-1}), p(\omega_n^{-2}), \ldots, p(\omega_n^{1-n}) \right) \]
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 11.4.** 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 to compute the discrete Fourier transform is to just evaluate each $p(\omega_n^j)$ separately, in $O(n^2)$ time overall. The fast Fourier transform algorithm applies divide and conquer to give the following improved running time.

**Theorem 11.5.** $F_n$ and $F_n^*$ can be computed in $O(n \log n)$ time.

$^5$We mention that there are other “continuous” variants for Fourier transforms that we do not discuss.
11.4.1 Application: multiplying polynomials.

Below we will have a more technical discussion that proves both fact 11.4 and theorem 11.5. 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. But as another basic application that requires no new definitions, consider the problem of multiplying two degree-$(n-1)$ polynomials $p(x)$ and $q(x)$, represented by coefficients $a, b \in \mathbb{C}^n$ in respectively. 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 $r(x) = p(x)q(x)$ at the $(2n)$th roots of unity $\omega_{2n}^k$ are given by

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

for each $i$. In contrast to convolutions, each value takes $O(1)$ time given the values on the RHS. Thus, given $F_{2n}p$ and $F_{2n}q$, computing $F_{2n}r$ is very quick, taking 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 $n$) 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 using the conjugate Fourier transform (and dividing by $2n$) to get the coefficients of $r$. By the fast Fourier transform (theorem 11.5), 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 11.6.** 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.
11.4.2 The fast Fourier transform

We defer the proof of Fact 11.4 until the end, as it is a purely mathematical fact. Let us 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 $(n - 1)$-degree with coefficients $a_0, \ldots, a_{n-1} \in \mathbb{C}$:

$$p(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}.$$  

For ease of discussion let us assume that $n$ is a power of 2. Our goal is to compute the vector of values $F_n a \overset{\text{def}}{=} \left(p(1), p(\omega_n), p(\omega_n^2), \ldots, p(\omega_n^{2^{n-2}})\right)$

To avoid ambiguity we first give a recursive spec for our algorithm. Given $n \in \mathbb{N}$ and a vector $a \in \mathbb{C}^n$ (indexed as $a_0, a_1, \ldots, a_{n-1} \in \mathbb{C}$ by convention), we define

$$\text{Fourier}(a_0, \ldots, a_{n-1}) = \text{the } n\text{th discrete Fourier transform}$$

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

where $p(x)$ is the polynomial $p(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$.

The divide and conquer argument exploits the following identity for $\omega_n$ (for all $n \in \mathbb{N}$):

$$\omega_{2^n} = \omega_n^2.$$  \hfill (11.4)

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

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

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

$$(F_{2n}a)_k \overset{(a)}{=} a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1}$$

$$\overset{(b)}{=} \left(a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{(2n-2)}\right)$$

$$+ \left(a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{(2n-1)}\right).$$  \hfill (11.5)

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 $\omega_{2n}$ are even, so by applying the identity (11.4), 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 $\omega_{n}^k$. More specifically, if we let $a_{\text{even}} = (a_0, a_2, \ldots, a_{2n-2})$, then we have

$$a_0 + a_2 \omega_{n}^k + \cdots + a_{2n-2} (\omega_{n}^k)^{n-1} = F_{a_{\text{even}}}$$

– the $n$th Fourier transform of the even-index coefficients!

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 (11.5). We have

$$a_1 \omega_{2n}^k + a_3 (\omega_{2n}^k)^3 + \cdots + a_{2n-1} (\omega_{2n}^k)^{(2n-1)} = \omega_{2n}^k \left(a_1 + a_3 (\omega_{n}^k)^2 + a_5 (\omega_{n}^k)^4 + \cdots + a_{2n-2} (\omega_{n}^k)^{n-1}\right)$$

$$= \omega_{2n}^k \left(a_1 + a_3 (\omega_{n}^k)^2 + a_5 (\omega_{n}^k)^4 + \cdots + a_{2n-2} (\omega_{n}^k)^{n-1}\right)$$

$$= \omega_{2n}^k (F_{a_{\text{even}}})_{\text{odd}}.$$  

Plugging all this back into (11.5), we have the clean identity

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

for $k = 0, \ldots, n-1$. This equation tells us that computing $F_{2n}a$ is largely reduced to computing $F_{a_{\text{even}}}$ and $F_{a_{\text{odd}}}$. More specifically, given $F_{a_{\text{even}}}$ and $F_{a_{\text{odd}}}$, we need only $O(2n)$ time to compute $F_{2n}a$. Meanwhile $F_{a_{\text{even}}}$ and $F_{a_{\text{odd}}}$ can be computed recursively, where each subproblem has half the size / dimension.

In fig. 11.9 on page 176, we give pseudocode for the recursive algorithm $\text{Fourier} (a_0, \ldots, a_{n-1})$, based on the divide-and-conquer approach outlined above. The code in fig. 11.9 is modified slightly for mathematics above to remove the assumption that $n$ is even. (The argument for odd $n$ is the exact same, except there is one more even-index coefficient than odd-index coefficient.)

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

$$T(1) = O(1)$$

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

175
11. Faster algorithms by divide and conquer

11.4. Fourier transforms

Fourier\left( a_0,\ldots,a_{n-1} \right)

\begin{verbatim}
/* We assume for simplicity that n is even. */

1. If \( n = 0 \) then return the empty vector.
2. If \( n = 1 \) then return \( a_0 \).
3. Let \( a_{\text{even}} = (a_0, a_2, a_4, \ldots) \) be the vector of \( \lceil n/2 \rceil \) even-index coefficients and let \( a_{\text{odd}} = (a_1, a_3, a_5, \ldots) \) be the vector of \( \lfloor n/2 \rfloor \) odd-index coefficients. // \( O(n) \)
4. Let \( (y_0, \ldots, y_{\lceil n/2 \rceil - 1}) = \text{Fourier}(a_{\text{even}}) \) // \( T(\lceil n/2 \rceil) \)
5. Let \( (z_0, \ldots, z_{\lfloor n/2 \rfloor - 1}) = \text{Fourier}(a_{\text{odd}}) \). // \( T(\lfloor n/2 \rfloor) \)
6. For \( k = 0, 1, \ldots, n-1 \), let \( x_k = y_k + \omega^n z_k \) // \( O(n) \)
7. Return \( (x_0, \ldots, x_{n-1}) \).
\end{verbatim}

Figure 11.9: The fast Fourier transform.

This is the same recurrence as in \texttt{merge-sort} (cf. section 1.2.4) and is solved by

\[ T(n) = O(n \log n). \]

This completes the proof of theorem 11.5.

11.4.3 Inverting \( F_n \) and \( F_n^* \)

Lastly we prove Fact 11.4, which described a near-inverse relationship between \( F_n \) and \( F_n^* \). Fact 11.4 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.

\textbf{Lemma 11.7.} For any \( n \)th root of unity \( \omega \) except for \( \omega = 1 \), we have

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

For \( \omega = 1 \), the LHS is \( n \).

\textit{Proof.} We have

\[ (1 - \omega) \left( 1 + \omega + \omega^2 + \cdots + \omega^{n-1} \right) = 1 - \omega^n = 0, \]

so either \( \omega = 1 \) or \( 1 + \omega + \cdots + \omega^{n-1} = 0 \). \qed

176
Now we prove Fact 11.4. We restate the claim below for the reader’s convenience.

**Fact 11.4.** 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 \( h \in \{0, \ldots, n - 1\} \), we have

\[
(F_n^* F_n a)_h \overset{(a)}{=} \sum_{j=0}^{n-1} (F_n a)_j (\omega_n^{-h})^j \overset{(b)}{=} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_k (\omega_n^j)^k (\omega_n^{-h})^j \\
\overset{(c)}{=} \sum_{k=0}^{n-1} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j \overset{(d)}{=} na_h.
\]

Here (a) and (b) expand out the definitions of \( F_n^* \) and \( F_n \). (c) interchanges sums. (d) is by lemma 11.7: unless \( k - h = 0 \) (or some other multiple of \( n \)), we have \( \sum_{j=0}^{n-1} (\omega_n^{k-h})^j = 0 \). \( \blacksquare \)

One implication of Fact 11.4 is as follows.

**Corollary 11.8.** A polynomial \( p(x) \) of degree \( n - 1 \) is uniquely identified by its value at the \( n \) roots of unity \( p(1), p(\omega_n), \ldots, p(\omega_n^{n-1}) \).

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

### 11.5 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

\[
(AB)_{1j} = \sum_k A_{1k} B_{kj}.
\]

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

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},
\]

where \( A_{11}, \ldots, B_{22} \in \mathbb{R}^{n/2} \) are all square matrices with half the dimension. Recall that

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

177
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 $k$th level has $7^k$ subproblems each with $O(n^{2^k})$ work.
2. There are at most $\lceil \log n \rceil$ levels.

Thus the total running time is

$$O\left(\sum_{i=1}^{\log(n)} \left(\frac{7}{2}\right)^i n\right) = O\left(n^{\log_7(7)}\right) \approx O\left(n^{2.8074}\right).$$

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.

\begin{align*}
(A_{11} + A_{22})(B_{11} + B_{22}) &= A_{11}B_{11} + A_{22}B_{11} + A_{11}B_{22} + A_{11}B_{22} & (11.6) \\
(A_{11} - A_{21})(B_{11} - B_{12}) &= A_{11}B_{11} - A_{21}B_{11} - A_{11}B_{12} + A_{21}B_{12} & (11.7) \\
(A_{12} - A_{21})(B_{21} + B_{22}) &= A_{12}B_{12} - A_{21}B_{21} - A_{21}B_{22} + A_{12}B_{22} & (11.8) \\
A_{11}(B_{12} - B_{22}) &= A_{11}(B_{12} - B_{22}) & (11.9) \\
(A_{11} + A_{12})B_{22} &= A_{11}B_{22} + A_{12}B_{22} & (11.10) \\
(A_{11} + A_{22})(B_{11} + B_{22}) &= A_{11}B_{11} + A_{11}B_{11} + A_{22}B_{11} + A_{22}B_{22} & (11.11) \\
(A_{11} - A_{21})(B_{11} + B_{12}) &= A_{11}B_{11} - A_{21}B_{11} + A_{11}B_{12} - A_{21}B_{12} & (11.12)
\end{align*}

Then

$$\begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix} = \begin{pmatrix}
(11.6) + (11.7) - (11.10) + (11.8) & (11.9) + (11.10) \\
(11.11) + (11.7) & (11.9) + (11.6) - (11.11) - (11.12)
\end{pmatrix}$$

Even more clever combinations and strategies exist which has driven the exponent down further. Currently, the best running time is $O(n^\omega)$ for $\omega \approx 2.37$ [AW20]. It is conjectured that $\omega$ will eventually be driven down to 2 (a natural lower bound). We should point out that the theoretically best algorithms are so complicated that not necessarily preferred practice. (The hidden constant strikes back!)
11. Faster algorithms by divide and conquer

11.6 Additional notes and references

Lecture 11 materials and comments. 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 12 materials and comments. Click on the links below for the following files:

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

11.7 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 explanation of how to use the recursive algorithm to solve the original problem.
4. An analysis of the running time.
5. A proof by induction, where the induction hypothesis is based on the recursive spec.

Exercise 11.1. 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 $O(n \log n)$.\(^6\)

3. Prove your algorithm is correct by induction, where the recursive spec is your induction hypothesis.

4. Analyze the running time of your algorithm.

**Exercise 11.2.** Here we assume the same model as exercise 11.1 – an array $A[1..n]$ of incomparable elements\(^7\) – 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.\(^8\)

**Exercise 11.3.** 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 $A$.\(^9\)

\(^6\)There is also a well-known $O(n)$ time algorithm which we are not asking for.

\(^7\)In addition to disallowing sorting, no hash tables are allowed as well.

\(^8\)Your algorithm should be faster than $O\left(\frac{n^2}{k}\right)$ (which is the trivial running time) for, e.g., $k = n^{1/3}$.

\(^9\)There may be more than one fixed point; any fixed point is fine.
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.

**Exercise 11.4.** 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 $P$. 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 $A[1..m]$ 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 $k$th largest element among the combined elements of $A$ and $B$. For simplicity you make assume that all the elements are distinct.\(^{10}\)

1. Define a recursive spec for a recursive algorithm to solve this problem.

2. Show that if either $m$ or $n$ is $\leq 5$ then we can find the $k$th 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 11.6.** 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

$$z = x + y$$

---

\(^{10}\)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.
where $x \in X$, $y \in Y$, and the choice of $(x,y) \in X \times Y$ is unique.\footnote{That is, for fixed $z$, there is only one choice of $x \in X$ and one choice of $y \in Y$ such that $z = x + y$.} You may assume that all the integers in $X$ are distinct (amongst themselves), and that all the integers in $Y$ are distinct (amongst themselves).\footnote{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.}

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 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 $Y$.\footnote{It might be helpful to work through very simple examples such as $X = Y = \{1\}$ and $X = Y = \{0,1\}$.}

3. (2 pt.) Suppose we now that 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.\footnote{It might help to work through $k = 4$ to build your intuition.}

**Hint:** For part 2, try “changing the input” to an algorithm from section 11.4 (that is not the FFT, at least directly).

**Exercise 11.7.** 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 $\alpha$ 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 $\beta < 1$, $T(n) \leq O(n^\gamma \log(n))$. 11
Chapter 12

Polynomial lower bounds

At the moment, we do not have any running time for SAT faster than \( O(2^n \text{poly}(m,n)) \). The strong exponential time hypothesis conjectures that it is impossible to solve SAT in \( O(2^{(1-\epsilon)n \text{ poly}(m)}) \) 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.

12.1 Disjoint sets

Let \( A = \{x_1, \ldots, x_n\} \subset \{0,1\}^m \) and \( B = \{y_1, \ldots, y_n\} \subset \{0,1\}^m \) 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 thinking 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 \( x_i \in A \) and \( y_j \in B \)?

Equivalently, in terms of vectors:

Is there an orthogonal pair of vectors \( x_i \in A \) and \( y_j \in B \)?

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 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? One might doubt that the problem has enough structure to take advantage of and do better. But we have already seen some algorithmic surprises under similar
circumstances and it is hard to rule out the possibility of a clever idea. That said, we can give the following conditional lower bound, which states that a substantially better dependency on $n$ would have dramatic implications for SAT as well.

**Theorem 12.1.** There exists a universal constant $c > 0$ such that, for all $\epsilon > 0$, a $O(n^{2-\epsilon} \text{poly}(m))$ time algorithm for the orthogonal vectors problem implies a $O\left(2^{(1-\epsilon)n} \text{poly}(m)\right)$-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 $f(x_1, \ldots, x_{n/2}, y_1, \ldots, y_{n/2})$.

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

$$(x_j \lor \overline{y_k})$$

is satisfied by any $x \in \{0, 1\}^{n/2}$ such that $x_j = \text{true}$, or by any $y \in \{0, 1\}^{n/2}$ such that $y_k = \text{false}$. 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

$$A_x = \{\text{clauses } C_i \text{ not satisfied by } x\} \subseteq \{C_1, \ldots, C_m\}.$$  

Similarly, for each $y \in \{0, 1\}^{n/2}$, we define a set

$$B_y = \{\text{clauses } C_i \text{ not satisfied by } y\} \subseteq \{C_1, \ldots, C_m\}.$$  

We define

$$\mathcal{A} = \{A_x : x \in \{0, 1\}^{n/2}\} \quad \text{and} \quad \mathcal{B} = \{B_y : y \in \{0, 1\}^{n/2}\}.$$  

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 $f(x, y) = \text{true}$.

We have $|\mathcal{A}| = |\mathcal{B}| = 2^{n/2}$. Thus a $O(n^{2-\epsilon} \text{poly}(d))$ algorithm for orthogonal vectors (with $n$ sets in dimension $d$) implies a $O\left(2^{(2^{n/2})^{2-\epsilon}} \text{poly}(m)\right) = O\left(2^{(1-\epsilon/2)n} \text{poly}(m)\right)$ time algorithm for SAT. \(\blacksquare\)

\(^1\)for simplicity; otherwise, add a dummy variable.
12.2 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 $s$ and $t$. 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. A simple algorithm single source shortest paths from every $s \in S$. This gives a $O(k(m + n \log n))$ running time, where we use Dijkstra’s shortest path algorithm. In unweighted graphs, this simplifies to $O(km)$.

**Theorem 12.2.** 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 $a_i = 1$, 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.

12.3 Longest common subsequence

Recall the longest common substring problem. We are given two strings $x$ and $y$ from some finite alphabet. A common substring is a string $z$ that appears
(non-consecutively) as a substring of both \( x \) and \( y \). We denote the length of the longest common subsequence of \( x \) and \( y \) by \( \text{LCS}(x, y) \). The longest common substring 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 straightforward dynamic program?

For simplicity we will consider longest common substring for bit strings \( x, y \in \{0, 1\}^m \) of the same length \( m \).

**Theorem 12.3.** 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.

We break down the proof into two main components. Recall that in the orthogonal vectors problem input we are given two sets of vectors \( A = \{x_1, \ldots, x_n\} \) and \( B = \{y_1, \ldots, y_n\} \). Our goal is to identify a single pair \( x_i \in A \) and \( y_j \in B \) such that \( \langle x_i, y_j \rangle = 0 \). The first step is to somehow simulate the inner product with an LCS computation. This is addressed by our first main lemma which describes a LCS-based gadget for verifying orthogonality.

**Lemma 12.4.** Let \( m \in \mathbb{N} \) be fixed. There are functions \( h_1, h_2 : \{1, 2\}^m \to \{1, 2, 3, 4\}^M \) 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 \( x, y \in \{0, 1\}^m \),

\[
\text{LCS}(h_1(x), h_2(y)) = \begin{cases} 
T + 1 & \text{if } \langle x, y \rangle = 0 \\
T & \text{if } \langle x, y \rangle > 0 
\end{cases}
\]

We prove this lemma below in section 12.3.1.

While we can now implicitly identify an orthogonal pair of vectors via LCS, the real emphasis of orthogonal vectors is identifying a pair of vectors faster than an all-to-all comparison. The next lemma describes how to simulate an all-to-all-comparison with a single edit distance function.

**Lemma 12.5.** 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 \( B_j \),

\[
T \leq \text{LCS}(A_i, B_j) \leq T + 1.
\]

Then in \( O(Mn) \) time, one can construct strings \( \tilde{A}, \tilde{B} \in \{1, 2, 3, 4, 5\}^{M'} \) of length \( M' = O(Mn) \), and select an integer \( U \), such that
12. Polynomial lower bounds

12.3. Longest common subsequence

We prove this lemma below in section 12.3.2.

If we assume the two lemmas above to be true, then we have the ingredient to prove theorem 12.3. Let us do so now, and address each lemma separately afterwards.

Proof of theorem 12.3. Given two sets $A$ and $B$ of $n$ bit strings in $m$ dimensions, by combining the above two lemmas, two strings $\vec{A}$ and $\vec{B}$ of length $O(n \text{poly}(m))$ such that $\text{LCS}(\vec{A}, \vec{B})$ indicates whether there is an orthogonal pair. A subquadratic algorithm for $\text{LCS}$, applied to $\vec{A}$ and $\vec{B}$, would imply a subquadratic (in $n$) algorithm for orthogonal vectors.

12.3.1 The LCS Orthogonality Gadget

We build up the gadget described in lemma 12.4 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 12.6. There are function $f_1, f_2 : \{0, 1\} \rightarrow \{1, 2\}^2$ such that for any two bits $x_1, x_2 \in \{0, 1\}$,

$$\text{LCS}(f_1(x_1), f_2(x_2)) = 1 - x_1x_2.$$  

Proof. Consider the following table computing the LCS between particular two bit strings.

<table>
<thead>
<tr>
<th>LCS</th>
<th>12</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Consequently we define

$$f_1(0) = 01, \quad f_1(1) = 11 \quad f_2(0) = 10 \quad f_2(1) = 00,$$

which satisfies the claim. ■
Bit strings

**Lemma 12.7.** There are functions $g_1, g_2 : \{1, 2\}^m \rightarrow \{1, 2, 3\}^M$, for $M = O(m^2)$, with the following properties.

1. $g_1$ and $g_2$ both take $O(M)$ to compute.
2. For all $x_1, x_2 \in \{0, 1\}^m$,
   \[
   \text{LCS}(g_1(x_1), g_2(x_2)) = 2m^2 - \langle x_1, x_2 \rangle.
   \]

**Proof.** By lemma 12.6, 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
\[
C = 3^{m'} = 33 \ldots 3
\]
where 3 is repeated $m'$ times. We use $C$ as a sort of divider between coordinates. We define $g_1$ and $g_2$ by
\[
\begin{align*}
g_1(x_1) &= C \cdot f_1(x_1) \cdot C \cdot f_1(x_2) \cdot C \cdot f_1(x_1) \cdot C \cdot f_2(y_1) \cdot C \cdot f_2(y_2) \cdot C \cdot f_2(y_m) \cdot C, \\
g_2(x_2) &= C \cdot f_2(y_1) \cdot C \cdot f_2(y_2) \cdot C \cdot f_2(y_3) \cdot C \cdot f_2(y_m) \cdot C,
\end{align*}
\]
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
\[
\text{LCS}(y_1, y_2) \geq (m + 1)m' + \sum_{i=1}^m \text{LCS}(f_1(x_i), f_2(y_i)).
\]

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.** Any 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 $i \neq j$, 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 2’s on the right of $A_i$ will not be matched. Similarly if $i > j$, then at least $m'$ of the 2’s on the left of $A_i$ will not be matched. Otherwise each of the $A_i$’s can contributed 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

$$\text{LCS}(g_1(x), g_2(y)) \leq (m + 1)m' + \sum_{i=1}^{m} \text{LCS}(f_1(x_i), f_2(y_i)).$$

Combined with the lower bound in the first claim, we obtain the equality

$$\text{LCS}(g_1(x), g_2(y)) = (m + 1)m' + \sum_{i=1}^{m} \text{LCS}(f_1(x_i), f_2(y_i)).$$

Moreover, we have $\text{LCS}(f_1(x_i), f_2(y_i)) = 1 - x_i y_i$ for all $i$ by lemma 12.6. Substituting into the above completes the proof of the lemma.

Normalization

**Lemma 12.4.** Let $m \in \mathbb{N}$ be fixed. There are functions $h_1, h_2 : \{1, 2\}^m \to \{1, 2, 3, 4\}^M$ 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 $x, y \in \{0, 1\}^m$,

$$\text{LCS}(h_1(x), h_2(y)) = \begin{cases} T + 1 & \text{if } \langle x, y \rangle = 0 \\ T & \text{if } \langle x, y \rangle > 0 \end{cases}$$

**Proof.** Let $g_1, g_2 : \{1, 2\}^m \to \{1, 2, 3\}^M$ be given by lemma 12.7, where $M$ is a polynomial in $m$. Recall that we have

$$\text{LCS}(g_1(x), g_2(y)) = 2m^2 - \langle x, y \rangle$$

for any two $x, y \in \{0, 1\}^m$.

Let $T = 2m^2 - 1$. Consider the function $h_1, h_2 : \{0, 1\}^m \to \{0, 1\}^{M+T}$ defined by

$$h_1(x) = 4^T g_1(x)$$

$$h_2(x) = g_2(x)4^T,$$
where \(4^T\) 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 4’s, or between \(g_1(x)\) and \(g_2(x)\), but not both. As such, we have

\[
\text{LCS}(h_1(x), h_2(y)) = \max\left\{ T, \text{LCS}(g_1(x), g_2(y)) \right\} = \begin{cases} 
2m^2 = T + 1 & \text{if } \langle x, y \rangle = 0 \\
T & \text{otherwise},
\end{cases}
\]
as desired.

### 12.3.2 The LCS \(\lor\)-gadget

Our goal in this section is to prove the second main lemma, which we first restate for convenience.

**Lemma 12.5.** 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 \(B_j\),

\[
T \leq \text{LCS}(A_i, B_j) \leq T + 1.
\]

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

\[
\text{LCS}(\bar{A}, \bar{B}) > U \iff \text{LCS}(A_i, B_j) = T + 1 \text{ for some } A_i \text{ and } B_j.
\]

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 \(\text{LCS}(A_i, B_j) = T + 1\). 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 \(O(n^{1.99} \text{poly}(M, T))\)-time running time identify such a pair \(A_i\) and \(B_j\) gives a \(O(n^{1.999} \text{poly}(M, T))\) 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 \(O(\text{poly}(M, T)n)\). We will design \(\bar{A}\) and \(\bar{B}\) so that \(\text{LCS}(\bar{A}, \bar{B})\) (somehow) reveals whether or not we have \(\text{LCS}(A_i, B_j) = T + 1\) for any pair of indices \(i\) and \(j\).

Recall that the \(A_i\)’s and \(B_j\)’s consist of characters from the set \(\{1, 2, 3\}\). Let \(D = 5^S\) be the string of \(S\) consecutive 5’s, for a sufficiently large parameter \(S \in \mathbb{N}\). 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 \(\text{LCS}(\bar{A}, \bar{B})\) will be incentivized
to match up the maximum number of \(D\)-blocks, \(2n - 1\). Any maximum matching between the \(2n - 1\) \(D\)-blocks of \(\hat{B}\) to the \(3n - 1\) \(D\)-blocks to \(\hat{A}\), 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 
\[
(2n - 1) \text{LCS}(D, D) + \sum_{i=1}^{n} \text{LCS}(A_i, B_{\pi(i)}) \quad \text{for some mapping } \pi : [n] \rightarrow [n] \text{ of } A_i\text{’s to } B_j. 
\]
The sum is \(> nT\) iff \(\text{LCS}(A_i, B_{\pi(i)})\) for one of these matches \(i\).

The formal argument consists of two parts. Let
\[
U = (2n - 1)S + nT
\]
Observe that \(\text{LCS}(\hat{A}, \hat{B}) \geq U\) 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 \(\text{LCS}(A_i, B_j) > T\) for some pair of \(A_i\) and \(B_j\).

**Lemma 12.8.** Suppose \(\text{LCS}(A_i, B_j) = T + 1\) for some \(i, j\). Then \(\text{LCS}(\hat{A}, \hat{B}) \geq U + 1\).

**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.

```
KBEKKDA1DA2DA3DA4DA5DDDDDB
BBDB3DB4DB5DB6DB7DB8DB9DB10DB11
```

The total cost of the matching is at least \((n - 1)T + T + 1 + (2n - 1)S\), since \(\text{LCS}(A_i, B_j) = T + 1\), 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. \(\blacksquare\)

**Lemma 12.9.** Let \(S \geq M - T\). Suppose \(\text{LCS}(A_i, B_j) \leq T\) for all \(i, j\). Then \(\text{LCS}(\hat{A}, \hat{B}) \leq U\).

**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 \(\hat{B}\) 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 $LCS(A_i, B_j)T$.

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 $S_i$ because we had initially assumed all of the $D$-blocks in $ar{B}$ 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

$$LCS(\bar{A}, \bar{B}) = (2n - 1)S + \sum_{i=1}^{n} \text{(net contrib. of } A_i) \leq (2n - 1) + nT,$$

as desired.

### 12.4 Additional notes references

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 12.2 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].

### 12.5 Exercises

**Exercise 12.1.** Assuming the strongly exponential time hypothesis, give a lower bound on the running time for any exact algorithm for subset sum.

**Exercise 12.2.**

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 $\alpha$-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 $\alpha$-approximation for computing the diameter of an unweighted and undirected graph, assuming the strong exponential time hypothesis.

\(^2\)The larger the better – I think that $3/2$ is doable. Anything greater than 1 is already interesting.
Chapter 13

Spanning trees

Let $G = (V, E)$ be an undirected graph with $m$ edges and $n$ vertices. Recall that a circuit in $F$ 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.

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

$$\text{span}(S) = \{ e \in E : S \text{ spans } e \}.$$ 

Observer that $\text{span}(\text{span}(S)) = \text{span}(S)$ for all $S$.

This chapter focus 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.

## 13.1 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 $\kappa$ 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 $F \neq \emptyset$, 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 concerns the maximum number of edges in any forest, and the minimum number of edges in any spanning set, in a connected graph. What is perhaps most interesting about the following lemma is the relationship it uncovers between the comparative sizes of forests and spanning sets: it says every spanning set has at least as many edges as any forest.

**Lemma 13.1.** 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 follow lemma extends the previous ones to disconnected graphs, by treating each connected component as if it were a connected graph on its own.

**Lemma 13.2.** Let $G = (V, E)$ be an undirected graph with $\kappa$ 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 $S_i = F \cap E_i$ be the restriction of $F$ to $G_i$. Let $n_i = |V_i|$. By lemma 13.1, 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 13.2 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 play off each other.

**Theorem 13.3.** 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 effect connectivity. But then $S$ is not minimal, a contradiction. ■

One takeaway is that all minimal spanning edge have the same size, and all maximal edge sets have the same size, which is not obvious *a priori*. It is extremely convenient that minimal equals minimum and maximal equals minimum – it suggests a sort of discrete analogue of convexity and concavity. We highlight again the way to analyzing forests and spanning sets together give a short proofs of powerful facts about forests and spanning sets taken alone. This is our first example of a more general phenomena called *packing and covering duality*.

**Theorem 13.3** 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 13.3 is a forest. The second algorithm terminates with a maximal forest, which by theorem 13.3 is also spanning.

An important special case is when a graph is connected. In this case, the spanning forest is a spanning tree.

**Corollary 13.4.** *Every connected graph has a spanning tree, with $n - 1$ edges.***
13.2 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 able 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 key lemma.

**Lemma 13.5.** Let $e \in E$. Then the following conditions are equivalent.

1. Every minimum weight spanning tree contains $e$.
2. For every set of edges $S \subseteq E - e$ that spans $e$ (and doesn’t include $e$),
   \[ w(e) < \max_{f \in S} w(S). \]

**Proof.** (1) implies (2). Suppose every minimum weight spanning tree $T$ contains $e$. Suppose by contradiction that (2) does not hold. 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 $T' \overset{def}{=} T - e + f$ is a forest. Since $T'$ has $n - 1$ edges, it is also a spanning tree. It has weight

\[ \bar{w}(T') = \bar{w}(T) - w(e) + w(f) \overset{(a)}{<} \bar{w}(T) \]

where (a) is because $w(f) < w(e)$ by assumption. But then $T$ is not a minimum weight spanning tree, a contradiction.

(2) implies (1). Suppose (2) holds. 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 $T - f + e$. 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. ■

Let us call an edge $e$ **good** if it satisfies condition (2) of lemma 13.5; in particular, all good edges are in all MST’s.

Now, consider the following abstract algorithm that simply tries to add good edges. Now there are 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. For the sake of discussion, we simply pretend we can identify good edges as well. Later we will identify concrete strategies to do so.

**generic-MST**($G = (V,E), w : E \rightarrow \mathbb{R}$)

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 13.5) must be in every minimum weight spanning tree. However it is not clear that these edges are sufficient to find the minimum spanning tree. Still, almost tautologically, we have the following.

**Lemma 13.6.** 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 13.5, 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.
13.2.1 The greedy algorithm

The following lemma gives our first concrete certificate for good edges.

**Lemma 13.7.** 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. \hfill \blacksquare

The key point to the above lemma is that the set $H$ gives a certificate that $e$ should be included 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`: repeated 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.

```plaintext
Greedy-MST(G = (V,E), w : E → R)

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in increasing order of weight.
2. Set $T ← \emptyset$.
3. For $i = 1, \ldots, n$:
   /* 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 $T ← T + e_i$.
4. Return $T$.
```

**Running time and the disjoint union data structure.** Correctness follows immediately as a special case of `generic-MST`. 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 if two vertices are already in the same component.

These operations are facilitated by the **disjoint union** data structure, which has 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 \( x \) and \( y \) 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 13.8.** There is a data structure that can implement the **union**, **same-set**, and **new-set** operations listed above, 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 13.9.** The greedy algorithm computes the MST in \( O(m \log n) \) time.

**Proof.** We have already addressed the running time above in lemma 13.6. 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.

Note that the \( O(m + n \alpha(n)) \) bound for the disjoint-union data structure was overkill, since sorting was already a bottleneck.

### 13.2.2 A parallel algorithm

For a set of vertices \( S \subset V \), the **cut** induced by \( S \), denoted \( \partial(S) \), is the set of edges with exactly one endpoint in \( S \):

\[
\partial(S) \overset{\text{def}}{=} \{u, v\} \in E : u \in S, v \notin S
\]

The name “cut” comes from the fact that removing \( \partial(S) \) from \( G \) cuts off \( S \) from the rest of the graph.
13. Spanning trees

13.2. Weighted graphs

 greedymst($G = (V,E), w : E \rightarrow \mathbb{R}$)

/* 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 $E = \{e_1, \ldots, e_n\}$ in increasing order of weight. // $O(m \log n)$
4. Set $T \leftarrow \emptyset$.
5. For $i = 1, \ldots, n$
   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 13.1: The greedy algorithm for MST implemented with the disjoint union data structure.

Lemma 13.10. Let $S \subseteq V$ be any set of vertices. Then the minimum weight edge in the cut induced by $S$ is a good edge.

Proof. Let $H = E - \partial(S)$. Then $E \setminus \text{span}(H) = \partial(S)$. The claim now follows from lemma 13.7. ■

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 $\partial(S)$. Add all of these edges to $T$.

parallel-mst($G = (V,E), w : E \rightarrow \mathbb{R}$)

/* Also known as Borůvka’s algorithm. */

1. $T \leftarrow \emptyset$
2. While $T$ more than 1 connected component:
   A. For each component $S$ of $T$, identify the smallest weight edge in $\partial(S)$.
   B. Add all of these edges to $T$.
3. Return $T$

202
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.

The contraction version of the parallel algorithm 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.

```plaintext
parallel-MST(G = (V, E), w : E → ℝ)
/* Also known as Borůvka’s algorithm. */
1. \( T ← \emptyset \)
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 13.11.** \( \text{parallel-MST} \) computes the minimum spanning tree, and can be implemented to run in \( O(m \log(n)) \).

**Proof.** Correctness follows from lemma 13.6 and lemma 13.10. 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 the corresponding endpoints. 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 it is that it is inherently parallel. Each vertex in the contracted graph can pick it’s 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.

### 13.2.3 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 to add the next component. Finding the next edge can be implemented very efficiently with a Fibonacci heap.

```plaintext
search-MST(G = (V, E), w : E → R)
/* Also known as Prim’s algorithm. */
1. Set $T ← \emptyset$ and $S ← \{v\}$ 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 $\partial(S)$.
   B. Set $T ← T + e$, $S ← S \cup e$.
3. Return $T$.
```

**Theorem 13.12.** search-MST returns the MST, and can be implemented to run in $O(m + n \log n)$ time.

\(^1\)We also point out that the theoretically fastest algorithms [Cha00; KKT95], not covered here, are based on Borůvka’s algorithm. See also exercise 13.9.
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 \(O(m + n \log n)\).■

13.2.4 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 make 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 unique, and (b) the algorithm would still return the same set of edges. One such way is to add \(\epsilon i\) to the weight of an edge taken in the \(i\)th iteration, and \(\epsilon n\) to the weight of any edge not taken, where \(\epsilon > 0\) is sufficiently small such that this change only acts as a tiebreaker. Our analysis above applies directly to this thought experiment, which in turn justifies the actual algorithm run on non-distinct weights.

13.3 Steiner trees

We now turn to a natural and useful generalization of spanning trees called Steiner tree. Let \(G = (V, E)\) be an undirected graph. Let \(U \subset V\) be a subset of vertices called terminals. A Steiner tree over \(T\) 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 ★’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 13.13.** The Steiner tree problem is \( \text{NP-Hard} \), 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 \( C_1, \ldots, C_m \).

1. For each variable \( x_j \), a terminal vertex.
2. For each clause \( C_i \), a terminal vertex.
3. For each assignment such as \( x_j = \text{true} \) or \( x_j = \text{false} \), a (non-terminal) vertex.

This creates a total of \( 3n + m + 1 \) 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

\[
2n + 3m + \binom{2n}{2} = O(m + n^2)
\]

edges. A high-level diagram of the construction is given below.
The idea behind the proof is as follows. Ultimately, we need to connect the variables and clauses, via assignments. Let us say that a Steiner tree “takes an assignment” if it the edge between the corresponding edge and assignment is included in the tree. The terminals corresponding to the terminals requires the Steiner tree to take at least one edge incident to each variable. The terminals corresponding to each clauses requires us to take at least one edge incident to every clause. We can also assume that every minimum Steiner tree has exactly one edge to each variable and to each clause, because when there is more than one, we can replace one of them with an edge between the corresponding assignments. This implies that every Steiner tree contains at least \( m + n + n - 1 = m + 2n - 1 \) edges – \( n \) for the variables, \( m \) for the clauses, and then (at least) \( n - 1 \) to connect the assignments of the variables to one another. A satisfying assignment for \( f \) maps to a Steiner tree with \( m + 2n - 1 \) edges by connecting the clauses and variables to the corresponding assignments with \( m + n \), and then connecting the \( n \) assignments we are using arbitrarily with \( n - 1 \) to connect the assignments of the variables to one another. A satisfying assignment for \( f \) gives a satisfying solution for \( f \).

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 \( x \). We take the corresponding \( n \) edges between variables and assignments. For each clause, we select an edge corresponding to the single-variable assignment that satisfied 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 \( x_i = \text{true} \) and \( x_i = \text{false} \), and \( T \) remains a Steiner tree. Likewise if a clause is two edges incident to two different satisfying vertex assignments, we can delete one of these edges and replace it with an edge between the 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 that are at the opposite endpoints. Suppose there are $k$ of them. None of these assignments are connected to each other by the $m+n$ edges, and 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. ■

Lecture materials and comments. Click on the links below for the following files:

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

### 13.4 Exercises

Please see [Eri19, Chapter 7] for further exercises.

**Exercise 13.1.** 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. \footnote{To be clear, a cycle specifically means a circuit with no repeating vertices (except at the common vertex where the walk starts and ends).}

2. $S$ is spanning iff $S$ contains a spanning forest.

3. $S$ is spanning iff $|S| \geq n-1$.

4. $G$ has distinct edge weights if $G$ 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 13.2.** Consider the following algorithm that claims to compute the MST (in a connected graph).
13. Spanning trees

13.4. Exercises

Kent Quanrud
Spring 2022

decremental-MST

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in decreasing order of weight.

2. For $t = 1, 2, \ldots, n$:
   
   A. Unless $E - e_t$ is disconnected:
      
      1. $E \leftarrow E - e_t$.

   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.

**Exercise 13.3.** 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 13.4.** 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.

**Exercise 13.5.** 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 13.6.** Let $G = (V, E)$ be an undirected graph with distinct nonnegative edge lengths. The bottleneck length of a path in $G$ is the length of the longest edge in the path. Prove that the MST of $G$ contains the path of shortest bottleneck length between every pair of points.

---

3Of course one could build an entirely new MST from scratch. Can one do better?
Exercise 13.7. Let $G = (V, E)$ be an undirected graph with distinct nonnegative edge weights $w : E \rightarrow \mathbb{R}$. For a spanning tree $T$, we say that the bottleneck weight of $T$ is the maximum weight edge in $T$, $\max_{e \in T} w(e)$. Consider the problem of computing the minimum

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.)

Exercise 13.8. Recall the notion of “bottleneck shortest paths” from exercise 13.6. Design and analyze a $O(m + n)$-time algorithm for computing the shortest bottleneck $(s, t)$-path.

Exercise 13.9. 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.

Exercise 13.10. 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 $F \subset E - e$ that (a) has size $|F| \leq k$ and (b) spans $e$ (and doesn’t include $e$),

$$w(e) < \max_{f \in F} w(f).$$

Prove that $e$ is $k$-great iff every $k$-MWF contains $e$.

2. Design and analyze an algorithm for computing the $k$-MWF.

---

4Here’s step 1: compute the median edge weight in $O(m)$ time.

5It may be helpful to obtain a running time generically as a function of $k$, and then optimize $k$. 

210
Exercise 13.11. For $p > 0$, the $L_p$-norm of a vector $x \in \mathbb{R}^k$ is defined by

$$\|x\|_p \overset{\text{def}}{=} \left( \sum_{i=1}^{k} |x_i|^p \right)^{1/p}.$$  

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

$$\|x\|_{\infty} \overset{\text{def}}{=} \max_i |x_i|.$$  

Let $G = (V, E)$ be an undirected graph with real-valued edge lengths $\ell : E \rightarrow \mathbb{R}$. 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 $w$,

$$\|w\|_p \overset{\text{def}}{=} \left( \sum_{i=1}^{k} |\ell(e_i)|^p \right)^{1/p}.$$  

Analogously we define the $L_0$ and $L_{\infty}$ 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

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 $L_{\infty}$-distance from $s$ to $t$.
Chapter 14

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

$$\text{find a maximum cardinality packing of } (s,t)\text{-paths}.$$ 

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

$$\text{find the minimum cardinality } (s,t)\text{-cut.}$$

Both can be understood as a generalization of reachability, discussed before. Reachability was concerned with whether there is a single connection from $s$ to $t$. Both the maximum flow and minimum cut problems measure the strength of the connection from $s$ to $t$.

\(^1\)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.
We are interested in these questions both algorithmically and (graph-)structurally. Algorithmically the problems are highly non-trivial, as there are exponentially many possible paths from \( s \) to \( t \) to take into account. It is not obvious that there is a polynomial time algorithm for either problem.

### 14.1 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.

\[
\text{(max (s,t)-path packing)} \leq \text{(min (s,t)-cut)}.
\]  

Indeed, let \( P \) denote a path packing, and let \( C \subseteq E \) be an \((s,t)\)-cut. We have

\[
|P| = \sum_{p \in P} 1 \leq \sum_{p \in P} |p \cap C| \leq |C|.
\]

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.

![Conceptual sketch of packing and covering paths](image)

The inequality (14.1) inspires some basic questions. Is the inequality ever equal? Is the inequality ever strict?

### 14.1.1 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 it to a growing collection.
14. Packing and covering paths

14.1. Duality

Kent Quanrud
Spring 2022

\text{greedy-path-packing}(G = (V, E), s, t)

1. \( P \leftarrow \emptyset \).

2. While there is a path \( p \) in \( G \) from \( s \) to \( t \):
   \( \quad \text{// BFS or DFS} \)
   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.

\text{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 \textit{locally optimal} solutions. \text{greedy-path-packing} terminates when no more paths can be added to the current solution. \text{greedy-cut} terminates when no more edges can be removed from the current solution. That is:

\textbf{Lemma 14.1.} \text{greedy-path-packing} returns a \textit{maximal} \((s, t)\)-\textit{path packing} \( P \), and \text{greedy-cut} returns a \textit{minimal} \((s, t)\)-\textit{cut}.

Tautologically, we have

\[ (\text{maximal path packing}) \leq (\text{maximum path packing}) \]

and

\[ (\text{minimal } (s, t)\text{-cut}) \geq (\text{minimum } (s, t)\text{-cut}). \]

But these relations are only useful if we can strengthen the inequalities to equalities. Unfortunately, simple counter examples show that they are not equal. Consider the following graph.

214
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 counter examples that exaggerate the difference in size between a maximal and maximum path packing.

It is also easy to come up with counter examples that show that a minimal \((s, t)\)-cut is not a minimum \((s, t)\)-cut.

The point is that

\[
\text{(maximal path packing)} \neq \text{(maximum path packing)}
\]

and

\[
\text{(minimal } (s, t)\text{-cut}) \neq \text{(minimum } (s, t)\text{-cut}).
\]

This is different from our previous discussion on spanning trees (chapter 13). 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 solution in 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.
14.2 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 have 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 14.2. 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.
14. Packing and covering paths

14.2. Augmenting paths

Consider the opposite edges of a path in a graph. If we cancel out the edges, we should be allowed to take the opposite edge in subsequent paths. This leads to the following algorithm, due to Ford and Fulkerson [FF56].

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.

\[
\text{augmenting-paths}(G = (V,E), s,t) \]

/* Ford and Fulkerson [FF56] */

1. \(P \leftarrow \emptyset\)

2. While there is an \((s,t)\)-path \(p: s \leadsto t\):
   
   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. \(S \leftarrow \{\text{vertices still reachable from } s\}\).

4. Return the \((s,t)\)-path packing \(P\) and the \((s,t)\)-cut \(\partial^+(S)\).

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.

Next we will argue that augmenting-paths returns an optimum solution. To analyze augmenting-paths we use the following definitions. First, we define the residual graph of a path packing as the graph obtained by removing the paths from the graph, and adding their

\(^3\)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.
14. Packing and covering paths

14.2. Augmenting paths

reverse. The augmenting-paths algorithm maintains a residual graph of the current path-(or walk)-packing at any point in time. The key to the proof is the following observation.

Observation 14.3. Let \( S \subset V \) be any set of vertices with \( s \in S \) and \( t \notin T \). Let \( p : s \leadsto t \) be an \((s,t)\)-path. Then \( p \) has exactly one more edge for \( \partial^+ (S) \) than from \( \partial^- (S) \). In particular, the residual graph w/r/t \( p \) decreases \( |\partial^+ (S)| \) by 1 and increases \( |\partial^- (S)| \) 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 \( k = \partial^+ (S \mid G) \) number of edges of the out cut of \( S \) in the original input graph. Let us observe the changes to \( \partial^+ (S \mid G) \) over the course of the augmenting-paths algorithm. Each iteration, we increase our path packing by one, and the size of \( \partial^+ (S) \) 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 \( \partial^+ (S) \) is a minimum \((s,t)\)-cut.

We have now accomplished two things. First, we have established a mathematical connection between the maximum number of edge-disjoint paths and the minimum number of edges in a \((s,t)\)-cut. This is called Menger’s theorem.

Theorem 14.4 (Menger’s theorem). In any directed graph with integer capacities, and fixed source and sink, we have

\[
\text{(maximum } (s,t)\text{-path packing)} = \text{(minimum } (s,t)\text{-cut)}.
\]

In particular, there exists an integral maximum flow.

Second, we have obtained an algorithm computing the maximum \((s,t)\)-path packing and minimum \((s,t)\)-cut with the following guarantee.
14. Packing and covering paths

14.3 Multicommodity flow

Theorem 14.5 ([FF56]). Let $G = (V, E)$ be a directed graph with $m$ edges, and let $s, t \in V$ be distinct vertices. Let $\lambda$ be the size of the max-flow/min-cut from $s$ to $t$. Then augmenting-paths($G, s, t$) produces both a maximum 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, $\lambda$ is at most $m$, so augmenting-paths is polynomial time. However, the next chapter will consider the generalization where edges have capacities $c : E \rightarrow \mathbb{R}_{>0}$, that allow us to reuse an edge $e$ up to $c(e)$ times. Here $\lambda$ may be exponentially large, and this running time is no longer satisfactory.

14.3 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 14.6. 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 $x_j = \text{true}$ and the other will encode the assignment $x_j = \text{false}$.

   (a) We introduce two variables $A^-_j$ and $A^+_j$.

   (b) Let $i_1, \ldots, i_k$ be the indices of the clauses $C_i$ satisfied by $x_j = \text{true}$. For each index $i_\ell$, we create two vertices $B^-_{i_\ell,j}$ and $B^+_{i_\ell,j}$. We add edges to form a path from $A^-_j$ to $A^+_j$. We add edges from $A^-_j$ to $B^-_{i_\ell,j}$, from $B^-_{i_\ell,j}$ to $B^+_{i_\ell,j}$ for each $i_\ell$, and from $B^+_{i_\ell,j}$ to $B^-_{i_{\ell+1},j}$ for each $\ell < k$. 

219
14. Packing and covering paths

14.3. Multicommodity flow

Kent Quanrud
Spring 2022

(c) We similarly construct an \((A_j^-, A_j^+)-path\) for the clauses satisfied by \(x_j = \text{false}\).

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 \(x_j\).

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. 14.1, 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 “\(x_j = \text{true}\)” path or the “\(x_j = \text{false}\)” 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.
14. Packing and covering paths

14.4. Exercises

Lecture materials and comments. Click on the links below for the following files:

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

14.4 Exercises

Exercise 14.1. 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 14.2. 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 14.3. 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 computing a single $(a,b,c)$-path (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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.
Chapter 15

Network flow

Recall the edge-disjoint path problem in directed graphs. The input consists of a directed graph $G = (V,E)$ and two vertices $s,t$. The goal is to compute the maximum number of edge-disjoint paths from $s$ to $t$. We can think of this as a packing problem, where we are “packing” $(s,t)$-paths into the edges.

We also discussed a dual covering problem, called the minimum $(s,t)$-cut problem. Here the goal is to compute the minimum cardinality set of edges $C \subseteq E$ whose removal leaves no paths from $s$ to $t$. Such an edge set $C$ is called an $(s,t)$-cut. We can think of this as a covering problem where an edge $e$ “covers” any path containing it, and the goal is to take the minimum set of edges covering any path.

It is easy to see that the maximum number of edge-disjoint $(s,t)$-paths is at most the minimum number of edges in an $(s,t)$-cut. After all, the $(s,t)$-cut contains at least one edge from every $(s,t)$-path, and in a path packing, the edges are forced to be disjoint. The general type of inequality, $(\max \ldots) \leq (\min \ldots)$, is universal to packing and covering problems. The interesting question is understanding if and when the values are equal.

The most natural algorithm for edge-disjoint paths, which greedily finds and removes $(s,t)$-paths from the graph, does not work. (This is in contrast to spanning trees where greedy algorithms do work.) However, an extension of the greedy algorithm does work: the key trick was to update a “residual graph” that included the reverse of each edge already taken in a selected path. This is commonly known as the augmenting-paths algorithm or the Ford-Fulkerson algorithm [FF56]. The augmenting path algorithm not only computed a maximum path packing, but the residual graph at termination also revealed a minimum $(s,t)$-cut. In fact the proof compares the $(s,t)$-path-packing and the $(s,t)$-cut that the algorithm produces and argues that they have the same size. Coupled with the “easy” $(\max \ldots) \leq (\min \ldots)$ inequality mentioned above, this simultaneously
certifies both the \((s,t)\)-path packing and the \((s,t)\)-cut to be optimal. Consequently we have the following equality between the optimum values of our dual problems (for any graph):

\[
\text{(max \# edge-disjoint} \ (s,t)\text{-paths)} \ = \ \text{(min \# edges in any} \ (s,t)\text{-cut)}.
\]

The algorithm itself takes \(O(m\lambda)\) time where \(\lambda\) is the optimum value; in particular it is at most \(m\).

### 15.1 Edge capacities, fractional path packings, and capacitated cuts

We continue our discussion by extending the graph with positive edge capacities \(c : E \to \mathbb{R}_{>0}\). The interpretation from a path-packing point of view is that now we want to select the maximum quantity of \((s,t)\)-paths such that the total quantity of paths using a particular edge \(e\). To formalize this, 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 \(e\),

\[
\sum_{p \in \mathcal{P}_{s,t} : e \in p} x_p \leq c(e). \quad (15.1)
\]

A fractional path packing \(x\) is said to be \textbf{integral} if every weight \(x_p\) is an integer. The \textbf{maximum (capacitated) path packing} problem is to compute the path packing \(x\) (satisfying (15.1) for all \(e\)) of maximum total size:

\[
\text{maximize} \quad \sum_{p \in \mathcal{P}_{s,t}} x_p \quad \text{over} \ x : \mathcal{P}_{s,t} \to \mathbb{R}_{\geq 0}
\]

\[
\text{s.t.} \quad \sum_{p \in \mathcal{P}_{s,t} : e \in p} x_p \leq c(e) \quad \text{for all} \ e. \quad \text{(Max-Paths)}
\]

(Max-Paths) can still be understood a packing problem, but now it is a \textit{capacitated} packing problem. Below we let (Max-Paths) denote the optimum value of the path packing problem formulated above.
Edge capacities generalize our minimum \((s, t)\)-cut problem from minimizing the number of edges in the cut, to minimizing the total capacity of the edges in the cut. This is called the minimum (capacitated) \((s, t)\)-cut. In a sense, we are interpreting the capacities as costs, in which case we want to compute the minimum \((s, t)\)-cut. One precise formulation is as follows:

\[
\text{minimize } \sum_{e \in C} c(e) \text{ over all } C \subseteq E \quad \text{(Min-Cut)}
\]

\[
s.t. |p \cap C| \geq 1 \text{ for all } p \in \mathcal{P}_{st}.
\]

This is still a covering problem – we need a set of edges that “cover” all the \((s, t)\)-paths – but the capacities are now costs for taking the edges.

Now how do the capacitated versions of these problem relate to the uncapacitated versions previously discussed?

**Integer capacities.** Suppose each edge capacity \(c(e)\) as an integer. Then mathematically the problem is equivalent to the uncapacitated problem where we make \(c(e)\) copies of every edge \(e\). Augmenting paths gives a \(O(m\lambda)\) time algorithm although this is no longer polynomial time because \(\lambda\) may be very large. We also have that the equality between the maximum number of edge-disjoint paths and the minimum number of edges in any \((s, t)\)-cut. This translates to \((\text{Max-Paths}) = (\text{Min-Cut})\) for integer capacities. Moreover, the augmenting paths algorithm tells us that the maximum path packing can be attained by an integral path packing.

**Rational capacities.** Now suppose each edge capacity \(c(e)\) is rational valued. We can reduce to the integer capacitated case above by multiplying all the capacities by a large common denominator \(D \in \mathbb{N}\). Applying our observations above for the scaled up, integer capacities, and then dividing everything by \(D\), gives us \((\text{Max-Paths}) = (\text{Min-Cut})\) for rational-valued capacities.

**Real-valued capacities.** Lastly, suppose each edge capacity \(c(e)\) was simply real-valued. We cannot simply clear denominators and reduce to the integer case. That said, we can still approximate \(c\) with rational capacities, which we denote \(\tilde{c}\), and apply our analysis. For fixed \(\epsilon > 0\), we can choose rational capacities \(c(e) > 0\) such that \(|c(e) - \tilde{c}(e)| \leq \epsilon\) for all \(e\). The “\((\text{Max-Paths}) = (\text{Min-Cut})\)” theorem for the rational capacities \(\tilde{c}\) then translates to

\[
|((\text{Max-Paths}) - (\text{Min-Cut}))| \leq O(\epsilon m)
\]

for the (original) real-valued capacities \(c\). Taking the limit \(\epsilon \to 0\) gives us \((\text{Max-Paths}) = (\text{Min-Cut})\) for real-valued capacities as well.

We summarize our discussion in the following.
15. Network flow
15.2. Network flow.

Theorem 15.1. 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.

15.2 Network flow.

We now introduce a relaxation of fractional path packing that, in some sense, “forgets” there was every any path to being with. To set the scene, imagine each edge as a (one-directional) pipe where the capacity limits how much liquid can flow. Imagine one fractional unit of an $(s, t)$-path as one unit of liquid running from $s$ to $t$ along this path. If we zoom out (within this physical analogy), the path packing will look like generically mixed liquid running from $s$ to $t$. We will notice the amount of liquid along each edge, and that that amount of liquid going into a vertex is always the same as the amount of liquid going out (except at $s$ and $t$); but we won’t necessarily see the underlying paths. In fact the paths don’t really matter, from this point of view.

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 $v$:

$$\sum_{e \in \partial^-(v)} f_e = \sum_{e \in \partial^+(v)} f_e.$$

**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. In particular, if we let $\delta = \min\{f(e_1), f(e_2)\}$, then subtracting $\delta$ 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; it is entirely analogous to “uncrossing paths” in the previous chapter. $^1$

$^1$Some of the literature defines flows so that at most one of $f(e_1)$ or $f(e_2)$ to be positive at any point in time. We do not emphasize this point here, knowing that opposite flow can easily be canceled out.
Net flow at the terminals. In general, the net flow into a vertex $v$ is defined as the difference

$$\sum_{e \in \partial^-(v)} f_e - \sum_{e \in \partial^+(v)} f_e.$$

Positive net flow into $v$ means more flow goes into $v$ then leaves it; negative net flow into $v$ means more flow leaves $v$ then enters it. It is convenient to also define the net flow out of a vertex $v$ as the negation of the net flow into $v$.

Conservation of flow can then be defined as requiring that every non-terminal vertex has net flow 0. Conservation of flow implies that the negative net flow out of $s$ equals the net flow into $t$. If we imagine the flow as a physical liquid, then we can visualize a net flow out of $s$ as passing through any non-terminal vertex, and leaving the network at $t$ (where it forms a positive net flow into $t$). Since flow is conserved at all the non-terminals, the flow leaving the system at $t$ – the net flow into $t$ – is exactly equal to the net flow out of $s$. For a more formal proof, we have

$$0 = \sum_{e \in f_e} f_e - f_e$$

$$\equiv \sum_{v \in V} \sum_{e \in \partial^-(v)} f_e - \sum_{e \in \partial^+(v)} f_e$$

$$\equiv \sum_{e \in \partial^-(s)} f_e - \sum_{e \in \partial^+(s)} f_e + \sum_{e \in \partial^+(t)} f_e - \sum_{e \in \partial^-(t)} f_e$$

$$= (\text{net flow into } s) + (\text{net flow into } t)$$

Here (a) distributes each $f_e - f_e$ 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. Now rearranging the equality obtained above gives

$$(\text{net flow out of } s) = -(\text{net flow into } s) = (\text{net flow into } t),$$

as desired.

The net flow out of $s$; or 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. A compact formulation of the problem (recapping everything introduced above) is given by

$$\text{maximize } \sum_{e \in \partial^+(s)} f(e) - \sum_{e \in \partial^-(s)} f(e) \text{ over } f : E \rightarrow \mathbb{R}_{\geq 0}$$

$$\text{s.t. } \sum_{e \in \partial^+(v)} f(e) = \sum_{e \in \partial^-(v)} f(e) \text{ for all } v \in V \setminus \{s,t\}, \quad \text{(MaxFlow)}$$

$$f(e) \leq c(e) \text{ for all } e \in E.$$
The objective describes the net flow out of $s$; i.e., the size of $f$. The first set of constraints describe conservation of flow, and the second set of constraints are the edge capacity constraints. Note that all the constraints are objects are either mathematical equalities or inequalities; as such it is an example of something called a *linear program*. (More on this later; we mention in passing that *(Max-Paths)* also described a linear program, with exponentially many variables.)

**Max flow vs max path packing.** As alluded to above, the maximum $(s,t)$-flow problem has strong connection to the maximum $(s,t)$-path packing problem. Indeed, given an $(s,t)$-path packing $x : \mathcal{P}_{st} \rightarrow \mathbb{R}_{\geq 0}$, consider the $(s,t)$-flow $f$ where the flow on an edge equals the total quantity of paths using that edge:

$$f_e = \sum_{p \in \mathcal{P}_{st} : e \in p} x_p.$$

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) $\leq$ (Max-Flow).*

**Max flow vs Min cut.** Now consider any $(s,t)$-cut of the form $C = \partial^+(A)$, where $A \subseteq V$ with $s \in A$ and $t \notin A$. We have

$$|f| = \text{(net flow out of $s$)} \overset{(i)}{=} \sum_{v \in A} \text{(net flow out of $v$)}$$

$$= \sum_{v \in A} \sum_{e \in \partial^+(v)} f(e) - \sum_{e \in \partial^-(v)} f(e) \overset{(d)}{=} \sum_{e \in \partial^+(A)} f(e) - \sum_{e \in \partial^-(A)} f(e).$$

Here (c) observes that all $v \in A - s$ have net flow 0. (d) observes that every edge with both endpoints in $A$ appear 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 $A$, so to speak. The net flow out of $A$ is always bounded above by the capacity of $\partial^+(A)$, as

$$\sum_{e \in \partial^+(A)} f(e) \leq \sum_{e \in \partial^+(A)} c(e).$$

Here (e) is by nonnegativity of flow, and (f) is because the flow obeys edge capacities. All put together, we have

$$|f| \leq \sum_{e \in \partial^+(A)} c(e).$$
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

$$(\text{Max-Flow}) \leq (\text{Min-Cut}).$$

To recap, we have now established that

$$(\text{Max-Paths}) \leq (\text{Max-Flow}) \leq (\text{Min-Cut}).$$

But we already know that $(\text{Max-Paths}) = (\text{Min-Cut})$. So “sandwiching” inequality above into one great equality, we have the following theorem.

**Theorem 15.2.** Let $G = (V, E)$ be a directed graph with positive edge capacities and $s, t \in V$. Let $(\text{Max-Paths})$, $(\text{Min-Cut})$, and $(\text{Max-Flow})$ denote the optimum values of the maximum $(s,t)$-path packing, the minimum $(s,t)$-cut, and the maximum $(s,t)$-flow, respectively. Then

$$(\text{Max-Paths}) = (\text{Max-Flow}) = (\text{Min-Cut}).$$

Moreover, if the capacities are integer, then there is an integral maximum $(s,t)$-path packing, and an integral maximum $(s,t)$-flow.

### 15.3 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. This gives us a polynomial time algorithm for maximum flow with unit capacities. However it does not scale with integral capacities as it requires a number of iterations equal to the total flow.

#### 15.3.1 Residual graphs

Before proceeding to an algorithm for flow, it is helpful to extend the notion of residual graphs, previously defined for path packings, to flows. The following approach is exactly as one would expect.

Let $e \in E$ be a fixed flow, and suppose we have a flow vector $f$ that sends some flow along $e$. Note that $0 \leq f(e) \leq c(e)$ by definition.
Observe that we could still $c(e) - f(e)$ more units of flow in the same direction while still satisfying the capacity constraints. But can also effectively route $c(e)$ 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.

This generalized notion of flow allows us to extend the augmenting paths algorithm as follows. Here we make one change that’s only natural: whenever we find an augmenting path, we route as much flow as possible subject to the conservation constraints.

\[
\text{augmenting-paths}(G = (V, E), c : E \rightarrow \mathbb{R}_{\geq 0}, s, t)
\]

// Edmonds and Karp [EK72]
1. $f \leftarrow \emptyset$
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 $\gamma$ be the minimum (residual) capacity of any edge in $p$
   C. add $\gamma$ 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 $S$

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.

### 15.3.2 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\((G = (V, E), \epsilon : E \rightarrow \mathbb{R}_{>0}, s, t)\)

// Edmonds and Karp [EK72]
1. \(f \leftarrow 0\)
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 \(\gamma\) be the minimum (residual) capacity of any edge in \(p\)
   C. add \(\gamma\) 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 \(S\)

The key lemma is as follows.

**Lemma 15.3.** For all \(v \in V\), the distance from \(s\) to \(v\) in the residual graph never decreases.

Let us assume first assume lemma 15.3 and see why it useful in deriving a polynomial time algorithm.

Consider a directed edge \(e = (u, v) \in E\).

1. Whenever \(e\) is removed from the residual graph, \(e\) must have been in the shortest \((s, t)\)-path.

   ![Diagram 1](image1.png)

   In particular, we must have \(\ell(v) = \ell(u) + 1\).

2. When \(e = (u, v)\) is added back into the residual graph, the opposite edge \((v, u)\) was in the preceding shortest \((s, t)\)-path.

   ![Diagram 2](image2.png)

   Then we must have \(\ell(u) = \ell(v) + 1\).
If $\ell(v) = \ell(u) + 1$ at some point, and then later $\ell(u) = \ell(v) = 1$, then $\ell(u)$ and $\ell(v)$ must have switched order. Meanwhile lemma 15.3 asserts that the distance levels never decrease. This implies that $\ell(u)$ must have increased by two. Since the maximum distance is $n$, 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. The maximality ensures that some edge along the path will be removed from the residual graph. Since edges can be added and deleted at most $O(n)$ times, this implies that shortest-augmenting-paths will terminate with $O(mn)$ iterations. Each iteration takes $O(m + n)$ time to find the shortest path and update the graph.

In conclusion, if we assume lemma 15.3 is true, then we have the following.

**Theorem 15.4** ([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 15.3. We restate the claim for the reader’s convenience.

**Lemma 15.3.** For all $v \in V$, the distance 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 $\alpha$ be the level of $v$ before augmentation, and let $\beta$ 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 previous vertex in the shortest $(s,v)$ path after the augmentation.

\[ \begin{align*}
\text{BEFORE} & \\
(s) & \xrightarrow{\beta-1} u \xrightarrow{} v \\
(\text{by induction}) & \quad \beta - 1 \\
\text{AFTER} & \\
(s) & \xrightarrow{\beta-1} u \xrightarrow{} v
\end{align*} \]

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. ■

Note that the shortest augmenting path algorithm takes \(O(m^2n)\) time, no matter the magnitude of the capacities, assuming that basic arithmetic can be done in \(O(1)\) time. Such an algorithm is sometimes called strongly polynomial time. We point out that there have been many improvements in maximum flow and it remains an active area of research. For simplicity we restrict our attention to algorithms we have analyzed together.

15.4 Minimum cost metrics, linear programming and LP duality

15.4.1 Fractional minimum 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 \((s,t)\)-paths. An edge cost \(c(e)\) are now interpreted as the cost of one unit \(y(e)\). All put together, the fractional relaxation of minimum cost \((s,t)\)-cut is given by the following problem:

\[
\text{minimize } \sum_{e \in E} c(e)y_e \text{ over } y : E \to \mathbb{R}_{\geq 0} \text{ s.t. } \sum_{e \in p} y_e \geq 1 \text{ for all } p \in \mathcal{P}_{s,t}. \tag{15.2}
\]

Note that any \((s,t)\)-cut \(C\) corresponds to a \([0,1]^E\)-indicator vector that is feasible to (15.2).

(15.2) 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 \(y\), is \(\geq 1\); we can verify this constraint by computing the shortest \((s,t)\)-path w/r/t \(y\). For this reason, (15.2) is called a separable LP, where the shortest path computation provides a separation oracle.

An equivalent formulation of (15.2), then, is as follows.

\[
\text{Find the minimum cost set of edge lengths } y : E \to \mathbb{R}_{\geq 0} \text{ subject to } s \text{ and } t \text{ having distance 1 in the shortest path metric induced by } y.
\]

This problem, besides being a relaxation of \((s,t)\)-cut, is a very natural problem in its own right. For this reason, and to help distinguish the continuous nature of (15.2) for the discrete min-cut problem, we will also refer to the fractional min-cut problem as the \((s,t)\)-minimum cost metric problem. We will write \((\text{Min-Metric})\) to denote both the optimization problem and the value of the optimization problem formulated in (15.2) above.

### 15.4.2 Linear programs

The fractional versions of the \((s,t)\)-path packing and cut problem described above are examples of linear programs, a special class of mathematical optimization problems which we now introduce.

Linear programs (LP’s) 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

\[
\begin{align*}
\min/\max \langle b, x \rangle &= \sum_{j=1}^n b_j x_j \text{ over } x \in \mathbb{R}^n \\
\text{s.t. } A_1 x &\leq c_1, \quad A_2 x = c_2, \quad \text{and } A_3 x \geq c_3.
\end{align*}
\]

where \(A_1, A_2, A_3\) are matrices and \(b, c_1, c_2, c_3\) are vectors.

Clearly, linear programs are useful for modeling real problems where we seek continuous solutions. There are also many different uses of LP’s for understanding and solving discrete problems. A powerful feature of LP’s is that they are polynomial time solvable \(^2\), and conceptually it is easy to interact with these solvers as a black box. Moreover, real-world software for LP’s is well-developed and reliable in practice.

We now introduce two canonical classes of LP’s that capture most combinatorial problems.

\(^2\)More precisely, they are weakly polynomial time solvable, meaning the running times are polynomial in the bit complexity of the input.
Packing LPs. A packing LP is a linear program of the form
\[
\max \langle b, x \rangle \text{ over } x \in \mathbb{R}^n_{\geq 0} \text{ s.t. } Ax \leq c.
\]
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 \(\text{Opt}(P)\) 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. \(b = 1_{s,t},\) 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 otherwise to solve the LP, as we will see.

Covering LPs. A covering LP is a linear program of the form
\[
\min \langle c, y \rangle \text{ over } y \in \mathbb{R}^m_{\geq 0} \text{ s.t. } A^T y \geq b,
\]
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 \(\text{Opt}(C)\) 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.
1. \(c \in \mathbb{R}^E_{\geq 0}\) is the edge costs.
2. \(b = 1_{\mathcal{P}_{s,t}}\) 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.

\(^3\)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 \((C)\).
15.4.3 LP duality.

LP duality is about the relationship between the linear programs (P) and (C), 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 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

\[
\langle b, x \rangle \overset{\text{(a)}}{\leq} \langle A^T y, x \rangle \overset{\text{(b)}}{=} \langle y, Ax \rangle \overset{\text{(c)}}{\leq} \langle y, c \rangle.
\]

Here (a) is because \( x \geq 0 \) and \( A^T y \geq b \). (b) is by definition of the transpose. (c) is because \( y \geq 0 \) and \( Ax \leq c \). Thus, for a packing problem (P) and a covering problem (C) linked by duality, we have

\[ \text{Opt}(P) \leq \text{Opt}(C). \]

If this argument seems familiar, it is because we just saw it for packing and covering paths in section 14.1 above.

We ask the same question for packing and covering LP's as we did for packing and covering paths. When, if ever, is \( \text{Opt}(P) = \text{Opt}(C) \)? The all-important LP duality theorem (here restricted to packing and covering problems) states that in fact they are always equal.

**Theorem 15.5** (LP Duality for packing and covering). \( \text{Opt}(P) = \text{Opt}(C) \).

We note that theorem 28.3 holds even if \( A, b, \) and \( c \) have negative coefficients.

LP duality has important consequences for many combinatorial problems of interest – starting with max flow in the present discussion. Recall that (Max-Flow) is a packing LP and (Min-Metric) is a covering LP. Moreover, they are dual to one another. The LP duality theorem then tells us that (Max-Flow) = (Min-Metric).

Now, as (Min-Metric) is a linear relaxation of (Min-Metric), we automatically have (Min-Metric) \leq (Min-Cut). But we also know that both values are equal to (Max-Flow), giving us (Min-Metric) = (Min-Cut). That is, allowing for “fractional” cuts does not improve the optimum solution. In conclusion, we have the following.

**Corollary 15.6.** (Max-Flow) = (Min-Metric) = (Min-Cut); moreover, there is always an integral optimum solution to (Min-Metric).

We will (probably) not have time to really explore the many use cases and insights shed by linear programming, nor will we discuss algorithms to solve LP’s, nor the proof of LP duality. (Some of these discussions would require more mathematical prerequisites.) Nonetheless it is good to be aware of their central importance. One should look at maximum flow as a canonical example of a linear program that is particularly easy to understand.
Additional notes and references

This chapter overlaps with [KT06, Chapter 7] and [Eri19, Chapter 10]. The algorithm derived in exercise 15.4 is also from [EK72].

Lecture materials and comments. Click on the links below for the following files:

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

Exercises

Exercise 15.1. 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 15.2. 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 of \(f\), 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 15.3. Let \(G = (V, E)\) be a directed graph with integer edge capacities \(c : E \rightarrow \mathbb{N}\). 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 there first color may be any color; all that matters is that we
keep cycle though the colors in any order. 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.)

**Exercise 15.4.** This exercise develops a \( O\left( (m^2 + mn \log(n)) \log(\lambda) \right) \)-time algorithm for maximum \((s, t)\)-flow and builds on ideas from exercise 15.2.

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 \( \lambda / m \).

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 \( \lambda \) denotes the value of the maximum flow.⁵⁶ (The algorithm does not know the true value of \( \lambda \) \textit{a priori}.)

---

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.

⁵This is polynomial with respect to the bit complexity of the input.

⁶A 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 16

Applications of flows and cuts

There are 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.

16.1 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 reflects how much data can be carried through each cable. The vertex capacities model how much data can be passed through each router.

### 16.2 Bipartite matching

**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.

### 16.3 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 $y$.

![Diagram showing dependencies between projects]

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 $x_{i+1}$ and 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 16.3.

### 16.4 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.

<table>
<thead>
<tr>
<th>Team</th>
<th>Wins</th>
<th>Games Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>92</td>
<td>2</td>
</tr>
<tr>
<td>Baltimore</td>
<td>91</td>
<td>3</td>
</tr>
<tr>
<td>Toronto</td>
<td>91</td>
<td>3</td>
</tr>
<tr>
<td>Boston</td>
<td>90</td>
<td>2</td>
</tr>
</tbody>
</table>

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:

\[
\begin{array}{cccc}
\text{NewYork} & \text{Baltimore} & \text{Toronto} & \text{Boston} \\
\times & 1 & 1 & 0 \\
1 & \times & 1 & 1 \\
1 & 1 & \times & 1 \\
0 & 1 & 1 & \times \\
\end{array}
\]

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.

### 16.5 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 to encode these qualifications is via an \( \{0, 1\}^{m \times n} \) qualification matrix. Below we reproduce a \( 4 \times 4 \) 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]:

\[241\]
16. Applications of flows and cuts

16.6. Edge orientation

Kent Quanrud
Spring 2022

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 \( a_i \) 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.

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? (Here a particular individual can only be assigned to a particular job once.)

16.6 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 minimum degree is at most \( h \) (or declare that this is impossible).
16.7 Densest subgraph

Let $G = (V, E)$ be an undirected graph. The density of a graph $G$ is defined as the ratio

$$\text{density } G = \frac{|E|}{|V|}.$$

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 arises in the analysis of social networks, where a very dense subgraph can be interpreted as a tight-knit community.

16.8 Reductions to flows and cuts

We now explain how to solve each of the problems introduced above.

16.8.1 Vertex disjoint paths and vertex capacities

Recall the problem of vertex-capacitated flow from section 16.1.

**Theorem 16.1.** 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, \(v^-\) and \(v^+\). 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.

### 16.8.2 Bipartite matching

**Theorem 16.2.** The maximum cardinality matching reduces to an edge-disjoint paths problem with \(O(m + n)\) edges and \(O(n)\) vertices.

**Proof sketch.** The construction is as follows.

---

\(^1\)This is particularly convenient for computing the minimum vertex \((s,t)\)-cut, since the \(+\infty\)-capacity edges are always omitted.
1. We create a vertex $\ell_i$ for each individual $i \in [m]$, and a vertex $r_j$ for each job $j \in [n]$.

2. We add a directed edge $(\ell_i, r_j)$ whenever individual $i$ is qualified for job $j$.

3. We add a super-source vertex $s$, and a directed edge from $s$ to $\ell_i$ 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 $(s, \ell_i, r_j, t)$ 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 $\ell_i$ because $\ell_i$ has only one incoming edge. For each $j \in [n]$, 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 $(s, \ell_i)$ 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 individual $j$ is assigned at most once.

3. Each edge of the form $(\ell_i, r_j)$ 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 $\min\{m,n\}$.

### 16.8.4 Project selection

Recall the project selection problem from section 16.3.

**Theorem 16.3.** 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 to \(t\). Each such edge is given capacity equal to the (positive) cost of that project. We add a directed edge for every capacity with cost \(+\infty\).

Consider any set \(S \subset V\) that contains \(s\) and excludes \(t\), and the induced \((s, t)\)-cut \(\partial^+(S)\). 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. Consider assume the capacity of the cut is finite. All of the edges in the cut \(\partial(S)\) 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
\[
\left( \text{sum of profits of positive projects excluded from } S \right) - \left( \text{sum of costs of negative projects in } S \right).
\]

We can rewrite this as
\[
\left( \text{total sum of profits of all positive projects} \right) - \left( \text{sum of profits of all positive projects in } S \right) - \left( \text{sum of costs of negative projects in } S \right) \]
\[
= \left( \text{total sum of profits of all positive projects} \right) - \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.

16.8.5 Playoff Elimination

Recall the playoff elimination problem from section 16.4.
Theorem 16.4. 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 game. We create a directed acyclic graph with 4 layers, with the following vertices.

1. In the first layer we have a source \( s \).
2. 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 \( b \).
3. 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.
4. 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. Otherwise we declare that the home team has been eliminated.

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 outcome to all the games such that no other team has more than \( M \) wins. In one direction, given any set of
outcomes where the home team makes the playoffs, we can reassign any game the home team loses so that the home team wins. Now the home team will still make the playoffs, and also has \( M \) 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 \( G \) if and only if 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 \( t_b \) 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 \( M - w(t) \), 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 in a 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 is 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.

Below, we draw the flow network described above for the example described in section 16.4.
16.8.6 Edge orientation

Recall the edge orientation problem from section 16.6.

**Theorem 16.5.** 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 source $t$. 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 $v$. 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 16.6.** 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 16.5, 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 16.5 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.

### 16.9 Additional notes and references

We recommend [KT06] and [Eri19] for further discussions on applications of flow (including several of the applications discussed here).

**Lecture materials and comments.** Click on the links below for the following files:

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

### 16.10 Exercises

**Exercise 16.1.** Suppose we are given 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,

$$
\begin{pmatrix}
1.2 & 3.4 & 2.4 \\
3.9 & 4.0 & 2.1 \\
7.9 & 1.6 & 0.5
\end{pmatrix}
$$

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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 16.2. 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 \( R_j \subseteq [m] \) 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.
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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 16.3. Recall the project selection problem in section 16.3. Suppose we relax the problem so that the “precedence requirements” are not longer required to be acyclic. That is, let us 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 a polynomial
time algorithm would imply a polynomial time algorithm for SAT.

Exercise 16.4. Let $G = (V, E)$ be an undirected graph with positive edge weights
$w : E \rightarrow \mathbb{R}_{>0}$. 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 a polynomial
time algorithm would imply a polynomial time algorithm for SAT.

Exercise 16.5. In the playoff elimination problem (section 16.4), we asked
whether a particular team was mathematically eliminated from making the playoffs.
This allows for a perhaps that the home team would win all of its remaining
games, and allows for 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 as $k$ 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
a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 16.6. Given integer values $b : \mathbb{R}^V$, 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 a polynomial time algorithm would imply a polynomial time algorithm
for SAT.

Exercise 16.7. Recall that in chess, a rook (♖) is a piece that can move horizon-
tally 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 $4 \times 4$ chess board. Unavailable squares are marked by $\times$. Rooks are marked by $\#$.

```
X  X  X
X  X  X
X  X  X
X  X  X
```

Formally, the input consists of an $n \times n$ bit-array $A[1..n,1..n] \in \{\text{true, false}\}^{n \times n}$, where $A[i,j] = \text{true}$ indicates that square $(i,j)$ is available, and $A[i,j] = \text{false}$ 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.
Chapter 17

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 uncapacitated graphs, the maximum $(s,t)$-flow and the minimum $(s,t)$-cut can be computed in $O(m + n \lambda^2)$ time, where $\lambda$ 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 $O(m^2n)$ shortest-augmenting-paths algorithm from chapter 15. The second result improves on the augmenting paths algorithm from chapter 14. The improvement comes from connections to spanning trees (chapter 13) and this connection is interesting in its own right.

17.1 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
cuits, 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 $MF(m,n)$ denote the running time to compute max-flow in a graph with $m$ edges and $n$ vertices, then this approach would take 

$$\binom{n}{2} MF(m,n)$$

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 $n \cdot MF(m,n)$. Note that the best running time we have for $MF(m,n)$ (independent of the size of the flow) is $O(m^2n)$ via shortest augmenting paths (section 15.3.2), which gives a $O(m^2n^2)$-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. 17.1 on page 256. 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 17.2 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.** $\lambda(v_{n-1}, v_n) = \deg(v_n)$. 

255
17. Trees, flows, and cuts

17.1. Global minimum cut

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
      $\sum_{e = \{x,y\} \in E, j < i} c(e)$.

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 $\partial(S)$, where $S \subseteq \{v_1, \ldots, v_{n-2}\}$.

5. Compare $\partial(S)$ to the singleton cut $\partial(v_n)$, and return the smaller of the two cuts.

Figure 17.1: An algorithm computing the global minimum cut in $O(m(m + n \log n))$

Figure 17.2: 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 $\partial(v_n)$. 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 17.1** ([NI92b]). The global minimum cut in a weighted, undirected graph can be computed in $O(mn + n^2 \log n)$ time.

### 17.1.1 Maximum adjacency (MA) orderings.

To analyze the min-cut algorithm, we draw attention to the ordering of vertices produced in (2). Nagamochi and Ibaraki [NI08; NI92b] call this a maximum adjacency ordering, defined formally as follows.

**Definition 17.2.** 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 8.1) 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 17.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 17.3.** 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.
\textbf{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 \( v_1, \ldots, v_n \). \( \#O(m + n \log n) \) \textit{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 \( \text{MA-min-cut}(G') \) to obtain a cut of the form \( \partial(S) \), where \( S \subseteq \{v_1, \ldots, v_{n-2}\} \).
5. Compare \( \partial(S) \) to the singleton cut \( \partial(v_n) \) and return the smaller of the two cuts.

Figure 17.3: The min-cut algorithm from fig. 17.1 rewritten in terms of MA orderings.

\textbf{Lemma 17.4.} Let \( v_1, \ldots, v_n \) be a maximum adjacency ordering. Then \( \lambda(v_{n-1}, v_n) = \deg(v_n) \).

\textit{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 \( c > 0 \).

\textit{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 \( \partial(v_n) \). We need to show that \( \lambda(v_n, v_{n-1}) \geq b \).
1. Let $G_0 = G - v_n$ and let $\lambda_0$ 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 $\lambda_1$ 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 14.2). 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

$$\lambda(v_n, v_{n-2}) \overset{(a)}{=} a \geq b,$$

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

$$\lambda(v_{n-2}, v_n) \overset{(d)}{=} \lambda_1(v_{n-2}, v_n) \overset{(e)}{=} b,$$

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 $\lambda_0$ 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, $c = 0$. ■
17.2 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 \( G' \) if 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.

17.2.1 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 \( s, t \in V \),

\[
\lambda_H(s,t) \geq \min\{k,\lambda_G(s,t)\}.
\] (17.1)

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.
17. Trees, flows, and cuts
17.2. Sparsification and faster augmenting paths

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 (17.1).

**Theorem 17.5** ([Ni92a]). Let \(s, t \in V\) have connectivity \(\ell\) in \(G\). Then for all \(k \leq \ell\), \(s\) and \(t\) have connectivity \(\geq k\) in \(H_k\).

The key observation to proving theorem 17.5 is the following.

**Lemma 17.6.** Let \(C = \partial(S)\) 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 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 17.5. Let \( s, t \in V \) have connectivity \( \ell \) and let \( k \leq \ell \). Every \( \{s, t\} \)-cut \( C = \partial(S) \) has at least \( \ell \) edges, and by lemma 17.6, \( |C \cap H_k| \geq k \), as desired.

Theorem 17.5 gives us our first nontrivial sparsifier. With \( <k\ell \) edges, we can preserve all connectivities up to size \( k \). Don’t be fooled by its simplicity! This algorithm has inspired many more ideas.

### 17.2.2 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 \( F_1, F_2, \ldots \) 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 13.8, 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 17.7.** A maximal forest packing can be computed in \( O(m) \) time.

We will prove this at the end, in section 17.3.

### 17.2.3 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 \( \lambda \); the augmenting-paths algorithm takes \( O(m\lambda) \) time to compute the maximum flow. But we know that the first \( \lambda \) forests contain all the edges necessary to compute the maximum flow. If we knew the value of \( \lambda \) \textit{a priori}, then we could obtain a running time of \( O(m + n\lambda^2) \) by first computing the greedy forest packing in \( O(m) \) time, and then running Ford-Fulkerson in \( H_k \).

Of course we do not know \( \lambda \) \textit{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 \( i \)th iteration, we add the \( i \)th forest \( F_i \) to \( H \) before searching for the next path. If we do not find an augmenting path in the \( i \)th, 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 $i$th iteration occurs within the first $i$ forests, which has only $O(ni)$ edges. Consequently the $i$th iteration takes $O(ni)$ time. Summing over all $i$, we obtain the following.

**Theorem 17.8.** 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 $\lambda$ denotes the value of the maximum flow / minimum cut.

### 17.3 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 17.2. 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**

1. Let $v_1, \ldots, v_n$ be an MA-ordering of $G$.
2. $F_1, F_2, \ldots \leftarrow \emptyset$
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 $S \subseteq V$ and a vertex $v \in V \setminus S$, we denote the number of edges from $v$ to $S$ by

$$\deg(v \mid S) \triangleq (#\text{ edges from } v \text{ to } S)$$
17.3.1 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.

\[ \text{MA-forest}(G = (V, E), \text{ MA ordering } v_1, \ldots, v_n \text{ of } V) \]

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.

\[ \text{MA-forest}(G = (V, E), \text{ MA ordering } v_1, \ldots, v_n \text{ of } V) \]

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 \( v_1, \ldots, v_n \) be a fixed MA-ordering of \( G \). For each \( i \), let \( S_i = \{v_1, \ldots, v_i\} \). Let \( F = \text{MA-forest}(G, v_1, \ldots, v_n) \) be the forest returned by \( \text{MA-forest} \).

**Lemma 17.9.** \( F \) 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 cycle, 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 \( v_1, \ldots, v_n \) 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 17.10.** Let \( G' = G \setminus F \) be the graph obtained by removing \( F \). Then \( v_1, \ldots, v_n \) 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 \( \deg'(v_j \mid S_i) = \deg(v_j \mid S_{i-1}) - 1 \), 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.

### 17.3.2 Recursively packing forests

?? 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.

\[
\text{recursive-MA-forests}(G = (V,E), \text{MA-ordering } v_1, \ldots, v_n \text{ of } V)
\]

1. if \( E = \emptyset \) then return \( \emptyset \)
2. \( F_1 \leftarrow \text{MA-forest}(G, v_1, \ldots, v_n) \)
3. \( G_1 \leftarrow G \setminus F_1 \)
4. \( F_2, \ldots, F_k \leftarrow \text{recursive-MA-forests}(G, v_1, \ldots, v_n) \)
5. return \( F_1, \ldots, F_k \).

**Lemma 17.11.** Let \( F_1 \) be a spanning forest of \( G \), and let \( F_2, \ldots, F_k \) be a greedy forest packing of \( G - F_1 \). Then \( F_1, \ldots, F_k \) is a greedy forest packing of \( G \).
17.3.3 Shortcutting the recursion

**Lemma 17.12.** Consider an edge \( e = \{v_i, v_j\} \) where \( i < j \). Then recursive-MA-forests places \( e \) in the \( k \)th tree for \( k = \deg(v_j | S_{i-1}) \).

**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 \( v_{i_\ell} \). For \( \ell = 1, \ldots, k \), \( e_\ell \) is in \( F_\ell \).

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.

17.4 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 and comments.** 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.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 with \( k \) 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 18

Matchings and vertex covers

18.1 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. 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:

$\epsilon \cap f = \emptyset$ for all distinct $\epsilon, f \in M$.

\footnote{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 equality between the sizes of a matching and a vertex cover comes easily.

**Lemma 18.1.** Let \(M\) be a matching and let \(U\) be a vertex cover. Then \(|M| \leq |U|\).

**Proof.** Indeed,

\[
|M| \leq \sum_{e \in M} |e \cap U| \leq \bigcup_{e \in M} (e \cap U) \leq |U|.
\]

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."

We now set out to address the following two questions.

1. Is there a polynomial time algorithm for minimum vertex cover, or is it \(\text{NP-hard}\)?

2. Is there a polynomial time algorithm for maximum matching, or is it \(\text{NP-hard}\)?

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)\(^2\). A vertex cover must take 2 corners / vertices. A matching can only take one side / edge.

Which leaves the first 2 questions, which are algorithmic questions about matchings and vertex cover. The fact that the maximum matching and minimum vertex cover can have different size means that it is possible for one to be polynomial time and the other to be NP-Hard. This was not possible for \((s,t)\)-paths and cuts - since the quantities are equal, an algorithm that can identify the size of the maximum \((s,t)\)-path packing automatically can identify the size of the minimum \((s,t)\)-cut.

### 18.2 Vertex cover

We start with vertex cover.

**Theorem 18.2.** 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 \(S \subseteq V\), we denote its complement by \(\overline{S} \defeq V \setminus S\). We claim that \(S\) is an independent set iff \(\overline{S}\) is a vertex cover. If so, then finding the minimum vertex cover is equivalent to finding the maximum

\(^2\)That is, a graph with three vertices, with an edge between every pair.
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 $\overline{S}$. As this holds for all $e \in E$, $\overline{S}$ is a vertex cover.

Conversely, suppose $S \subset V$ is a vertex cover. Consider any two vertices $u, v \in \overline{S}$. If $u$ and $v$ were endpoints of an edge $e$, then $S$ would not cover $e$. Thus $\overline{S}$ is an independent set. ■

18.3 General matching

We now turn to the packing side of our primal-dual setup, matchings.

18.3.1 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 5 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 18.3.** Let \(B\) be a maximal matching and let \(A\) be a maximum matching. Then \(|B| \geq |A|/2\).

### 18.3.2 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 18.4.** A matching \(M\) is a maximum matching iff there are no augmenting paths.
18.3.3 Augmenting path algorithms

We now turn to matchings in general graphs. Lemma 18.4 suggests the following augmenting path algorithm similar to previous algorithms.

\begin{verbatim}
generic-augmenting-paths(G = (V,E))
1. M ← ∅
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.
\end{verbatim}

Certainly this algorithm would work – if it were implementable. Of the steps above, (2.A) is actually trickier than it looks. 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 18.5.** 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 edges.

**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.

![Image of a contracted graph]

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 18.6.** 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.

![Example](image)

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 18.7. 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 18.8. Let $W = (v_1, \ldots, v_\ell)$ be a shortest alternating walk such that $v_1$ and $v_\ell$ are both free. Then either (a) $W$ is an augmenting path or (b) $(v_1, \ldots, v_k)$ 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, $(v_1, \ldots, v_{k-1})$ is a path, and $(v_i, \ldots, v_k)$ is a cycle. If $(v_i, \ldots, v_k)$ is an even length cycle, then we could remove it from $W$ and obtain a shorter walk, a contradiction. So $(v_i, \ldots, v_k)$ is an odd length cycle. ■
We put everything together to obtain the following algorithm of Edmonds [Edm65].

\textbf{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.

\textbf{Theorem 18.9 ([Edm65]).} The maximum cardinality matching can be computed in \(O(mn^2)\) time.

\section*{18.4 Additional notes and references}

We also recommend section 5.2 in [Sch], lecture notes by Michel Goemans\(^3\), lecture notes by Chandra Chekuri\(^4\), and slides by Bob Tarjan\(^5\).

A 1991 interview of Jack Edmonds, in which he recounts his discovery of the blossom algorithm, can be found in [RSLM91]\(^6\). A 2015 issue of OPTIMA\(^7\) celebrated the 50th anniversary of the blossom algorithm with recent related results and includes another retrospective interview with Edmonds.

\(^3\)http://math.mit.edu/~goemans/18453S17/matching-nonbip-notes.pdf

\(^4\)https://courses.engr.illinois.edu/cs598csc/sp2010/Lectures/Lecture7.pdf


\(^7\)http://www.mathopt.org/Optima-Issues/optima97.pdf
Lecture materials and comments. Click on the links below for the following files:

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

18.5 Exercises

Exercise 18.1. 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 design and analyze a polynomial time algorithm, or show that a polynomial time algorithm implies a polynomial time algorithm for SAT.
Chapter 19

Lazy data structures

19.1 As easy as 1, 2, 3

We have all coded the line

```c
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:

```
1111111111111111111111111111111
```

we flip 32 bits! 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 `k`th bit flips every `2^{k-1}` times. So the number of bits flips is

$$
\sum_{k=1}^{\infty} \left\lfloor \frac{2n}{2^k} \right\rfloor \leq n \sum_{k=1}^{\left\lfloor \log_2 n \right\rfloor} \frac{1}{2^{k-1}} < 2n.
$$

\(^1\)Of course, for modern computers, 32 bits is no big deal either. But that’s besides the point.
The calculations are as if we flipped at most two bits with every increment. In the aggregate, an increment acts like a constant time operation; that is, 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.

19.2 Amortized analysis

In general, an operation takes $T$ amortized time if any sequence of $n$ invocations to the operation takes $nT$ total time. We can easily generalize this to multiple operations. Suppose we have some data structure that has $k$ different operations $\text{op}_1, \ldots, \text{op}_k$. Let $T_1, \ldots, T_k$ be $k$ different values. We say that each operation $\text{op}_i$ takes $T_i$ amortized time if the total running time of any sequence of operations $\text{op}_{i_1}, \text{op}_{i_2}, \ldots, \text{op}_{i_n}$ 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 binary counter is a good example of the practical tradeoffs of amortized analysis. The algorithm itself is simple and perhaps the most natural one. The analysis was more subtle, and we had to relax our insistence on worst-case bounds. Crucially, amortized analysis does not lose the rigor and robustness of worst-case analysis: we still require the algorithm to meet attain certain running times on average in the worst case over all inputs.

Amortized analysis can be tricky, since once has to “zoom out” to see why a sequence of operations is always good in the aggregate. At the end of the day, we appeal to the formal definition above to establish amortized running times. Fortunately there are a couple of well-worn techniques that sometimes offer an easier way to obtain amortized amounts. The two approaches here we discuss here are:

1. Credit schemes (a.k.a. the “banker’s method”).
2. Potential functions.

We explain each below and use binary counters as an illustrative example.

Credit schemes. The idea behind credit schemes is to have the cheap operations pays additional “credit” that can be exchanged from 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 $i$th bit gets $2^{i-1}$ units of credit in each increment. Recall also that we only flip the $i$th bit every $2^{i-1}$ increments. Consequently every time we have to flip the $i$th 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

$$\sum_{k=1}^{n} \left( \left( \text{time for } k\text{th increment} \right) + \left( \text{net increase in credit for } k\text{th increment} \right) \right) \leq O(n).$$

We also have

$$\sum_{k=1}^{n} \left( \left( \text{time for } k\text{th increment} \right) + \left( \text{net increase in credit for } k\text{th increment} \right) \right) = (\text{total time}) + \left( \text{total credit at the end} \right).$$

Combining these two equations, rearranging, and observe that the amount of credit is always nonnegative, we have

$$(\text{total time}) \leq O(n) - \left( \text{total credit at the end} \right) \leq O(n).$$

So again we conclude that increment takes $O(1)$ amortized time.

The analysis above leveraged the following facts that are are necessary for any legal “credit system”.

1. Initially, there was 0 total credits.

2. The total amount of credits is always nonnegative.

**Lemma 19.1.** Let operation $\text{op}_1, \ldots, \text{op}_k$ be $k$ operations. Suppose we have a credit scheme (obeying the requirements above) such that for each operation $\text{op}_i$, we have that

$$(\text{time for } \text{op}_i) + (\text{change in credit}) \leq T_i$$

for values $T_1, \ldots, T_k \in \mathbb{N}$. Then each $\text{op}_i$ 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 \( op_{i_1}, \ldots, op_{i_m} \). On one hand, we have
\[
\sum_{j=1}^{m} \left( \text{time for } op_{i_j} \right) + \left( \text{change in credit for } j\text{th operation} \right) \leq \sum_{j=1}^{m} T_{i_j}
\]
by assumption. On the other hand we have
\[
\sum_{j=1}^{m} \left( \text{time for } op_{i_j} \right) + \left( \text{change in credit for } j\text{th operation} \right) = (\text{total time}) + (\text{total credit at the end}).
\]
Combining these two equations, rearranging, and observe that the amount of credit is always nonnegative, we have
\[
(\text{total time}) = \left( \sum_{j=1}^{m} \left( \text{time for } op_{i_j} \right) + \left( \text{change in credit for } j\text{th operation} \right) \right) - (\text{total credit at the end}) \leq O\left( \sum_{j=1}^{m} T_{i_j} \right),
\]
as desired.

19.2.1 Potential function arguments.

The second method we present for amortized analysis is called the potential function method, and is inspired by the use of potential functions in physics. The idea is to define \( \Phi \) as a nonnegative function of the state of our data structure. It is important that \( \Phi \) is initially 0, and always nonnegative. We then try to obtain an upper bound on
\[
(\text{running time}) + (\text{change in } \Phi)
\]
for each operation, which leads to an amortized running time. Henceforth, we write \( \Delta \Phi \) as a shorthand for the change in potential \( \Phi \). We note that there are similarities between the notions of “potential” \( \Phi \) here and the “total credit” above; this connection is discussed further at the end of the section. The graph below charts a potential function \( \Phi \) going up and down over time; the red lines highlight segments where \( \Phi \) is increasing, and which increases the amortized cost; the green lines highlight segments where \( \Phi \) is decreasing, which decreases the amortized cost.
We first demonstrate the use of this method on binary counters. We define a potential function \( \Phi \) by

\[
\Phi = (\text{# bits set to 1}).
\]

Note that \( \Phi \) is 0 initially, and always nonnegative.

Consider \( \Phi \) when we call \( i++ \). Recall that the net effect of \( i++ \) is to flip \( h \) bits from 1 to 0 (for some \( h \in \mathbb{N} \)), and (at most) one bit from 0 to 1. Thus the operation takes \( O(h) \) time, and the potential \( \Phi \) decreases by \( h - 1 \). That is, \( \Delta \Phi \leq 1 - h \). The decrease \( \Delta \Phi \) by \( h \) offsets the time spent for flipping the \( h \) bits. Together we have that for any increment,

\[
(\text{running time}) + \Delta \Phi = O(1) \tag{19.1}
\]

Now consider a sequence of \( n \) increments. We have

\[
\sum_{k=1}^{n} (\text{time for } k\text{th increment}) \overset{(a)}{=} \sum_{k=1}^{n} ((\text{time for } k\text{th incr.}) + (\Delta \Phi \text{ for } k\text{th incr.})) - (\Phi \text{ at the end}) \overset{(b)}{=} O(n) - (\Phi \text{ at the end}) \overset{(c)}{\leq} O(n).
\]

Here, in (a), we observe that the sum of changes in \( \Phi \) from beginning to end is simply the value of \( \Phi \) at the end. (Note that \( \Phi = 0 \) initially). In (b), we apply (19.1) to each increment. (c) observes that \( \Phi \) is always nonnegative.

The potential-based argument above is much more general then for binary counters. The only point specific to binary counters was (19.1). The following lemma gives sufficient conditions so that for other applications, obtaining the analog of (19.1) implies the corresponding amortized bounds.

**Lemma 19.2.** Let operation \( \text{op}_1, \ldots, \text{op}_k \) be \( k \) operations, \( \Phi \) a nonnegative potential function initially set to 0, and \( T_1, \ldots, T_k \) values, such that each time we call \( \text{op}_i \), we have

\[
(\text{time for } \text{op}_i) + \Delta \Phi \leq T_i. \tag{19.2}
\]

Then each \( \text{op}_i \) 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 $\text{op}_{i_1}, \ldots, \text{op}_{i_m}$. Let $\Phi_0 = 0$ denote the initial potential, and for each $k \in \mathbb{N}$, let $\Phi_k$ denote the potential after the $k$th operation $\text{op}_{i_k}$. Let $\Delta_k \Phi = \Phi_k - \Phi_{k-1}$ denote the change in potential from the $k$th operation. We have

\[
\text{time for } \text{op}_{i_1} + \cdots + \text{time for } \text{op}_{i_m} = T_{i_1} + \cdots + T_{i_m} - \Phi_m \leq T_{i_1} + \cdots + T_{i_m},
\]

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 (19.2) to each operation. (c) is because the potential is always nonnegative.

Remark 19.3. 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.

19.3 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 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 section 19.3) 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. 19.2 for pseudocode.

```
1 2 n 2n
```

Now, it seems like common sense to allocate generously so that we don’t have to copy as often. But in terms of analyzing the performance, we clearly don’t have a $O(1)$ worst-case update time. To justify this approach, we turn to amortized analysis.

**Theorem 19.4.** 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.

We present three proofs illustrating three different methods of amortized analysis.

---

2For lack of a better word.
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 19.2: Appending an element to an array-backed list, doubling the capacity when needed.

**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

\[
O(k) + \sum_{i=1}^{\lceil \log k \rceil} \left( \text{time to double in size from } 2^i \text{ to } 2^{i+1} \right)
\]

\[
= O(k) + \sum_{i=1}^{\lceil \log k \rceil} O(2^i) = O(k).
\]

So each insertion takes \(O(1)\) amortized time.

**Proof by charging.** Whenever we append an element, we first pay an addition 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.

**Another proof by potential method.** We define a potential

\[
\Phi = \max\{n - \lfloor m/2 \rfloor, 0\},
\]

287
where \( m \) is the capacity of the array, and \( n \) is the number of elements in the list. Note that \( \Phi \) is always nonnegative, and equal to 0 when the array is list.

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 \( \Phi \) 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 19.2 and conclude that the array list supports \( O(1) \) time per insertion.

It may be desirable to extend our array-backed list with a remove operation, that removes the last element from the list. (Together, append and remove would define an array-based stack.) Removing the element is fairly easy to do; we clear out the corresponding entry in the area, and decrease our counter tracking the size of the list. Unfortunately, after removing many elements, we may end up with an array of far greater capacity than needed. We would prefer to keep the capacity of the array within a constant factor of the size of the list. This question is explored further in exercise 19.1.

## 19.4 Lazy search trees

The reader has likely encountered binary search trees (BST’s) before and we give a brief overview. The idea is that we want to maintain a collection of ordered keys under insertions and deletions. Other auxiliary operations of interest include (a) retrieving the first key that comes before or after a query, (b) aggregate (e.g., sum) the values associated with all keys \( k \) in the range \( a < k < b \), for given points \( a \) and \( b \), and (c) list all the keys \( k \) in the range \( a \leq k \leq b \), in order. We will assume the keys are distinct for simplicity though it is easy to adjust the ideas to accommodate duplicate keys.

Binary search trees arrange the data in a rooted tree. Each node contains one key and each key belongs to one node. For each subtree we maintain the following simple rule. Let \( k \) be the key at the root of the tree. Any key in the subtree that is \(< k\) is placed in the left subtree. Any key in the subtree that is \(> k\) is placed in the right subtree. This rule makes it easy to navigate the tree. Suppose we are looking for a particular key \( k \). If the root contains \( k \), then we are done. Otherwise we compare \( k \) to the key at the root and recurse on the left or
right subtree appropriately. See fig. 19.3 for a diagram sketch, and fig. 19.4 for pseudocode.

It is easy to insert keys while maintaining the invariant. We search for where the key would be, if it were in the tree as a leaf. We place the key there as a leaf.

For both search and insertion, the worst case time to traverse a tree is entirely proportional to the height of the tree. Ideally, a tree would be balanced, where for each node, each subtree contains at most half of all the descendants. This would give a height of $O(\log n)$, which is the minimum possible height. Given the list of keys already in sorted order, it is easy to build a perfectly balanced binary search tree with height $O(\log n)$. But when the set of keys are changing, it is easy to insert keys in an order that leads to a completely unbalanced tree. In the worst case the tree can have height $\Omega(n)$ and this ruins the running time of all our operations.

Thus considerable attention is paid to maintaining a balanced binary tree as keys are inserted and deleted. Many ingenious schemes have been invented to achieve this. Red-black trees are one such data structure sometimes included in introductory data structure classes. Splay trees are another ingenious approach analyzed later in chapter 24. Here we will describe and analyze a very simple and lazy approach that heavily leverages amortized analysis.

The high-level idea is very simple. We start from the standard (unbalanced) binary search tree, with the following adjustment. As we insert keys, whenever we notice that a subtree rooted at a node $x$ is very unbalanced – meaning, one subtree of $x$ has at least twice as many nodes as the other – then we rebuild the subtree as a balanced tree. We call this strategy lazy rebuilding.

We can extend the code in fig. 19.4 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
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.)

```c
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, $k$).
```

```c
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 19.4: Implementation of a binary search tree mapping keys to values.

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.\(^3\)

\(^3\)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
In the following, we let \( n \) denote the number of elements in the binary search tree.

**Theorem 19.5.** 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 \(|T_v|\)

We will define a potential \( \Phi_v \) 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 \( \Phi_v \) 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 ensures that when \(|T_x| + |T_y|\) is odd, an as-balanced-as-possible division where \(|T_x| - |T_y| = 1\) still has potential 0. Put another way, the \(-1\) term ensures that whenever \( v \)'s key is a median of the set of keys in the subtree \( T_v \), we have \( \Phi_v = 0 \). More precisely, \( \Phi_v = 0 \) iff \( v \) represents the median over \( T_v \).

We now define a global potential \( \Phi \) as the sum of all the individual potentials,

\[
\Phi = \sum_v \Phi_v.
\]

Initially, when the tree \( \Phi = 0 \). 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 \( \Phi_v \) 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.

Suppose \( v \) has \( k \) nodes in its subtree \( T_v \). Then \( v \)'s potential, \( \Phi_v \), must have been at least \( k/3 - 1 \). Optimally rebalancing the subtree \( T_v \) takes \( O(k) \) time and decreases the potential \( \Phi_v \) to 0 for every node in \( T_v \), included \( v \). The decrease in \( \Phi_v \) alone pays for the time spent rebuilding.

Combining the two claims above gives \( O(\log n) \) amortized time overall. \( \blacksquare \)
19.4.1 Lazy deletions

Suppose now we wanted to support deletions as well. Here it is not entirely obvious what to do even in an old-fashioned BST. If we simply remove a node from the tree, unless it is a leaf, it creates a hole, and we have to scramble to fill it with some other node. One way to do this is to conduct another search to find the successor or predecessor node in the tree and somehow reattach subtrees appropriately. This approach (and others) can be quite messy and is also slow in the worst case.

Here we will analyze a much simpler and lazier alternative. We simply mark the node as deleted, and adjust the search routine to bypass keys that are marked for deletion. The remaining issue is that after many deletions, it can be that a vast majority of elements is marked for deletion and this is inefficient. 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 $O(n)$.

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 rebuild the entire tree in $O(m) = O(n)$ time.

Lemma 19.6. At any point in time, less than half the nodes in the tree are marked for deletion.

Lemma 19.7. At any point in time, the tree has size $O(\log n)$, 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 19.8. 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 $\Phi_s$, but here we should clarify that $|T_s|$ 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 \( \Phi \) that counts the total number of vertices marked for deletion. The total potential is now \( \Phi + \Phi \).

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 \( \Phi \) by \( O(\log n) \), while the time spent rebuilding a subtree is paid for by the accompanying decrease in \( \Phi \).

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 \( \Phi \) by 1. When we rebuild on account of deletions, \( \Phi \) is at least \( m/2 \). The rebuild takes \( O(m) \) time and decreases \( \Phi \) to 0. So the decrease in potential pays for the rebuild. We conclude that deletion takes \( O(\log n) \) time as well.

**Remark 19.9.** 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.

## 19.5 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 assuming immutability can greatly simplify the reasoning in your programs. If you hand off the data to some function, you are promised that the function won’t mess with your data. This is especially helpful in concurrent programming, when it is otherwise hard to keep track of all the changes made by other threads. Immutability also allows the compiler to be more aggressive when producing optimized lower level code.

A (singly) linked list is inherently immutable. A linked list consists of nodes, each of which contain 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.

\(^4\)http://learnyouahaskell.com/chapters - have fun!
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 so that the data structure is changed only from our own perspective.

Immutability means that when we want to change a small part of an object, we instead have to make an entire copy of that object that incorporates the small small change. This might seem wasteful. But often it is not so burdensome in many object-oriented systems that then do be composed of individually small objects with pointers to many others. Suppose we want to update an object that has some links to other objects. When we make a new version of that copy, we don’t have to follow the links and recursively copy those objects as well. We simply copy over the pointers to the objects in the new version.

As a concrete example, consider the lazy search trees in section 19.4. They can easily be made mutable. The primary difference is that, when we insert a key at the bottom, we have to rebuild the nodes along the path from the root to the leaf. Note that we don’t have to rebuild the subtrees hanging off the paths along the way. (Unless the algorithm calls for us to rebuild the entire tree, which we account for separately.) Fortunately for us, the lazy search trees ensure that the height of the tree is at most \( O(\log n) \). So we only create \( O(\log n) \) nodes. Each node is constant size, so this all takes \( O(\log n) \) time.

A free benefit of immutable data structures is that you automatically produce snapshots of all previous states of the data structure.

### 19.5.1 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 will not suffice because we are inserting and deleting from opposite ends of a list. In fact it seems difficult to achieve constant (worst-case) update time. But we will relax our requirements and allow for constant **amortized** update time. We challenge the reader to try to figure how to implement an immutable queue with $O(1)$ update time before proceeding below the fold.

It turns out that there is a clever way to achieve this, using only two stacks (which are already immutable). Here are the ingredients.

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 19.10.** The two-stack queue implements insertion and deletion in $O(1)$ amortized time.

**Proof.** We maintain a potential function $\Phi$ 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 $\Phi$ is the number of elements in the input stack. Meanwhile the potential $\Phi$ decreases to 0.

If we allow for amortized running time, then many data structures can be made immutable while retaining the same asymptotic performance. We refer the reader to [Oka99] for more examples.

**19.6 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], and [Blu11a, Chapter 7] for additional notes on amortized analysis. Slightly different lazy rebuilding strategies for BSTs are described in [Eri13b].

**Lecture materials and comments.** Click on the links below for the following files:

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

### 19.7 Exercises

**Exercise 19.1.** Recall the array-backed lists from section 19.3, where we showed how to support append in $O(1)$ amortized time via the “doubling trick”. As discussed at the end of section 19.3, 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.

---

5https://www.youtube.com/watch?v=3MpzavN3Mco
Exercise 19.2. 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.

\[
\text{multipop(stack } S, \ k)\\
\]

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 19.3. Section 19.5.1 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 only two immutable stacks.

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. \text{push-front}(x): inserts element $x$ at the front of the queue.
2. \text{push-back}(x): inserts element $x$ at the back of the queue.
3. \text{pop-front}(): removes the first element of the queue.
4. \text{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.)

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 while the back stack, 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.
Analyze the double-ended queue data structure described above and prove that each of the four double-ended queue operations take $O(1)$ amortized time.
Chapter 20

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 25) that we have not introduced; this algorithm is provided in section 20.A 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 19, are not data structures.

We mention that one interesting application of the push-relabel technique is an algorithm by Hao and Orlin [HO94] to the global directed minimum cut problem. We likely will not have time to discuss this topic, but we have left the details in the text for the interested reader.

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.
20.1 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.

20.1.1 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 \(f : E \rightarrow \mathbb{R}_{\geq 0}\) 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 \(v\):
   \[
   \sum_{(u,v) \in \partial^-(v)} f(u,v) \geq \sum_{(v,u) \in \partial^+(v)} f(v,u).
   \]

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
\[
\hat{f}(v) \overset{\text{def}}{=} \sum_{e \in \partial^-(v)} f(u,v) - \sum_{e \in \partial^+(v)} f(v,u).
\]

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 \(f_e\).

**Lemma 20.1.** 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. \(\blacksquare\)
20. Push-relabel

20.1. Push-Relabel Max Flow

Kent Quanrud
Spring 2022

20.1.2 Labels

The push-relabel framework\(^1\) maintains a preflow \(f\) and a set of vertex levels \(\ell : V \rightarrow \mathbb{Z}_{\geq 0}\) 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:

\[\ell(v) = \begin{cases} n & \text{if } v = s \\ 0 & \text{otherwise.} \end{cases}\]

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 \(u\).

\(^1\)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.
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 \( \ell \) to serve a similar role.

**Lemma 20.2.** 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 of these edges 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 20.2, applied to \( s \) (which has \( \ell(s) = n \)), is that \( d(s, t) \geq n \) – which can only mean the following:

**Corollary 20.3.** 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 \( \ell \) will always induce a minimum \((s, t)\)-cut, as follows.

**Lemma 20.4.** Let \( f \) be an \((s, t)\)-preflow and \( \ell \) 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 \( i \in \{1, \ldots, n - 1\} \) such that for \( U = \{v : \ell(e) > i\} \), \( \partial^+(U) \) 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 \( \ell \), observe that there exists an index \( i \in \{1, \ldots, n - 1\} \) for which there are no vertices of level \( i \). Let \( U = \{v : \ell(e) > i\} \); since the \( i \)th level is empty, we have \( \bar{U} = \{v : \ell(e) < i\} \). \( U \) contains \( s \) and \( \bar{U} \) contains \( t \), so \( \partial^+(U) \) 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, \( \partial^+(U) \) has capacity equal to the size of \( f \).
20.1.3 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 20.4, 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.

The algorithm repeats the operations above until there are no more active vertices. At that point, by lemma 20.4, \( f \) 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 a tight vertex, and identifying a forward edge from a given vertex, and so forth. But in most cases, it is easy to see how to use some simple data structures to facilitate the operations (e.g., tracking the active vertices in a list), and the bottleneck will come the total number of basic operations.

\(^2\)Some scaling algorithms do not necessarily push the maximum amount.
Relabelings.

**Lemma 20.5.** 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 has level \( \leq \ell(s) + n - 1 = 2n - 1 \). Moreover, only active vertices get relabeled. \( \blacksquare \)

Saturating pushes.

**Lemma 20.6.** 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. \( \blacksquare \)

Non-saturating pushes.

The trickiest to analyze is non-saturating pushes and here we employ amortized analysis.

**Lemma 20.7.** 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 \( \Phi \) by one. A saturating push increases \( \Phi \) by at most \( O(n) \). A relabel increases \( \Phi \) by one.

Since there are at most \( O(mn) \) saturating pushes, and \( O(n^2) \) relabels, the total increase to \( \Phi \) is \( O(mn^2) \). Moreover, \( \Phi \) is initially zero, and always nonnegative. Charging each non-saturating push to a decrease in \( \Phi \), we conclude that there are at most \( O(mn^2) \) non-saturating pushes. \( \blacksquare \)

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.

### 20.2 Top-down push-relabel

Section 20.1.3 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. More specifically, any push-relabel algorithm makes at most $O(mn^2)$ non-saturating pushes, which is the bottleneck. We also showed that there are at most $O(n^2)$ relabels and $O(mn)$ saturating pushes.

There are many possible strategies within this framework, and perhaps different approaches can lead to different performance in practice, or better upper bounds (especially for non-saturating pushes). The main decision is which active vertex to select at a given moment in time. Here are a few possible examples.

1. Select the active vertex $v$ with highest label $\ell(v)$.
2. Select the active vertex $v$ with 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 lowest excess $\hat{f}(v)$.
7. Select an active vertex $v$ uniformly at random.

In this section we consider the first strategy, always addressing the active vertex of highest level.

**Lemma 20.8.** 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 vertex of $v$ being promoted to a higher label. Each vertex gets promoted at most $O(n)$ times. ■
The lemma above reduces the number of non-saturating pushes 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 20.9.** 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 ???. We note that the FIFO strategy also obtains a $O(n^3)$-running time; we leave the proof to the reader as exercise 20.1.

### 20.2.1 $O(n^2 \sqrt{m})$ 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 20.10** (Cheriyan and Maheshwari [CM89] and Tunçel [Tun94]). The highest-level 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-level rule makes at most $O(n^2 \sqrt{m})$ non-saturating pushes; we leave it to the reader to devise a $O(n^2 \sqrt{m})$ implementation.

Define a potential $\Phi$ 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 $v$.

$\Phi$ is initially 0. A relabeling or a saturated push can increase $\Phi$ by at most $n$. So the total increase to $\Phi$ is at most $O(mn^2)$. On the other hand, a saturating push at a vertex $v$ de-activates $v$ and decreases $\Phi$, by at least the number of vertices $w$ 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 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 a $k \in \mathbb{N}$ 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 $O(n^2k)$ small non-saturating pushes overall. Meanwhile each big push decreases $\Phi$ 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. ■

20.3 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

\[
\text{every vertex has excess at most } 2\Delta.
\]

We only make non-saturating pushes of size exactly $\Delta$. (That is, even if we could push more, we truncate the push to $\Delta$ 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 $2\Delta$.

We continue to push $\Delta$ units at a time until all active vertices have excess less than $\Delta$. This means we can divide $\Delta$ in half while still maintaining the invariant that every vertex has excess at most $2\Delta$. We then continue with this smaller value of $\Delta$.

Since we are repeatedly halving, it is convenient to maintain $\Delta$ 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 $\Delta < 1/2$, and our 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 $\Delta$ at most $O(\log(U))$ times before the algorithm terminates.

Now, consider the following potential functions which allows us to amortize

\(^3\)Like an elevator.
non-saturating pushes.

\[ \Phi = \frac{1}{\Delta} \sum_{v \in V} \hat{f}(v) \ell(v). \]

**Lemma 20.11.** Relabeling and saturating pushes take \( O(1) \) amortized time (w/r/t \( \Phi \)). 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 \( \Delta \) units of flow to a lower-level, \( \blacksquare \)

The remaining operation to analyze – unique to scaling – is when we decrease \( \Delta \). Here we have the following.

**Lemma 20.12.** Dividing \( \Delta \) in half takes \( O(n^2) \) amortized time.

**Proof.** Observe the potential is at most \( O(n^2) \) because \( \hat{f}(v) \leq 2\Delta \) for all \( v \). Halving \( \Delta \) doubles the potential \( \Phi \), which gives the amortized running time. \( \blacksquare \)

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 20.13** (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.

### 20.4 Additional notes and references

Preflows were introduced by Karzanov [Kar74], who employed them as part of a blocking flow subroutine. (Blocking flows are a different approach to max flow that we have not discussed.) 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], Frank [Fra11], Schrijver [Scho93], and Williamson [Wil19]; [Fra11; Wil19] also include the Hao-Orlin minimum cut algorithm from section 20.B. 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].
Lecture materials and comments. Click on the links below for the following files:

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

20.5 Exercises

Exercise 20.1. In section 20.2, 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.) Show that this push-relabel algorithm (appropriately implemented) takes $O(n^3)$ to find the maximum flow.  

Exercise 20.2. In the push-relabel framework, we have seen that different strategies for selecting active vertices can lead to better running times. Here we will consider a variation of push-relabel that we will call greedy push-relabel. The idea is to repeatedly select the push of maximum total size. We will assume the capacities are integral (which will be important for the proof).

$$\text{greedy-push-relabel}(G = (V,E), c : E \to [U], s, t)$$

1. Initialize a preflow $f$ and levels $\ell : V \to \mathbb{Z}_{\geq 0}$ (as usual).
2. Until $f$ is a flow:
   A. Let $v \in V - \{s,t\}$ have maximum excess $\hat{f}(v)$.
   B. Push or relabel from $v$ (as usual).
3. Return $f$.

This exercise asks you to analyze greedy-push-relabel as follows. We point out that the upper bounds from section 20.1.3 come for free since greedy-push-relabel is a special case of the generic push relabel algorithm. You may also want to look at section 20.3 for inspiration.

4Hint: We already know that there are at most $O(mn)$ saturating pushes, which is at most $O\left(\frac{n^3}{m}ight)$. So the real question is why there are at most $O\left(\frac{n^3}{m}ight)$ non-saturating pushes.
1. Suppose the maximum total excess over all non-terminals at a fixed point in time is \( X \); i.e., \( \sum_{v \in V - \{s, t\}} \hat{f}(v) = X \). Prove that there are at most \( O(n^2) \) non-saturating pushes before the total excess over all non-terminals is less than \( X/2 \).

2. Prove an upper bound, as small as possible, on the total number of push-relabel operations made by greedy-push-relabel.

   You will probably want to use the previous part in your argument. The integrality of the capacities should play a role in your analysis.

   (Food for thought: A full implementation of greedy-push-relabel would require some data structures so you can quickly identify the next active vertex. How might you implement the algorithm? What is your actual running time?)

---

**Exercise 20.3.** 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, 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. \( \text{AddSourceEdge}(v, \gamma) \): Add an edge \((s, v)\) with capacity \( c(e) = \gamma \). If \( \ell(v) < \ell(s) \), we saturate \( e \) by setting \( f_e = \gamma \).

2. \( \text{RemoveSinkEdge}(v, \gamma) \): Decrease the capacity of an edge \((v, t)\) by \( \gamma \). (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 \( \ell \) RemoveSinkEdge operations (possibly intermixed).

1. For each call to \( \text{AddSourceEdge}(v, \gamma) \), 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 $\text{DecreaseSinkEdge}(v, \gamma)$ 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 $\gamma$. 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. $\text{Init}(G = (V, E), c : E \rightarrow \mathbb{R}_{\geq 0}, s, t)$ (where we compute the initial max flow) takes $O(mn^2)$ amortized basic operations.

2. $\text{AddSourceEdge}(v, \gamma)$ takes $O(n^2)$ amortized basic operations.

3. $\text{DecreaseSinkEdge}(v, \gamma)$ takes $O(n)$ amortized basic operations.

That is, after computing the initial flow, and then processing an intermixed sequence of $k$ calls to $\text{add-source-edge}$ and $\ell$ calls to $\text{decrease-sink-edge}$, 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 20.2.1?)

### 20.A 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 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 20.14. 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. See chapter 25 for more details and explanations of the dynamic tree operations.

We 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.

\( \blacksquare \)
20.B Directed minimum cut via one push-relabel

We now shift gears to a different but related problem: minimum directed cut. A directed cut is a set of edges $C \subset E$ whose removal makes the graph not strongly connected. That is, there exists some pair $s, t \in V$ for which $C$ is an $(s, t)$-cut. The goal is to find the minimum weight directed cut.

Of course, we can reduce the problem to $(s, t)$-cut by enumerating all pairs. This requires $O(n^2)$ $(s, t)$-cut computations. We can do better by fixing $s$, and then enumerating all $t \in V - s$. For each $t$, we compute the minimum $(s, t)$-cut and the minimum $(t, s)$-cut. The question is whether one can do better.

**Rooted connectivity.** A slight change in perspective that is very helpful is to instead focus on the rooted connectivity problem. Fix a vertex $r \in V$, called the root. A rooted cut, or $r$-cut, is a set of edges forming an $(r, t)$-cut for some $t \in V - r$. The minimum rooted cut asks for the minimum weighted rooted cut. Of course an algorithm for minimum $r$-cut solves the minimum directed cut problem, by picking any $r \in V$, and computing the minimum $r$-cut in both $G$ and the graph obtained by reversing all the edges in $r$.

Now, to find the minimum $r$-cut, we can of course enumerate all $t \in V - r$ and compute the minimum $(r, t)$-cut for each. This approach seems excessive for a couple reasons.

1. After computing the minimum $(r, t)$-cut for some $t$, if this cut is not the minimum $r$-cut, then we know that $t$ must be on the $r$-side of the minimum $r$-cut. So we can reduce the size of the graph slightly by contracting $t$ into $r$ before continuing to the next choice of $t$.

2. After computing the minimum $(r, t)$-cut, we (typically) also have a maximum $(r, t)$-flow to certify it. It seems somewhat wasteful to throw away all this work when computing the $(r, t')$-cut for the next sink $t'$. Perhaps there is some way to extend

These informal observations motivate Hao and Orlin’s algorithm [HO94], which computes the minimum $r$-cut with a single push-relabel algorithm. The algorithm still enumerates all choices of $t \in V - r$, but is careful to continue the progress from one $(r, t)$-cut problem to the next. At a high level, the algorithm is as follows.

1. Pick some vertex $t_1 \in V - r$, and compute the minimum $(r, t_1)$-cut with push-relabel. (The cut is certified by a preflow, which we explain in greater detail below.) Take note of the value of this cut.

2. Now we want to pick a new vertex $t_2$ as the next sink, and convert $t_1$ into a source (effectively contracting it with $r$). We pick a vertex $t_2$ with $\ell(t_2) = 1$
(the existence of which is assured by some modifications to the push-relabel algorithm) as the next sink. We make $t_2$ is a source by pushing as much flow as possible out of $t_2$ as to saturate all its edges, and set $\ell(t_1) = n$. (So $t$ will acts as a second source in addition to $r$.) Taking the existing flow and labels as a starting point, we continue the push-relabel algorithm as to compute the minimum $(\{r, t_1\}, t_2)$-cut.

3. We then select $t_2$ with $\ell(t_2) = 2$, convert $t_2$ into a source (just like for $t_1$), and continue. In this fashion we iterate through all the other sinks.

Above we only give a high-level description and many details have yet to be filled in. An important difference from before is that we only run push-relabel until we find a minimum $(r, t)$-cut certified by a preflow – and not necessarily to the point where $f$ is a flow. To this we make the following observations about preflows.

**Max-preflow min-cut.** Given a preflow $f$, we define the size of $f$ to be the net flow at the sink vertex. Lemma 20.1 can be rephrased as saying that every preflow $f$ contains a flow $f'$ of the same size. This implies the following.

**Lemma 20.15.** The maximum size of any $(s, t)$-preflow equals the minimum capacity of any $(s, t)$-cut.

**Lemma 20.16.** Let $f$ be a preflow and $\ell$ be a set of levels satisfying the items (I) and (II). Suppose there exists an index $i > 0$ such that:

1. There are no vertices $v$ with level $\ell(v) = i$.
2. There are no active vertices $v$ with level $\ell(v) < i$.

Then for $T = \{v : \ell(v) < i\}$, $\partial^{-}(T)$ is a minimum $(s, t)$-cut.

**Proof.** In general, the capacity of $\partial^{-}(T)$ in the residual graph is reduced by exactly the total excess of all vertices in $T$. Since $T$ has no active vertices, this is exactly the size of $f$. Meanwhile the capacity of $\partial^{-}(T)$ is 0 in the residual graph. $\blacksquare$

Now we are prepared to discuss the modifications to the push-relabel algorithm. Our goal is to compute a preflow $f$ and level set $\ell$ as to satisfy the conditions of lemma 20.16. To this end, we always treat the active vertex of the lowest level. (This strategy is compatible with both the link-cut trees approach of ?? and the scaling approach of section 20.3). If, in the course of treating the lowest level first, a level becomes empty, we immediately trigger lemma 20.16 and have obtained a minimum $(r, t)$-cut. Now, we want to ensure if the $i$th sink $t_i$ is at level $\ell(t_i) = i - 1$, then we will be able to find the $(i + 1)$th sink $t_{i+1}$ at level $i$. This could be an issue, for example, if the empty layer induces the minimum $(r, t)$-cut is at level $i$. To address this, after creating the empty layer via relabel and extracting the cut,
we undo that relabel. (Alternatively, we cancel the relabel and output the cut anyway, knowing that it would have been certified as the minimum $r$-cut). Thus we finish the iteration without actually creating an empty layer, and guaranteeing that the next sink can be found in the next level.

Note that in this algorithm, we never have to relabel a vertex beyond level $n - 1$ (except when a sink $t$ is converted to a source, in which case its level is set to $n$).

If we use the link-cut tree implementation, then we obtain the following running time.

**Theorem 20.17** (Hao and Orlin [HO94]). *The minimum $r$-cut can be found in $O(mn \log (n^2 / m))$ deterministic time.*

Similarly obtains the analogous result via scaling. See also [Fra11] for the description of a $O(n^3)$ running time without data structures or scaling.
Chapter 21

Random Sampling, Searching, and Sorting

21.1 Basic probability

Probability theory is simple and intuitive. Let us start with a familiar example.

Suppose we flip a coin in the air, and press pause. Will it land heads or tails? Obviously, we don’t know yet. But we can state without ambiguity that half the time it will land heads, and half the time it will land tails. What does it mean to 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. (The only tricky part is sticking to them!) We assume the reader has some acquaintance already with random events and variables, but we will still review the basics. Probability theory assumes an uncertain world where events occur with fixed probabilities. Each event $A$ had a probability between 0 and 1, denoted

$$ P[A] \in [0,1]. $$

For every event $A$, there is the complementary event, $\bar{A}$, of $A$ not occurring. We always have

$$ P[A] + P[\bar{A}] = 1. $$

(21.1)
Joint events. For any two events $A$ and $B$, one can define the conjunctive event that both $A$ and $B$ occurs: denoted

"$A \land B$" or "$A \cap B$" or "$A$ and $B$".

Sometimes we also write $P[A,B]$ to denote the probability of $A \land B$.

In general, given $A$ and $B$, there are four disjoint events induced by their combination:

1. $A \land B$: The event that both $A$ and $B$ occur.
2. $A \land \overline{B}$: The event that $A$ occurs and $B$ does not.
3. $\overline{A} \land B$: The event that $A$ does not occur and $B$.
4. $\overline{A} \land \overline{B}$: The event that neither $A$ nor $B$ occurs.

The four joint events listed are mutually exclusive – only one of them can be realized – and exhaustive – one of the will be realized. So we have the following identity:

$$P[A \land B] + P[A \land \overline{B}] + P[\overline{A} \land B] + P[\overline{A} \land \overline{B}] = 1$$

Suppose event $A$ occurs with positive probability. Whenever $A$ occurs, then of course exactly one of $B$ occurs or $\overline{B}$ occurs. So we have

$$P[A] = P[A,B] + P[A,\overline{B}]$$

If we divide both sides by $P[A]$, we have

$$1 = \frac{P[A,B]}{P[A]} + \frac{P[A,\overline{B}]}{P[A]}.$$

This equation looks probabilistic – we have two nonnegative terms summing to 1. We can interpret the first term on the right-hand side, $P[A,B]/P[A]$, as saying that

of the times that event $A$ occurs, $B$ also occurs $P[A,B]/P[A]$ fraction of the time.

We call this the conditional probability of event $B$ conditional on event $A$, denoted $P[B \mid A]$ and defined as the ratio

$$P[B \mid A] = \frac{P[B \mid A]}{P[A]}.$$
Likewise we have the conditional probabilities $P[\bar{B} \mid A]$, $P[B \mid \bar{A}]$, $P[\bar{B} \mid \bar{A}]$, and so forth.

We always have

$$P[A \wedge B] \leq \min\{P[A], P[B]\}$$

(21.1). It is generally not true that

$$P[A \mid B] = P[A]$$

(21.2)

for two events $A$ and $B$. The equation above is equivalent to the one with $A$ and $B$ flipped, as well as

$$P[A \wedge B] = P[A] P[B]$$

(21.3)

Events $A$ and $B$ are said to be independent events in the special case where (21.3) holds.

**Unions of events.** We also have the disjunctive event that either $A$ or $B$ occurs, denoted

"$A \lor B$" or "$A \cup B$" or "$A$ or $B$".

If $A \lor B$ occurs, then exactly one of the following events occurs:

$$A \wedge B, A \wedge \bar{B}, \bar{A} \wedge B.$$  

Consequently we have

$$P[A \lor B] = P[A \wedge B] + P[A \wedge \bar{B}] + P[\bar{A} \wedge B].$$

Recall that $P[A] = P[A \wedge B] + P[A \wedge \bar{B}]$, and similarly for $P[B]$. Adding $P[A \wedge B]$ to both sides of the identity above gives


This identity reflects a venn-diagram, so to speak, where the two regions $A$ and $B$ give rise to their “union” $A \lor B$ and their intersection $A \wedge B$.

The identity above leads to the following extremely useful union bound.

**Lemma 21.1** (Union bound).

$$P[A \lor B] \leq P[A] + P[B],$$

with equality iff $P[A \wedge B] = 0$. 

318
21. Random Sampling, Searching, and Sorting

21.2. Randomized minimum cut

Recall the minimum cut problem in undirected graphs. The input consists of a connected, undirected graph $G = (V, E)$ with positive edge capacities $c : E \rightarrow \mathbb{R}_{>0}$. A cut is a set of edges $C \subseteq E$ whose removal disconnects the graph. The goal is to

$$\text{minimize} \sum_{e \in C} c(e) \text{ over all cuts } C \subset E. \quad (21.4)$$

This problem is polynomial time solvable. Whatever the optimum cut is, it must be a minimum $(s,t)$-cut for some pair of vertices $s$ and $t$. Thus to find the global minimum, one can guess $s$ and $t$ by looping over $V$, and compute the minimum $(s,t)$-cut for each choice of $s$ and $t$. (Better yet: fix $s$, and loop over all $t$.)

Previously, we also studied an algorithm due to [NI92b] that runs in $O(mn)$ time.

For a set of vertices $S$, let $\partial(S)$ denote the set of edges with exactly one endpoint in $S$. $\partial(S)$ is called the cut induced by $S$. The induced cuts are also the inclusionwise minimal cuts, and it suffices to consider only the induced cuts when solving (Min-Cut).

We will study a stunning algorithm discovered by Karger [Kar93], that has been influential beyond minimum cut. Consider the following description of Karger’s algorithm.

Repeatedly sample edges in proportion to their capacities until there is only one cut from which we have not yet sampled any edges. Return this cut.

This algorithm is clearly ridiculous. For the unweighted setting, the above algorithm is equivalent to the following, equally absurd approach (see exercise 21.7).

Independently assign every edge $e \in E$ a weight $w_e \in [0,1]$ uniformly at random. Build the minimum weight spanning tree $T$ w/r/t $w$. Let $e$ be the heaviest edge in $T$. Return the cut induced by the two components of $T - e$.

Compare the two approaches above. Of course we know how to compute the minimum spanning tree; among other approaches, we can repeatedly add the smallest weight edge to $T$ that does not create a cycle. On the other hand, in the first approach, it might appear difficult to keep track of which cuts we have and have not sampled from, being that there are so many cuts. This can be addressed by contracting the graph.

Suppose we sample an edge $e = \{s,t\}$. Then we know that any cut $\partial(S)$, where $s \in S$ and $t \notin S$, has now been sampled from. Thus we can safely contract $e$; replacing $s$ and $t$ with
random-contractions($G = (V,E), e$)

1. while $|E| > 1$
   A. sample $e \sim c$
   B. $G \leftarrow G/e, c \leftarrow c/e$
2. let $E = \{e\}$
3. return edges in the original graph that contracted to $e$

Figure 21.1: A randomized minimum cut algorithm due to Karger [Kar93].

Figure 21.2: A few iterations (from left to right, top to bottom) of the random-contractions algorithm applied to a barbell graph.

Now imagine we contract edges as we sample them. Eventually there are only two vertices left in the contracted graph, which represent two connected components in the input graph. These components induce the only cut we have not yet sampled from, and this is the cut that we return.

Pseudocode for the contraction algorithm is given in fig. 21.1. Here, for an edge $e \in G$, we let $G/e = (V/e, E/e)$ denote the graph obtained by contracting

\footnote{More precisely, for every edge $f$ of the form $\{s, z\}$ or $\{t, z\}$, we create a new edge $\{u, z\}$ with the same capacity. If $s$ and $t$ both have edges to the same vertex $z$, we can either create two edges from $u$ to $z$ with the appropriate capacities, or make a single edge from $u$ to $v$ with the same capacity. We remove $s$ and $t$ and its incident edges from the graph, replacing them with $u$ and the newly created edges incident to $u$.}
e, and we let $c/e : E/e \rightarrow \mathbb{R}_{>0}$ denote the corresponding capacities. Figure 21.2 sketches a few iterations of the algorithm applied to a barbell graph.

The intuition behind random-contractions is as follows. Here we describe the intuition for unweighted graphs for simplicity. (The intuition is the same for weighted graphs, except replacing “many edges” with “large capacity”, etc.) Suppose we have an unweighted graph $G = (V, E)$, and let $C \subset E$ be the minimum cut. Since $C$ is the minimum cut – keyword minimum – there are presumably very few edges in $C$. If we randomly sample an edge $e \in E$, then hopefully $e \notin C$.

If $C$ “survives” this round, then we have all made some progress because there is one less vertex in the graph after contracting $e$. In the next round, $C$ is still the minimum cut, so the high-level logic from the first round still holds. Thus we can repeatedly sample edges and preserve the hope that we avoid $C$.

The above argument hinges on how much smaller $C$ is than all of $E$. If we can argue that $C$ is always a small fraction of $E$, then that gives hope that $C$ survives to the end. On the other hand, if $C$ is even a small constant fraction of $G$, we will probably sample from $C$ after a constant number of rounds. Observe also that over time, $C$ becomes a larger and larger fraction of $E$, as we contract and remove edges outside of $C$.

The key observation is that every vertex $v$ induces a cut $\partial(v)$, which must have at least as many edges as $C$. Thus the minimum cut is at most the minimum degree in the graph. In turn, since the number of edges in $E$ is the sum of degrees (divided by 2), the minimum cut $C$ is at most a $2/n$ fraction of the total number of edges! This observation holds initially in the input graph and thereafter in the contracted graphs, although $n$ decreases by 1 in each iteration.

On the first iteration, $C$ has at most a $2/n$ chance of being hit. On the second iteration, assuming $C$ survived the first iteration, $C$ has (at most) a $2/(n - 1)$ chance of being hit. Continuing in this fashion, assuming $C$ survived the first $i - 1$ iterations, $C$ has a $2/(n - i + 1)$ chance of being hit in the $i$th iteration. If one combines these problems, one discovers that $C$ has a $\geq 1/\binom{n}{2}$ chance of surviving all $n - 1$ rounds. We can repeat the experiment $\binom{n}{2} = O(n^2)$ (a polynomial!) number of times to find the minimum cut with constant probability, and $O(n^2 \log n)$ times to find the minimum cut with high probability.

In the sequel, we formalize the the above argument, as well as extend it to positive capacities. For ease of notation, for a set of edges $C \subset E$, we denote the sum of capacities over $C$ by

$$
\bar{c}(C) \overset{\text{def}}{=} \sum_{e \in C} c(e).
$$

**Lemma 21.2.** Let $C^*$ be the minimum cut in $(G, c)$, and suppose $e \notin C$. Then $C^*$ is (or maps to) to the minimum cut in the contracted graph $(G/e, c/e)$.
**Lemma 21.3.** \( \bar{c}(E) \geq \frac{\lambda n}{2} \).

**Proof.** Every vertex \( v \) has weighted degree \( \bar{c}(\partial(v)) \geq \lambda \) since \( \partial(v) \) is a cut. Thus

\[
\bar{c}(E) = \sum_v \bar{c}(\partial(v)) \geq \frac{\lambda n}{2}.
\]

**Lemma 21.4.** Let \( e \sim c \). Then \( \mathbb{P}[e \in C^*] \leq \frac{2}{n} \).

**Proof.** We have \( \mathbb{P}[e \in C^*] = \frac{\bar{c}(C^*)}{\bar{c}(E)} \leq \frac{2}{n} \) by (a) lemma 21.3.

**Lemma 21.5.** Let \( C^* \) be a minimum cut. With probability \( \geq \frac{1}{n^2} \), random-contractions returns \( C^* \).

**Proof.** For \( k \in \mathbb{Z}_{\geq 0} \), let \( E_k \) be the event that we have not sampled \( C^* \) after \( k \) iterations. Initially, \( \mathbb{P}[E_0] = 1 \), and we want to show that \( \mathbb{P}[E_{n-2}] \geq 1/(\binom{n}{2}) \). By lemma 21.4, we have

\[
\mathbb{P}[E_k \mid E_{k-1}] \geq 1 - \frac{2}{n-i(k-1)} \quad \text{for each} \quad k \in [n].
\]

The probability of succeeding (event \( E_{n-2} \)) is at least

\[
\mathbb{P}[E_{n-2}] = \prod_{k=1}^{n-2} \mathbb{P}[E_k \mid E_{k-1}] \geq \prod_{i=3}^{n} \left( 1 - \frac{2}{i} \right) = \frac{1}{\binom{n}{2}}.
\]

Thus with probability about \( 1/n^2 \), the random contraction algorithm returns the minimum cut. To find the minimum cut with constant probability, we rerun the algorithm \( O(n^2) \) time and return the best cut. To find the minimum cut with high probability, we rerun the algorithm \( O(n^2 \log n) \) times.

The nice thing about repetition is that we can run the randomized trials in parallel. Moreover, a single instance of the contraction algorithm (via its connection to minimum spanning trees) can be made to run in polylog\( (n) \) time with polynomially many processors. Thus one obtains a randomized parallel algorithm for minimum cut.

**Corollary 21.6.** A randomized minimum cut can be computed in parallel in polylogarithmic time with a polynomial number of processors.
That said, random-contractions is not just an algorithm. It is also a profound structural observation about the number of minimum cuts in an undirected graph. In the above algorithm, any fixed minimum cut is returned with probability $1/n^2$. This implies that there are at most $n^2$ minimum cuts in the graph!

**Corollary 21.7.** There are at most $n^2$ minimum cuts in a graph.

### 21.3 Random variables

A *finite random variable* models an unrealized and uncertain object $X$ that takes one of a finite set of values, $\{x_1, \ldots, x_k\}$. For each outcome $x_i$, “$X$ equals $x_i$” is an event, with a fixed probability, denoted $P[X = x_i]$. These probabilities sum to 1:

$$\sum_{i=1}^{k} P[X = x_i] = 1.$$

For example, we can describe a coin toss as a random variable $X \in \{\text{heads, tails}\}$. Let a fair coin be tossed. If the coin comes up heads, then $X = \text{heads}$. If the coin comes up tails, then $X = \text{tails}$. We have

$$P[X = \text{heads}] = P[X = \text{tails}] = \frac{1}{2}.$$  

Observe that these probabilities sum to 1.

If we have two random variables $X \in \{x_1, \ldots, x_k\}$ and $Y \in \{y_1, \ldots, y_\ell\}$, then their product $(X, Y)$ forms a random variable in the set $\{(x_i, y_j) : i = 1, \ldots, k, j = 1, \ldots, \ell\}$. We have probabilities of the form

$$P[X = x_i, Y = y_j]$$

that gives the probability that $(X, Y) = (x_i, y_j)$. It is not generally true

$$P[X = x_i, Y = y_k] = P[X = x_i] P[Y = y_k]$$

In the special case where the above holds for all $x_i$ and $y_j$, then $X$ and $Y$ are said to be independent.

For example, suppose $X, Y \in \{\text{heads, tails}\}$ describe coin tosses. If they described different coin tosses, then they would be independent random variables,

---

$^2$One can also define continuous variable (e.g., that take values continuously between 0 and 1), where sums are replaced by variables.
21. Random Sampling, Searching, and Sorting

21.4 Averages

Thus $X$ and $Y$ are independent random variables. If they described the same coin, then we would have

$$P[X = \text{heads}, Y = \text{heads}] = P[X = \text{tails}, Y = \text{tails}] = \frac{1}{4},$$

while

$$P[X = \text{heads}, Y = \text{tails}] = P[X = \text{tails}, Y = \text{heads}] = 0.$$

Here, $X$ and $Y$ are not independent.

21.4 Averages

When a random variable $X$ takes on real values, we can have a well-defined and quantitative notion of “averages”, called the expected value.

**Definition 21.8.** 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 $E[X]$, is the weighted sum

$$E[X] \overset{\text{def}}{=} \sum_x P[X = x] \cdot x;$$

where the sum is over all values of $x$ where $P[X = x] > 0$.

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 21.9 (Linearity of expectation).** Let $X, Y \in \mathbb{R}$ be two random variables. Then

$$E[X + Y] = E[X] + E[Y].$$
The proof of linearity of expectation is left as exercise 21.1. 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 generally 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:

\[
\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y].
\]

Of course, this makes total sense.

21.5 Randomized sorting

Let us return to the very basics: sorting. Recall that the goal is to take an unordered list of comparable elements (e.g., numbers) and return them in a list in sorted order. Previously we studied the merge-sort algorithm which ran in \( O(n \log n) \). Here we will study a randomized algorithm that is remarkably simple, called quick-sort, that is often the preferred one is very simple. The idea is very simple: choose a pivot uniformly at random, divide the elements by the pivot, and recurse on both halves. See fig. 21.3 for pseudocode.

The running time is proportional to the total number of comparisons made by the algorithm. It is not impossible that the algorithm makes \( O(n^2) \) comparisons (how?). However, the algorithm is randomized, and a more useful measure is the average number of comparisons.

We remark that a probabilistic analysis of quick-sort seems much more complicated than typical exercises in probability. We are analyzing a randomized algorithm, where one randomized decision affects the dynamics for all randomized decision. There are overwhelmingly many “butterfly effects” to consider.

The saving grace is that we are not trying to map out all the probabilistic outcomes with complete precision. We will only be interested in analyzing the running time on average; that is, in the aggregate in a certain probabilistic sense.
21. Random Sampling, Searching, and Sorting

21.5. Randomized sorting

Kent Quanrud
Spring 2022

```plaintext
quicksort(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 21.3: A randomized sorting algorithm.

We will show that quick-sort takes \( O(n \log n) \) time in expectation against any input. This is still a worst case analysis in the sense that it holds for any input. This is not to be confused with the performance of an algorithm against a randomized input from a particular distribution – that is called average case analysis.

**Theorem 21.10.** 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 \( i \)th 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. Let us first consider \( X_{ij} \) for fixed \( i < j \). Observe that the rank \( i \) and rank \( j \) numbers are compared to each other if and only if 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 \( \frac{2}{j - i + 1} \). That is,

\[
\mathbb{E}[X_{ij}] \overset{(a)}{=} \mathbb{P}[X_{ij} = 1] = \frac{2}{j - i + 1}.
\]

Note that (a) holds for any \( \{0,1\} \)-random variable.

Let us now return to the entire 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

\[
\mathbb{E} \left[ \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} \right] \overset{(b)}{=} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] \overset{(c)}{=} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n} \sum_{k=1}^{n-i} \frac{2}{k} \leq O(n \log n),
\]

326
as desired. Here (b) applies linearity of expectation; the average sum is equal to the sum of averages. (c) is from our analysis for a single $X_{ij}$ above.

**Remark 21.11.** Moreover, one can show that the running time is $O(n \log n)$ with exceedingly high probability – though to prove this we need generalizations of the law of large numbers called *concentration inequalities*. 

### 21.6 Randomized approximations for SAT

Recall the SAT problem. We have a Boolean formula $f(x_1, \ldots, x_n)$ in conjunctive normal form, with $n$ variables and $m$ clauses. The goal is to find an assignment $x_1, \ldots, x_n \in \{\text{true}, \text{false}\}$ that satisfies as many classes as possible.

As we know, we do not know how to solve this exactly in polynomial time and it is considered unlikely that we will. So instead researchers often develop *approximation algorithms* for SAT. Here the goal is 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. Instead we will design algorithms that are *provably* competitive with the optimum solution.

Given a SAT formula $f$, let OPT denote the maximum number of clauses that are satisfiable. For $\alpha \in [0, 1]$, an \textit{$\alpha$-approximation algorithm} for SAT is an algorithm that produces an assignment that satisfies at least $\alpha \OPT$ clauses. While obtaining an exact algorithm is NP-Hard, for $\alpha < 1$, it is not necessarily NP-Hard to obtain an $\alpha$-approximation algorithm for SAT. Here we will develop randomized approximation algorithms that obtain

Consider the special case of 3-SAT. Here we have $m$ clauses, each of which have exactly 3 variables.

\[
\text{random-SAT}(f(x_1, \ldots, x_n))
\]

1. for each $i \in [n]$, draw $x_i \in \{\text{true}, \text{false}\}$ independently and uniformly at random.

2. return $x_1, \ldots, x_n$

Consider a single clause; e.g., $(x_1 \land \bar{x}_2 \land x_3)$. Of all ways to assign the three variables ($x_1, x_2, x_3$ in the example) values in $\{\text{true}, \text{false}\}$, there is only one way that does \textit{not} satisfy the clause. That is, each clause is satisfied with probability $7/8$.

In expectation, we have

\[
\mathbb{E}[\text{clauses satisfied}] \triangleq \sum_{i=1}^{m} \mathbb{P}[\text{ith clause is satisfied}] = \frac{7}{8}m.
\]
(a) applies linearly of expectation. Since \( m \geq \text{OPT} \), we obtain a \( 7/8 \)-approximation ratio (in expectation).

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 21.12.** For all \( k \in \mathbb{N} \), there is a randomized \((1 - 1/2^k)\)-approximation algorithm for \( k \)-SAT.

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 \( P = \mathcal{NP} \). 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 the proof in the Fall 2020 randomized algorithms course [Qua20b].

### 21.7 Exercises

**Exercise 21.1.** Prove linearity of expectation (theorem 21.9).

**Exercise 21.2.** Let \( A \) and \( B \) two events. Prove that \( A \) and \( B \) are independent iff \( \overline{A} \) and \( \overline{B} \) are independent.

**Exercise 21.3.** 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 21.4.** For \( k \in \mathbb{N} \), 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? 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 21.5.** 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 number of biased coin tosses you make may in expectation may depend on \( p \).)

**Exercise 21.6.** Recall the selection problem. Given an input \( A[1..n] \) of \( n \) comparable elements, the goal is to find the rank \( k \) element (that is, the \( k \)th largest element) in \( A \). You may assume the elements are distinct for simplicity.

Previously we studied a deterministic, linear time algorithm called **median-of-medians**. The algorithm was rather non-trivial – it recursively computes an approximate median to be used as a pivot. Here we will study a much simpler algorithm called **quick-select** that uses randomization instead of divide-and-conquer. The idea is very simple and similar to **quick-sort**: select a pivot randomly. In practice, this simpler algorithm is faster than **median-of-medians**.

\[ \text{quick-select}(A[1..n], k) \]

// 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 \( \ell \) 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.

---

\(^3\)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 think about a fair coin, where \( p = 1/2 \).
The goal of this exercise is to prove that quick-select takes $O(n)$ time in expectation. We ask you to prove this in two different ways which offer two different perspectives.

1. **Approach 1.** Analyze quick-select similarly to quick-sort, based on the sum of indicators $X_{ij}$.
   
   You may want to do 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 $O(n)$. Use this to obtain a $O(n)$ expected running time, overall.

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 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 $k$th element or throw out at least $1/4$ of the remaining elements.

   (b) 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 $O(1)$.

   (c) 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 $\leq O((4/3)^i)$

   (d) Finally, use the preceding part to show that the expected running time of quick-select is $O(n)$.

---

4The trickiest is the family of $X_{ij}$ where $i < k < j$. As an intermediate step, you might show that expected sum is $O(k \log (n/k))$. Is this increasing or decreasing in $k$?

5Hint: Exercise 21.4.
**Exercise 21.7.** Consider the randomized algorithm for minimum cut based on building the minimum spanning tree w/r/t randomized weights, described in section 21.2.

1. Prove that this algorithm is equivalent to the random contractions algorithm for unweighted graphs.

2. Adjust the randomized spanning tree algorithm to account for weights, and prove its correctness.

**Exercise 21.8.** Let $G = (V,E)$ be an undirected graph. For $k \in \mathbb{N}$ a $k$-cut is a set of edges whose removal disconnects the graph into at least $k$ connected components. Note that for $k \geq 3$, the minimum $k$-cut problem cannot easily be reduced to $(s,t)$-flow. In fact, the problem is NP-Hard when $k$ is part of the input.

1. Design and analyze a randomized algorithm that runs in $n^{ck}$ randomized time for a constant $c$ and returns the minimum $k$-cut with high probability. (The smaller the constant $c$, the better.)

2. How does your algorithm relate to the preceding statement that $k$-cut is NP-Hard when $k$ is part of the input?

**Exercise 21.9.** Consider the minimum cut problem in undirected graphs. For $\alpha \geq 1$, we say that a cut $C$ is an $\alpha$-approximate minimum cut if its capacity is at most $\alpha$ times the capacity of the minimum cut.

1. Let $C$ be an $\alpha$-approximate minimum cut. Prove that the random-contractions algorithm returns $C$ with probability $\geq 1/\binom{n}{\lceil 2\alpha \rceil}$.

2. Show the number of $\alpha$-approximate minimum cuts is at most

$$\binom{n}{\lceil 2\alpha \rceil}.$$ 

**Exercise 21.10.** 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.

For each light bulb $L_i$, there are 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
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, or (b) prove that a polynomial time algorithm implies a polynomial time algorithm for SAT.

1. (7 points) Decide if there exists a way to flip the switches to turn on all the light bulbs.

2. (3 points) Decide if there exists a way to flip the switches to turn on at least three-fourths of the light bulbs.
Chapter 22

Hashing and Heavy Hitters

22.1 Trends

Google has an interesting web page called “Google trends”, which tracks surging search queries around the world in real time. Right now, there is 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

\(^1\)Maybe 7 billion? See https://www.internetlivestats.com/google-search-statistics/
how to 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.

### 22.2 Streaming

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. The algorithm cannot simply write down everything and solve the problem offline, because it is restricted to using very little space compared to the input size. For example, if there are $n$ items in the stream, then the algorithm might be restricted to $n^{1/4}$ space, or even better, $O(\log n)$ space. Because 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 store.

We formalize the heavy hitters problem as follows. We have $n$ elements in a stream $e_1, \ldots, e_n$, where each element is from some large universe $\{1, \ldots, m\}$. 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 $n$ elements, the relative frequency of an element is the total frequency divided by $n$.

Ideally we could keep track of the exact frequencies of all the elements. This is impossible with less than $\min\{m, n\}$ bits\(^2\). Here, and unlike standard algorithmic settings, $O(n)$ or $O(m)$ bits is considered far too expense. We have entered an algorithmic regime where simple and exact computations cannot be taken for granted. Our first step, then, is to identify some new problems that are both tractable and useful. We may have to relinquish exactness and consider approximations, where we allow for some error that we can analyze and control. And, last but not least, we will 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 $\binom{n+m-1}{m-1}$ ways to make a frequency
Given a fixed parameter \( \epsilon \in (0, 1) \), an \( \epsilon \)-heavy hitter is an element with relative frequency \( \geq \epsilon \). The heavy hitters problem is to identify all of the \( \epsilon \)-heavy hitters for an input parameter \( \epsilon > 0 \). Note that there can only be \( (1/\epsilon) \)-many \( \epsilon \)-heavy hitters, which preserves some hope that we can identify all of them with space proportional to \( 1/\epsilon \), rather than \( n \).

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 \( \epsilon \). Equivalently, we want to estimate the absolute frequency of each element up to an \( \epsilon n \)-additive factor. At first it might seem impossible to estimate \( n \)-many counts with \( o(n) \) space. We have already argued that \( n \) exact counters is impossible. Allowing for \( \epsilon \)-relative error, however, means that \( 0 \) is a satisfactory estimate for all but at most \( \lfloor 1/\epsilon \rfloor \) elements. That is, the room for approximation is

If we can estimate the absolute frequency of each element up to additive error \( \epsilon \), then one can find all \((3\epsilon)\)-heavy hitters by considering all of the elements with estimated value at least \( 2\epsilon \). Such a list might also include \( 1/\epsilon \) extra elements who have relative frequency \( < 3\epsilon \), but still have frequency \( > 1/\epsilon \). 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 \textit{a priori} which of the elements are the \( \epsilon \)-heavy hitters, then \( \epsilon \)-frequency estimation is trivial. Namely, we would maintain a counter for each of the \( \lfloor 1/\epsilon \rfloor \) heavy hitters. All the other elements are assigned frequency 0. Of course this is an absurd algorithm, 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 obtain a similar effect.

---

vector of \( m \) items where the total sum is \( n \). (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 \( {m+n-1 \choose m-1} \) outcomes, so we must have \( 2^k \geq {m+n-1 \choose m-1} \), hence \( k \geq \log \left( {m+n-1 \choose m-1} \right) \). Here \( \log \) denotes \( \log_2 \). We also have

\[
{m+n-1 \choose m-1} = \left( \frac{m+n-1}{m-1} \right)^{m-1}
\text{ and also } {m+n-1 \choose m} = \left( \frac{m+n-1}{n} \right)^n 
\]

Thus

\[
k \geq (m-1) \log \left( 1 + \frac{n}{m-1} \right) \text{ and } k \geq n \log \left( 1 + \frac{m-1}{n} \right).
\]

Slightly better low bounds can be obtained via Stirling’s approximation, which is also related to entropy.
22.3 New Probabilistic Tools

In this lecture we introduce two basic and useful probabilistic tools: hash functions, and Markov’s inequality.

22.3.1 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 will discuss hash tables in detail in the following chapter. We will also use hash functions to solve the heavy-hitters problem; let us now define them mathematically.

Loosely speaking, a hash function is a randomly constructed function \( h : \mathbb{M} \to \mathbb{K} \) where the values \( h(i) \) are (in a qualified sense) randomly distributed through \( k \). A collision is a pair of distinct indices \( i_1 \neq i_2 \in \mathbb{M} \) such that \( h(i_1) = h(i_2) \). In most applications, \( m \) is much (much, much) larger than \( k \). In this case, there are necessarily many “collisions” (see exercise 23.1). 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 \( i \in \mathbb{M} \), a value \( h(i) \in \mathbb{K} \) independently and uniformly at random. This produces an “ideal hash function”, defined as follows.

**Definition 22.1.** An ideal hash function \( h : \mathbb{M} \to \mathbb{K} \) is a uniformly random function \( h : \mathbb{M} \to \mathbb{K} \). That is, each \( h(i) \) is drawn from \( \mathbb{K} \) independently uniformly at random.

An ideal hash function \( h \) is particularly easy to reason about. For example, for every input \( i \) and possible output \( j \in \mathbb{K} \), we have

\[
P[h(i) = j] = \frac{1}{k}.
\]

We also have, for any \( \ell \) distinct inputs \( i_1, \ldots, i_\ell \in \mathbb{M} \) and \( \ell \) possible outputs \( j_1, \ldots, j_\ell \in \mathbb{K} \), we have

\[
P[h(i_1) = j_1, h(i_2) = j_2, \ldots, h(i_\ell) = j_\ell] = \frac{1}{n^{\ell}}.
\]

Ideal hash functions are a good model to keep in mind when designing randomized algorithms. In reality, however, ideal hash functions are very expensive to make and store. Indeed, one has to have \( m \log k \) bits to be able to describe all of the possible functions from \( \mathbb{M} \) to \( \mathbb{K} \) (as there are \( k^m \) such functions, and we must
pay the logarithm of this quantity). This is particularly ill-suited to our streaming setting where \( m \) is astronomical and the whole point 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 collision probabilities”, in the following sense.

**Definition 22.2.** A random hash function \( h : [m] \to [k] \) is universal if for any distinct indices \( i_1 \neq i_2 \in [m] \), we have

\[
P[h(i_1) = h(i_2)] = \frac{1}{k}.
\]

In contrast to ideal hash functions, universals hash functions can be constructed compactly by the following simple algorithm.

new-universal-hash-function(\( m, k \))

1. Pick a prime number \( p \) greater than \( m \).
2. Draw an integer \( a \in \{1, \ldots, p - 1\} \) uniformly at random.
3. Draw an integer \( b \in \{0, \ldots, p - 1\} \) uniformly at random.
4. Return the function

\[
h(x) = ((ax + b) \mod p) \mod k.
\]

**Theorem 22.3.** Consider the randomly constructed function \( h : [m] \to [k] \)

\[
h(x) = (ax + b \mod p) \mod k,
\]

where \( p \) is a prime number larger than \( m \), \( a \in \{1, \ldots, p - 1\} \) is drawn uniformly at random, and \( b \in \{0, \ldots, p\} \) is drawn uniformly at random. Then \( h \) is a 2-universal hash function.

The proof of theorem 22.3 is given as exercise 22.3. Instead we will return to the heavy hitters problem and focus on how to use these universal hash functions.

### 22.3.2 Markov’s inequality

Consider the following fact.

**Fact 22.4.** Let \( X \geq 0 \) be a nonnegative random variable. Then

\[
P[X \geq 2 \mathbb{E}[X]] \leq \frac{1}{2}.
\]
Fact 22.4 describes mathematically a fact that is intuitive and obvious – at least, from a counterfactual perspective: a nonnegative random variable cannot be more than two times its average more than half the time. For example, no more than 50% of the population can make two times more than the average income. No more than 50% of NBA players can have a scoring average more than two times the league-wide average. No more than 50% of the class can score more than two times the average score on an exam.\footnote{This last statement is only interesting if the average score is less than 50\%, which sometimes happens.} Another way to put it: if a nonnegative random variable has value $\geq 10$ at least half of the time, then of course its average is $\geq 5$.

A full proof of Fact 22.4 is as follows.

$$
\begin{align*}
\mathbb{E}[X] & \overset{(a)}{=} \sum_x P[X = x] \cdot x \\ & \overset{(b)}{=} \sum_{x: x \geq 2 \mathbb{E}[X]} P[X = x] \cdot x \\ & \overset{(c)}{\geq} 2 \mathbb{E}[X] \sum_{x: x \geq 2 \mathbb{E}[X]} P[X = x] \overset{(d)}{=} 2 \mathbb{E}[X] P[X \geq 2 \mathbb{E}[X]].
\end{align*}
$$

Here (a) is by definition of $\mathbb{E}[X]$. (b) drops some of the nonnegative values where $x < 2 \mathbb{E}[X]$. (c) is because all the values $x$ remaining in the sum are $\geq 2 \mathbb{E}[X]$. (d) is by definition of $P[X \geq 2 \mathbb{E}[X]]$. Rearranging the inequality gives the desired result.

One can extract a more general relation, called Markov’s inequality.

**Theorem 22.5** (Markov’s inequality). Let $X \geq 0$ be a nonnegative random variable, and let $\alpha > 1$. Then

$$
P[X \geq \alpha \mathbb{E}[X]] \leq \frac{1}{\alpha}.
$$

The reader is asked to prove Markov’s inequality (for general $\alpha$) in exercise 22.2.

22.4 Using hashing to approximate frequencies

Let us now return to the frequency estimation problem. We have elements from the set $[m]$ coming in a stream of $n$ total elements. The goal is to estimate the absolute frequency of each element up to an $\epsilon n$-additive factor. The crux of the problem is that we only want our total space usage to be (more or less) independent of the length of the stream, $n$, or the number of elements, $m$. 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...
22. Hashing and Heavy Hitters

22.4. Using hashing to approximate frequencies

Kent Quanrud
Spring 2022

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 $h : \{1, \ldots, m\} \rightarrow \{1, \ldots, w\}$. 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 $f_e$.

The 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 $h : \{1, \ldots, m\} \rightarrow \{1, \ldots, w\}$

3. for each item $e$ in the stream

   a. $A[h(e)] \leftarrow A[h(e)] + 1$

$A[h(e)]$ 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 $\epsilon n$ to $A[h(e)]$. Here the intuition is that the “noise” coming from other frequencies is spread out by the hash function over $\epsilon/2$ entries, so we would only expect $\epsilon n/2$ error for each element $e$. To translate “expected error” to “probability of error”, we use Markov’s inequality, as follows.

**Lemma 22.6.** For each element $e$, with probability $\geq 1/2$, we have

$$f_e \leq A[h(e)] \leq f_e + \epsilon n.$$  

**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

$$E[A[h(e)]] - f_e \overset{(a)}{=} \sum_{d \neq e} f_d P[h(d) = h(e)] \overset{(b)}{=} n/w \leq \frac{\epsilon}{2} n. \quad (22.1)$$

Here (a) is by linearity of expectation. (b) is because $h$ is universal. Now we have

$$P[A[h(e)] \geq f_e + \epsilon n] \overset{(c)}{=} P[A[h(e)] - f_e \geq 2 E[A[h(e)] - f_e] \overset{(d)}{=} \frac{1}{2}$$

Here (c) plugs in the inequality obtained in (22.1). (d) applies Markov’s inequality, where we note that $A[h(e)] - f_e \geq 0$.  

339
Given that it is impossible to track frequencies exactly in sublinear space, it is surprising 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.

### 22.5 Amplification

Section 22.4 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 $\epsilon n$-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 $1/2^{100} \approx 0.0000000000000000000000000000007886\ldots$.

The point is that independent trials magnify the probability of at least one success exponentially. For a specified probability of error $\delta \in (0,1)$, the algorithm count-min-sketch below makes $\lceil \log 1/\delta \rceil$ independent instances of hashed-counters($\epsilon$). 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.

**count-min-sketch($\epsilon > 0, \delta > 0$)**

1. build $d = \lceil \log 1/\delta \rceil$ instances $(A_1, h_1), \ldots, (A_d, h_d)$ of hashed-counters($\epsilon$) over the stream

2. to query an element $e$:
   - return $\min_{i=1,\ldots,d} A_i[h_i(e)]$

**Lemma 22.7.** For each element $e$, with probability $\geq 1 - \delta$, we have

$$\min_{i=1,\ldots,d} A_i[h_i(e)] \leq f_e + \epsilon n.$$

**Proof.** We have

$$\Pr \left[ \min_{i=1,\ldots,d} A_i[h_i(e)] > f_e + \epsilon n \right] \leq \prod_{i=1}^{d} \Pr \left[ A_i[h_i(e)] > f_e + \epsilon n \right] \leq \frac{1}{2^d} = \delta.$$
22. Hashing and Heavy Hitters

22.6. Extensions

Here (a) is by independence of each $A_i[h_i(e)]$. (b) is by lemma 22.6.

For $\delta$ set to a polynomial of $1/n$, the probability of error becomes low enough to take a union bound over all elements, as follows.

**Theorem 22.8.** With probability at least $1 - 1/n$, over a stream of length $n$, count-min-sketch($\epsilon, 1/n^2$) overestimates the total frequency of each element with additive error at most $\epsilon n$ and total space $O(\log(n)/\epsilon)$.

**Proof.** By lemma 22.7, 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 $\leq 1/n$.

22.6 Extensions

22.6.1 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-sketch has the convenient property of being a sketch. To handle multiple streams, we have an instance of count-min-sketch 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-sketch over the combined streams.
22.6.2 Turnstile streams

Consider the more general model where each item in the stream consists of an element $e$ and a value $\Delta$, signifying that we should increase the frequency count for $e$, $f_e$ by $\Delta$. $\Delta$ 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-sketch adapts immediately to turnstile stream, by simply adding $\Delta$ to $A_i[h_i(e)]$ for each instance $(A_i, h_i)$ of hashed-counters. The additive error is now $\epsilon$ times the sum of all $\Delta$'s in the stream.

22.7 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-sketch 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-sketch does not give unbiased estimates of the counters. Instead, count-min-sketch tries to be within a prescribed error with high probability. It is consistent. It is more important to be consistent than unbiased, since we can (psychologically) adjust for the bias. Many real-world apparatus are designed on this principle.

22.8 Exercises

Exercise 22.1. Let $h : [m] \rightarrow [k]$ be any fixed function.

1. Prove that the number of collisions is
   \[ \geq \frac{m(m-n)}{2n} \]

2. Show that the above inequality is tight when $n$ divides $m$.

Exercise 22.2. In section 22.3.2, we proved Markov’s inequality for the special case of $\alpha = 2$. Prove Markov’s inequality for general $\alpha > 1$.

Exercise 22.3. Show that the construction given in section 22.3.1 is indeed a universal hash function, using the steps listed below.

To recall the construction, we randomly construct a function $h : [m] \rightarrow [k]$ as follows. First, let $p$ be any prime number $> m$. 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
\[
h(x) = ((ax + b) \mod p) \mod n.
\]

1. Let $x_1, x_2 \in [m]$ with $x_1 \neq x_2$, and let $c_1, c_2 \in \{0, \ldots, p-1\}$ with $c_1 \neq c_2$. Show that the system of equations
   \[
   ax_1 + b = c_1 \mod p \\
   ax_2 + b = c_2 \mod p
   \]
   uniquely determines $a$ and $b$.

• Step 1 implies that the map $(a, b) \mapsto (ax_1 + b \mod p, ax_2 + b \mod p)$ is a bijection.

\footnote{Here it is helpful to know that division is well-defined in the set of integers $\mod p$ when $p$ is prime. In particular, “$a/b$” is defined as the unique integer $c$ such that $bc = a$.}
2. Let \( x_1, x_2 \in [m] \) with \( x_1 \neq x_2 \), and let \( c_1, c_2 \in \{0, \ldots, p-1\} \) with \( c_1 \neq c_2 \).

Show that

\[ \Pr[ax_1 + b = c_1, ax_2 + b = c_2] = \frac{1}{p(p-1)}. \]

(Here the randomness is over the uniformly random choices of \( a \) and \( b \).)

3. Fix \( x_1, x_2 \in [m] \) with \( x_1 \neq x_2 \), and \( c_1 \in \{0, \ldots, p-1\} \).

Show that

\[ \sum_{c_2 \in \{1, \ldots, p\} \atop c_2 \neq c_1} \Pr[ax + b = c_1, ax + b = c_2] \leq \frac{1}{m}. \]

*Hint: You may want to show that the number of values \( c_2 \) such that \( c_1 = c_2 \mod m \) is \( \leq \frac{p-1}{m} \).*

4. Finally, show that \( \Pr[h(x_1) = h(x_2)] \leq \frac{1}{m} \).

**Exercise 22.4.** The count-min-sketch data structure allows us to estimate the relative frequency of each element up to an \( \epsilon \)-additive factor with probability of error \( \leq 1/poly(n) \) with \( O(\log(n)/\epsilon) \) space. The original motivation, however, was to also obtain a list of \( \epsilon \)-heavy hitters. Explain how to adjust the count-min-sketch to also maintain a collection that contains all of the \( \epsilon \)-heavy hitters while holding onto at most \( O(1/\epsilon) \) items from the stream at any point in time, with probability of error \( \leq 1/n^2 \). Here your algorithm may also report a limited number of items with relative frequency in the range \([\epsilon/2, \epsilon)\), but no items with relative frequency less than \( \epsilon/2 \).

**Exercise 22.5.** In this exercise, we develop a refined analysis that can reduce the additive error substantially in many real settings.

Let \( S \) denote the sum of frequency counts of all elements that are *not* \( \epsilon \)-heavy hitters:

\[ S = \sum_{e : p_e < \epsilon} f_e. \]

Note that \( S \leq n \), and \( S \) might be much less than \( n \) when the stream is dominated by heavy hitters.

Show that, by adjusting count-min-sketch(\( \epsilon, 1/n^2 \)) slightly by increasing \( w \) to \([4/\epsilon]\), the additive error for every element is at most \( \epsilon S \) with probability of error \( \leq 1/n \).
Chapter 23

Hash tables and linear probing

23.1 Dictionaries

Nowadays it is difficult to imagine programming without dictionaries and maps. These data structures are defined primarily by the following two operations.

1. \texttt{set}(k,v): Associate the value \(v\) with the key \(k\).

2. \texttt{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 \texttt{dict} (short for dictionary) in Python are given in fig. 23.1.

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 \(i\)th 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.\(^1\)

\(^1\)Sometimes, constant factors matter.
23. Hash tables and linear probing

23.1. Dictionaries

Kent Quanrud
Spring 2022

Java Map Interface

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key</td>
<td>String</td>
<td>The key in the map.</td>
</tr>
<tr>
<td>value</td>
<td>Object</td>
<td>The value associated with the key.</td>
</tr>
</tbody>
</table>

Python Dictionary

5.5. Dictionaries

Another useful data type built into Python is the dictionary. Dictionaries are sometimes found in other languages as “associative memory” or “associative arrays.” Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable object (strings, numbers, tuples, or other dictionaries). One important use of dictionaries is to associate a value with its key. The following dictionary provides a good example of how a dictionary can be used to store and retrieve information.

```python
# Example usage of a dictionary

# Create a dictionary
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}

# Access dictionary values
print(my_dict['name'])  # Output: John
print(my_dict['age'])  # Output: 30
print(my_dict['city'])  # Output: New York

# Update dictionary values
my_dict['age'] = 31
print(my_dict['age'])  # Output: 31

# Add a new key-value pair
my_dict['country'] = 'USA'
print(my_dict)  # Output: {'name': 'John', 'age': 31, 'city': 'New York', 'country': 'USA'}

# Remove a key-value pair
del my_dict['city']
print(my_dict)  # Output: {'name': 'John', 'age': 31, 'country': 'USA'}
```

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 size $n + 1$ and copying everything over. Even more problematic is the case where the keys are not neatly organized to be a 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 would not want to allocate an array so big. One could alternatively reindex 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 using a hash table. However, the time complexity of insertions and deletions is higher (logarithmic), and the space usage is higher as well (since the tree may become unbalanced).
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. The celebrated splay tree data structure by Sleator and Tarjan [ST85] 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 than the $O(1)$ time of arrays. They are more complicated to implement, and require a lot of pointer chasing, which leads to many cache misses on the CPU.\(^3\)

We instead consider simpler randomized approaches to the dictionary problem; namely, hash tables. Hash tables combines 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 be easily be worth it. Hash tables are generally based on the following framework. Suppose that there are $n$ keys $k_1, \ldots, k_n$ from the set of integers $[U] = \{1, \ldots, U\}$, 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 $h : [U] \rightarrow [m]$. 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, sometimes simple and sometimes clever, to account for collisions, such as the following.

1. Make $\ell$ so large that even a single collision is unlikely. exercise 23.1 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 slot $j$. We study this first in section 23.2.

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 23.3.

---

\(^{2}\)Sometimes, log factors matter.

\(^{3}\)This last point can be helped to some extent by cache-oblivious versions.
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 \textit{cuckoo hashing}.

5. Suppose we want to insert a key \( k \) and \( A[h(k)] \) is occupied. We 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 \textit{linear probing}, and will be the topic of the second half of our discussion.

Ignoring the possibility of collisions, these hash tables have the appeal of potentially being \textit{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 \textit{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 \textit{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 negligible. (This is the same strategy used, for example, in Java’s ubiquitous ArrayList data structure.) We note that there are some \textit{distributed} computational settings where one wants to maintain a \textit{distributed dictionary}, and where simply rehashing items becomes expensive and impractical. We refer the reader to a technique called \textit{consistent hashing} that addresses this challenge [KLL+97]. Distributed dictionaries are particularly useful for caching on the web.

### 23.2 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-sketch data structure previously discussed.

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 larger and smaller $m$.)

We randomly construct a hash function $h : [U] \rightarrow [m]$. Here we analyze the setting where $h$ is a universal hash function, but later we will also explore stronger notions of independence. Exercise 23.2 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. 23.2. 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 23.1.** A randomly constructed function $h : [n] \rightarrow [m]$ is universal if, for any two indices $i_1 \neq i_2$, we have

$$P[h(i_1) = h(i_2)] = \frac{1}{m}.$$

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.

**Theorem 23.2.** Consider chaining with $n$ keys, an array $A[1, \ldots, m]$, and a universal hash function $h : [U] \rightarrow [m]$. Then each get and set takes $O(1 + n/m)$ time in
23. Hash tables and linear probing

23.3 4-wise independence

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

\[
E[|k' : h(k') = h(k)|] \overset{(a)}{=} \sum_{k' \neq k} P[h(k') = h(k)] \overset{(b)}{=} \sum_{k' \neq k} \frac{1}{m} \overset{(c)}{=} \frac{n-1}{m}
\]

Here (a) is by linearity of expectation. (b) is by universality. (c) is because there are \( n-1 \) other keys.

23.3 4-wise independence

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

\[
P[X_{i_1} = y_1, X_{i_2} = y_2, \ldots, X_{i_k} = y_k] = P[X_{i_1} = y_1] P[X_{i_2} = y_2] \cdots P[X_{i_k} = y_k].
\]

Lemma 23.3. Let \( X_1, \ldots, X_k \) be \( k \)-wise independent random variables. Then

\[
E[X_1 X_2 \cdots X_k] = E[X_1] E[X_2] \cdots E[X_k].
\]

Before proving lemma 23.3, let us give a simple example where \( k \)-wise independence matters. Let \( X_1, \ldots, X_k \in \{0, 1\} \) 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 come up heads is \( 1/2^k \), so \( E[X_1 \cdots X_k] = 1/2^k \).

2. Suppose each \( X_i \) was based on the same coin toss. That is, \( X_1 = \cdots = X_k \); they are certainly not 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 \( E[X_1 \cdots X_k] = 1/2 \).

Here there is an exponential gap between independent and non-independent coin tosses.
Proof of lemma 23.3. We have

\[
E[X_1 X_2 \cdots X_k] = \sum_{y_1, y_2, \ldots, y_k} y_1 y_2 \cdots y_k P[X_1 = y_1, X_2 = y_2, \ldots, X_k = y_k]
\]

by definition of expectation (a). (b) is by 4-wise independence. (c) is by definition of expectation, for each \(X_i\).

A concentration inequality for 4-wise independent sums. Recall Markov’s inequality: the probability of a nonnegative random variable, \(X \geq 0\), being at least twice its expected value, is at most \(1/2\). Here we consider an analogous but stronger inequality for sums of 4-wise independent random variables.

Lemma 23.4. Let \(X_1, X_2, \ldots, X_n \in \{0, 1\}\) be 4-wise independent variables where for each \(i\), \(E[X_i] = p\). Let \(\mu = pn = E[\sum_{i=1}^{n} X_i]\). Then for any \(\beta > 0\),

\[
P\left[\sum_{i=1}^{n} X_i \geq \mu + \beta\right] \leq \frac{\mu + 3\mu^2}{\beta^4}.
\]

Before proving lemma 23.4, let us compare it to Markov’s inequality. Given \(X_1, \ldots, X_n\) and \(\mu\) as in lemma 23.4, Markov’s inequality say that for all \(\alpha > 0\),

\[
P[X_1 + \cdots + X_n \geq (1 + \alpha)\mu] \leq \frac{1}{1 + \alpha}.
\]

(23.1)

Lemma 23.4 says that

\[
P[X_1 + \cdots + X_n \geq (1 + \alpha)\mu] \leq \frac{\mu + 3\mu^2}{\alpha^4\mu^4} \leq \frac{3}{\alpha^2\mu^4}.
\]

(23.2)

Here (d) applies lemma 23.4 with \(\beta = \alpha \mu\). Compare the RHS of (23.1) with the RHS of (23.2). In (23.2), the upper bound is decreasing in \(\alpha\) at a \(1/\alpha^4\), compared to \(1/\alpha\) in (23.1). Moreover, (23.2) is decreasing in the expected value \(\mu\) at a rate of \(1/\mu^2\). That is, the greater the mean, the smaller the probability of deviated from the mean. This is our first example of a concentration inequality. We will soon see why this helpful when we return to designing hash tables after proving lemma 23.4.
Proof of lemma 23.4. We have

\[ P \left[ \sum_{i=1}^{n} X_i \geq \mu + \beta \right] = P \left[ \sum_{i=1}^{n} X_i - \mu \geq \beta \right] \overset{(a)}{=} P \left[ \left( \sum_{i=1}^{n} X_i - \mu \right)^4 \geq \beta^4 \right] \]

\[ \overset{(b)}{\leq} \frac{E \left[ \left( \sum_{i=1}^{n} X_i - \mu \right)^4 \right]}{(\beta - \mu)^4}. \]

The key step is (a), where we raise both sides to the fourth power. (b) is by Markov’s inequality. We claim that

\[ E \left[ \left( \sum_{i=1}^{n} X_i - \mu \right)^4 \right] \leq \mu + 3 \mu^2, \]

which would complete the proof. We first have

\[ E \left[ \left( \sum_{i=1}^{n} X_i - \mu \right)^4 \right] = E \left[ \left( \sum_{i=1}^{n} (X_i - p) \right)^4 \right] \]

because \( \mu = pn \). Now, \( \sum_{i=1}^{n} (X_i - p) \) expands out to the sum

\[ \sum_{i=1}^{n} (X_i - p)^4 + \binom{4}{2} \sum_{i<j} (X_i - p)^2 (X_j - p)^2 + \left( \sum \text{monomials w/ some } (X_i - p) \text{ w/ degree 1} \right). \]  \( (23.3) \)

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

\[ E \left[ (X_i - p)^4 \right] = p(1-p)^4 + (1-p)p^4 \leq p(1-p). \]

2. For each \( i \neq j \), we have

\[ E \left[ (X_i - p)^2 (X_j - p)^2 \right] \overset{(c)}{=} E \left[ (X_i - p)^2 \right] E \left[ (X_j - p)^2 \right] \overset{(d)}{\leq} p^2(1-p)^2. \]

Here (c) is because of pairwise independence. (d) is because

\[ E \left[ (X_i - p)^2 \right] = p(1-p)^2 + (1-p)p^2 \leq p(1-p). \]
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,

\[ E[(X_1 - p_1)^3(X_2 - p_2)] = 0, \]

where \( (e) \) is by pairwise independence, and \( (f) \) is because \( E[X_2 - p_2] = 0. \)

Plugging back in to above, we have

\[ E\left[ \left( \sum_{i=1}^{n} X_i - \mu \right)^4 \right] = np(1-p) + \binom{n}{4} \left(p^2(1-p)^2\right) \leq np + 3(np)^2, \]

as desired.

Remark 23.5. The claim would hold even for \( X_i \) not identically distributed (as long as they are 4-wise independent and are each in \([0,1]\)). This restricted setting suffices for our applications and simplifies the exposition.

### 23.3.1 Hashing with linked lists with 4-wise independent functions

Recall the hash tables with chaining. If we use a 4-wise independent hash function rather than a universal hash function, then we get better bounds. In general, the bounds improve with more independence.

**Theorem 23.6.** Suppose we throw \( n \) balls into \( n \) bins, where each ball is goes in a bin uniformly at random, and the distribution of balls into bins is 4-wise independent. For any fixed \( \delta \in (0,1) \), the max-load is at most \( O(n^{1/4}/\delta^{1/4}) \) with probability at least 1 \(-\delta\).

The proof is left as exercise 23.4.
23.4 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 $h : \{1, \ldots, U\} \rightarrow \{1, \ldots, m\}$. 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 23.7.** 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 23.7 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 23.7, 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 23.7, 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 1964 [Knu63]. Knuth is sometimes called the “father of the analysis of algorithms”, and he is credited with formalizing the subject and popularizing \( O \)-notation.\(^5\) As Knuth tells it\(^6\), this was the first algorithm he ever formally analyzed, and therefore, arguably, the first algorithm that anyone has ever (so) formally analyzed. He showed that for ideal hash functions, the expected time of any operation is \( O((n/(m-n))^2) \); in particular, a constant, whenever \( m \) is bigger than \( n \) 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.

Post-Knuth, there remained a question of how much independence was required to get constant running time in expectation. Around 1990, Schmidt and Siegel [SS89; SS90] showed that \( O(\log n) \)-wise independence sufficed\(^7\). 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!\(^8\)

Here we give a simplified analysis of the result of [PPR09]. 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. Other proofs of the constant time bound can be found in [Nel16; Tho15].

**Theorem 23.8.** Let \( h \) be 5-wise independent. For \( m \geq 8n \), linear probing takes expected constant time per operation.

**Proof.** 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

\(^{5}\)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](https://en.wikipedia.org/wiki/Donald_Knuth).

\(^{6}\)See for example this interview: [https://www.youtube.com/watch?v=Wp7GAKLSGrI](https://www.youtube.com/watch?v=Wp7GAKLSGrI).

\(^{7}\)Alan Siegel taught me algorithms.

\(^{8}\)Here \( 5 ! \) denotes the factorial of 5.
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, and $x$ occupies a cell in the run containing $A[h(x)]$. Each operation with an item $x$ takes time at most proportional to the length of the run containing $x$.

Let $i = h(x)$, and let $R$ be the run at index $i$. Note that $R$ and its length $|R|$ are random. We have

$$
E\left[ \text{running time} \right] \leq E[|R|] = \sum_{\ell=1}^{n} \ell \cdot P[|R| = \ell] \\
\leq \sum_{k=1}^{1 \log n} 2^k \cdot P[2^k \leq |R| \leq 2^{k-1}].
$$

(23.4)

For each $k \in \mathbb{N}$, let

$$
I_k = [i - (2^k - 1), i + 2^k - 1]
$$

be the interval of length $2^{k+1} - 1$ centered at $i$. If $R$ has length $|R| < 2^k$, and contains $i$, then $R$ must be contained in $I_{k+1}$. Moreover, if $R$ has length $> 2^{k-1}$, then at least $2^{k-1}$ items other than $x$ hash to $R$. Thus for each $k$, we have

$$
P\left[ 2^{k-1} < |R| \leq 2^k \right] \leq P\left[ \text{at least } 2^{k-1} \text{ other items hash into } I_k \right].
$$

Since $h$ is 5-wise independent, conditional on $h(x) = i$, the remaining hash values are 4-wise independent, and each lands in $I_k$ with probability $p = |I_k|/m$. Let

$$
\mu = E\left[ \frac{\# \text{ other items hashing into } I_k}{|I_k|} \right].
$$
We have

\[ \mu = \frac{|I_k|n}{m} \leq 2^{k-2}, \]

where (a) is because \( m \geq 8n \). We have

\[ \mathbb{P}\left( \text{(# other items hashing into } I_k) > 2^{k-1} \right) \leq \frac{4 \max \{\mu, \mu^2\}}{(2^{k-1} - \mu)^4} \leq \frac{4 \left(2^{k-2}\right)^2}{(2^{k-2})^4} \leq \frac{1}{2^{2k-6}}. \]

Here (b) is by lemma 23.4. Plugging back into RHS(23.4) above, we have

\[ \mathbb{E}\left[ \text{running time (up to constants)} \right] \leq \text{RHS}(23.4) \leq \sum_{k=1}^{\lceil \log n \rceil} 2^k \cdot \frac{1}{2^{2k-6}} = 2^6 \sum_{k=1}^{\lceil \log n \rceil} \frac{1}{2^k} \leq 2^6. \]

A constant!

### 23.5 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 is also the most 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.
23.6 Exercises

Exercise 23.1. Let \( h : [n] \rightarrow [\ell] \) be a hash function, with \( \ell \geq n \).

1. Suppose \( h \) is an ideal hash function. What is the exact probability that \( h \) has no collisions (i.e., \( h \) is injective)?

2. Suppose \( h \) is a universal hash function. Show that for \( \ell \geq n^2 \), \( h \) has no collisions with probability \( \geq 1/2 - 1/2n \).

Exercise 23.2. Consider the particular case of hash tables with chaining with \( k = 2n \) and an ideal hash function \( h : [n] \rightarrow [k] \).

1. Consider a particular array slot \( A[i] \). Show that the probability that \( A[i] \) has \( \geq \ell \) items hashed to it is

\[
P[ \text{at least } \ell \text{ items being hashed to } A[i] ] \leq \frac{1}{\ell! 2^\ell}.
\]

2. Show that, with probability of error \( \leq 1/n^2 \), the maximum length is at most \( O(\log n) \).

3. Show that, with probability of error \( \leq 1/n^2 \), the maximum length is at most \( O(\log(n)/\log \log n) \) (maybe with a slightly higher hidden constant than in part 2).

Exercise 23.3. 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 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 exercise 23.1). (We may have to retry if there is a collision.) If the total size (summing the lengths of the first array and each of the second arrays) comes out to bigger than (say) \( 5n \), we try again.

1. For each \( i = 1, \ldots, n \), let \( k_i \) be the number of keys that hash to the \( i \)th cell. We have

\[
\text{(sum of array sizes of our data structure)} \leq n + \sum_{i=1}^{n} k_i^2.
\]
Show that
\[ \sum_{i=1}^{n} k_i^2 \leq \text{(total # of ordered pairs of colliding keys in } h_0). \]

(Here an ordered pair is defined by a coordinate pair \((i, j)\), where \(i, j \in [n]\) and are allowed to be equal.)

2. Show that
\[ \mathbb{E}[\text{total # of ordered pairs of colliding keys}] \leq 2n. \]

3. Show that
\[ \mathcal{P}[\text{(sum of all array sizes) > } 5n] < 1/2. \]

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. Note that, due to the reliance on resampling, this approach works better in static settings, when the set of keys is fixed.

**Exercise 23.4.** Prove theorem 23.6.
Chapter 24

Self-adjusting trees

This and the next lecture are about two data structures. The first is called splay-trees, which is a completely different approach to the problem of maintaining binary search trees that perform as well as balanced binary trees. The second is called link-cut trees, which uses splay-trees as a critical ingredient. These two data structures were introduced by Sleator and Tarjan [ST85]. While these two data structures are fairly advanced as far as data structures go, we choose to present them for the following reasons. First, they provide an advanced example of amortized analysis, allowing us to test the techniques developed in the last lecture. Second, the link-cut data structure will be applied in subsequent lectures to problems related to minimum spanning trees and maximum flow. These future lectures will combine ideas from both our recent study of data structures, and the preceding studies of graph algorithms. And besides, they’re amazing results in their own right.

Splay trees are binary search trees that constantly updates itself 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.

24.1 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.

---

1Link-cut trees were previously introduced by Sleator and Tarjan [ST83] with a different implementation; the simpler version here based on splay trees is from [ST85].

2In particular, we wouldn’t expect the reader to memorize all the details of the analysis.
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. 24.1.
24. Self-adjusting trees

24.2 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.

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 24.1. splay(x) takes $O(\log(n))$ amortized time.
Proof. 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. Here we will do the first and third steps, and declare the bounds for each rotation. We will then analyze each rotation separately in subsequent sections, which will complete the overall proof.

We first define the potential of a node $x$ as

$$(\text{potential } x) = \log(\text{number of nodes in the subtree rooted at } x)$$

We define the total potential of a tree (or subtree) $T$ as the sum of potential of its nodes,

$$(\text{total potential}) = \sum_{x \in T} (\text{potential } x).$$

The local rotations have the following guarantees w/r/t the potential functions defined above. Here we only state the bounds and will prove them separately in subsequent sections.

Lemma 24.2. Consider a single zig-zag (or zag-zig) making a node $x$ the root of a subtree. Then

$$2 + \Delta(\text{total potential}) \leq 2\Delta(\text{potential } x).$$

Lemma 24.3. Consider any zig-zig or zag-zag rotation making a node $x$ the root of a subtree. Then

$$2 + \Delta(\text{total potential}) \leq 3\Delta(\text{potential } x).$$

Lemma 24.4. Consider any zig or zag rotation making a node $x$ the root of a subtree. Then

$$\Delta(\text{total potential}) \leq \Delta(\text{potential } x).$$

Suppose that all three lemma’s hold holds, and we splay $x$ to the top. We have

$$2(\text{total } \# \text{ rotations}) + \Delta(\text{total potential}) \leq 1 + 3\Delta(\text{potential } x) \leq O(\log n)$$

as desired. (a) applies lemma 24.2 or lemma 24.3 (where we point out that the inequality $2 + \Delta(\text{total potential}) \leq 3\Delta(\text{potential } x)$ is valid for any double rotation), and lemma 24.4 to the last rotation if necessary. (b) observes that $\Delta(\text{potential } x) \leq (\text{potential } x) \leq O(\log n)$. ■

It remains to prove lemma 24.2, lemma 24.3, and lemma 24.4.
Zig-Zag

Lemma 24.2. Consider a single zig-zag (or zag-zig) making a node \( x \) the root of a subtree. Then

\[
2 + \Delta(\text{total potential}) \leq 2\Delta(\text{potential } x).
\]

Proof. Let us label the relevant subtrees and nodes as follows.

We let \( A, \ldots, D \) also denote the size of the respective subtrees. We have

\[
\Delta(\text{total potential}) = \log(1 + A + B) + \log(1 + C + D) - \log(2 + A + B + C) - \log(1 + B + C).
\]

We also have

\[
\Delta(\text{potential } x) = \log(3 + A + B + C + D) - \log(1 + B + C).
\]

Now,

\[
\log(1 + A + B) + \log(1 + C + D) \overset{(a)}{\leq} 2 \log \left( \frac{2 + A + B + C + D}{2} \right) = 2 \log(2 + A + B + C + D) - 2
\]

by (a) concavity. Thus

\[
\Delta(\text{total potential}) + 2 \\
\leq 2 \log(2 + A + B + C + D) - \log(2 + A + B + C) - \log(1 + B + C) \\
\leq 2\Delta(\text{potential } x),
\]

as desired. \( \blacksquare \)

Zig-zig (and zag-zag)

Lemma 24.3. Consider any zig-zig or zag-zag rotation making a node \( x \) the root of a subtree. Then

\[
2 + \Delta(\text{total potential}) \leq 3\Delta(\text{potential } x).
\]
Proof. Let us label the relevant subtrees and nodes as follows.

We let $A, \ldots, D$ also denote the size of the respective subtrees. We have

$$\Delta(\text{total potential}) = \log (2 + B + C + D) + \log (1 + C + D) - \log (2 + A + B + C) - \log (2 + A + B).$$

We also have

$$\Delta(\text{potential } x) = \log (3 + A + B + C + D) - \log (1 + A + B).$$

Observe that

$$\log (2 + B + C + D) - \log (2 + A + B + C) \leq \Delta(\text{potential } x)$$

by monotonicity. This leaves the terms $\log (1 + C + D) - \log (2 + A + B)$. We also have

$$\log (1 + C + D) + \log (1 + A + B) \leq 2 \log \left( \frac{2 + A + B + C + D}{2} \right) \leq 2 \log (2 + A + B + C + D) - 2.$$

Thus

$$2 + \log (1 + C + D) - \log (2 + A + B) \leq 2 \log (2 + A + B + C + D) - \log (1 + A + B) - \log (2 + A + B) \leq 2 \Delta(\text{potential } x).$$

This gives the overall claim that

$$2 + \Delta(\text{total potential}) \leq 3 \Delta(\text{potential } x).$$
Zig and zag

**Lemma 24.4.** Consider any zig or zag rotation making a node $x$ the root of a subtree. Then $\Delta(\text{total potential}) \leq \Delta(\text{potential } x)$.

**Proof.** By symmetry it suffices to analyze the zig operation. We refer to the relevant quantities per the following diagram.

We have

$$\Delta(\text{total potential}) = \log(1 + B + C) - \log(1 + A + B).$$

We also have

$$\Delta(\text{potential } x) = \log(1 + A + B + C) - \log(1 + A + B).$$

The claim follows immediately. ■

This gives the last missing proof from theorem 24.1, establishing the theorem.
Chapter 25

Range Queries and Dynamic Trees

25.1 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 \( \text{max} \) 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 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 than 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 \(i\)th 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.)

### 25.2 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 25.1.** One can support path queries in a rooted tree in $O(\log^2(n))$ 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.

---

**25.3 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, $\text{link}(w, v)$ make $v$ a child of $w$.

**Cut.** Given a vertex $v$ that is not a root, $\text{cut}(v)$ 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.

### 25.3.1 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.

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.

**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.

---

\(^2\)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 [ST85] or [Dem12a] for that analysis.
25. Range Queries and Dynamic Trees

25.3. Dynamic trees

Kent Quanrud
Spring 2022

Link. Recall that \( \text{link}(v, w) \) 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 \( \text{link}(v, w) \), 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 \( \text{find-root}(v) \) 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 \( \text{sum-path}(v) \) by calling \( \text{access}(v) \), and then return the sum in \( v \)'s auxiliary tree. Of course, summation can be replaced with other aggregate operations as discussed in section 25.1.

25.3.2 Analyzing \text{access}

It remains to define and analyze \text{access}. Before defining access, we first introduce a helpful auxiliary function called \text{clip}. \text{clip} takes as input a vertex \( v \). That vertex \( v \) is part of some auxiliary path in the path decomposition. \text{clip} adjusts the path decomposition so that \( v \) is the last vertex in its auxiliary path. Psuedocode for \text{clip} is as follows.

\[
\text{clip}(v)
\]

// 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 auxiliary 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.

\text{clip} consists of one splay operation and a constant number of pointer changes, hence takes \( O(\log n) \) amortized time.

Now we describe \text{access}. \text{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 \( \text{clip} \ v \). 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 \( \text{clip} \ u \), 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, \( \text{clip}(u) \) 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 root of its splay tree. 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 25.2.** 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 25.2). 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 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

\[
\left\{ \text{heavy} \right\} \times \left\{ \text{active} \right\} \times \left\{ \text{light} \right\} \times \left\{ \text{inactive} \right\}
\]

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

\[
C = \text{# changes to the path decomposition},
L = \text{# active light edges created},
L' = \text{# light active edges destroyed},
H = \text{# active heavy edges destroyed},
H' = \text{# heavy active edges created}.
\]

Then

\[
C \leq L + H + L' + H' \leq 2L + H + H' \leq 2(L + H) + n.
\]

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

\[
L + H \leq 2L + 1 \leq O(\log n),
\]

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 heavy was heavy. Thus, excluding changes due to access, we have \(O(\log n)\) with respect to the potential.
25. Range Queries and Dynamic Trees

25.4 Roots, re-rooting, and an application to dynamic disjoint union

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 25.2, 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 25.3.** With $O(n \log n)$ additional overhead, access, link, and cut each take $O\left(\log^{2}(n)\right)$ amortized time.

25.4 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 $v$.

Union($u, v$) corresponds rerooting $v$ and then linking 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

---

\(^3\)Here we restrict ourselves to unions over elements in different sets. This distinction was not necessary before.
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 be able to query connectivity in a dynamically updated graph.
Chapter 26

Dynamic Connectivity and MST’s

This topic, while advanced, was chosen because it sits at the intersection of data structures and graph algorithms.

26.1 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\(^1\) 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.

### 26.1.1 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 26.1.** 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.

### 26.1.2 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.

\(^1\)Other balanced binary search trees would also work.
**Lemma 26.2.** Euler tour trees support link in $O(\log n)$ amortized time.

26.1.3 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 26.3.** Euler tour trees support cut in $O(\log n)$ amortized time.

26.1.4 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.

26.1.5 Re-root.

(Todo... rotate...)
26. Dynamic Connectivity and MST’s

26.2 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 $E'_i \overset{\text{def}}{=} E_i \setminus E_{i-1}$ 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., $\partial(v) \cap E'_i$, where $\partial(v)$ 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 26.1, 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 $F_i$ 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 $\ell$. Then we remove $e$ from $E_\ell$ (and $E_{\ell+1}$, $E_{\ell+2}$, ... ) 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, $e \in F'$. We first remove $e$ from $F_\ell$ (and $F_{\ell+1}, F_{\ell+2}, ...$). Then we look for an edge to replace it. For each level $i$ from $\ell$ 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.

26.2.1 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.
### 26. Dynamic Connectivity and MST's

#### 26.2. Dynamic Connectivity

---

**insert(e)**

```plaintext
/* 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 $\ell$ be the level of $e$
      // remove $e$ from each forest that contains it
   B. call cut($u,v$) on $F_\ell, \ldots, F_L$
   C. for $i = \ell, \ell + 1, \ldots, L$
      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 $b$)
         // $|A| \leq \frac{1}{2}(|A| + |B|) \leq 2^{l-1}$, 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 $F_i, F_{i+1}, \ldots, F_L$ and exit
         b. else  // $f$ is spanned by $A$ and can be pushed to level $i - 1$
            1. add $f$ to $E_{i-1}$
```
26. Dynamic Connectivity and MST’s

26.3 Dynamic MST

For each edge $e$, we define a potential function $\Phi_e$ 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 26.1). Meanwhile the potential increases by $O\left(\frac{\log^2 n}{1}\right)$ since the maximum level is $O(\log n)$. Thus insert takes $O\left(\frac{\log^2 n}{1}\right)$ 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 $O\left(\frac{\log^2(n)}{1}\right)$ 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 $\Phi_d$ 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 $O\left(\frac{\log^2(n)}{1}\right)$.

**Theorem 26.4** ([HLT01]). In a dynamically changing graph, in $O\left(\frac{\log^2 n}{1}\right)$ amortized time per edge insertion and edge deletion, we can support connectivity queries in $O(\log n)$ amortized time.

26.3 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:

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 26.1.

**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 $\ell$ where $F_\ell$ 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 26.5** ([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.

### 26.4 Exercises

---

*In class, we proved $O\left(\log^2(n)\right)$ running times. But a more advanced analysis gives $O(\log n)$.  

---
Exercise 26.1. 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.

26.5 Additional notes and references

Euler trees are from [HK99] and the dynamic connectivity and MST are from [HLT01]. See also the lecture notes and video by Demaine [Dem12b].
Chapter 27

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 [Axh15] 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.

27.1 Preliminaries

Let \( \mathbb{R}^n \) be the set of real-valued, \( n \)-dimensional vectors of the form

\[ x = (x_1, \ldots, x_n) \]
where \( x_1, \ldots, x_n \in \mathbb{R} \). Two vectors \( x, y \in \mathbb{R}^n \) sum to make a vector \( x + y \in \mathbb{R}^n \), defined by

\[
x + y = (x_1 + y_1, \ldots, x_n + y_n).
\]

For \( a \in \mathbb{R} \) and \( x \in \mathbb{R}^n \), we can scale \( x \) by \( a \) to obtain the vector \( ax = (ax_1, \ldots, ax_n) \).

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.

### 27.2 Convex functions

Let \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) 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:

- \( f : \mathbb{R} \rightarrow \mathbb{R} \) defined by \( f(x) = x^2 \).
- \( f : \mathbb{R} \rightarrow \mathbb{R} \) defined by \( f(x) = |x| \).

Some higher dimensional examples include \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) 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 \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) is convex and \( \alpha > 0 \), then \( \alpha f \) is convex. If \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) and \( g : \mathbb{R}^n \rightarrow \mathbb{R} \) are both convex, then \( f + g \) is convex. If \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) is convex, and \( A : \mathbb{R}^m \rightarrow \mathbb{R}^n \) 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

$$f'(x) \in \mathbb{R}^n,$$

where $f'_i(x) = \frac{d}{dx_i} f(x)$ 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

$$g(t) \overset{\text{def}}{=} f(x_t) = f(tx_1 + (1-t)x_0).$$

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 27.1.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be differentiable. Then $f$ is convex iff for all $x, y \in \mathbb{R}^m$,

$$f(y) \geq f(x) + \langle f'(x), y - x \rangle. \quad (27.1)$$
Theorem 27.2. Let
\[ x = \text{arg} \min f(x), \]
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 (27.1). Let \( x_0, x_1 \in \mathbb{R}^n \). Let \( x_t = tx_1 + (1-t)x_0 \). We want to show that \( f(x_t) \leq tf(x_1) + (1-t)f(x_0) \).

**Proof.** Suppose \( f \) is convex. We have
\[
\langle f'(x), y - x \rangle \overset{(a)}{=} \frac{f(x + \lambda(y - x)) - f(x)}{\lambda} = f((1 - \lambda)x + \lambda y) - f(x) = \frac{f((1 - \lambda)x + \lambda y) - f(x)}{\lambda} \\
\leq \frac{(1 - \lambda)f(x) + \lambda f(y) - f(x)}{\lambda} \overset{(c)}{=} f(y) - f(x),
\]
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 (27.1). 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
\[
f(x_1) \geq f(x_t) = f(x_t) + \langle f'(x_t), x_1 - x_t \rangle = f(x_t) + \langle f'(x_t), (1-t)(x_1 - x_0) \rangle,
\]
and
\[
f(x_0) \geq f(x_t) = f(x_t) + \langle f'(x_t), x_0 - x_t \rangle = f(x_t) + \langle f'(x_t), t(x_0 - x_t) \rangle,
\]
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. \( \square \)

**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 27.2.** Let \( f : \mathbb{R}^n \to \mathbb{R} \) be continuously differentiable and convex. A point \( x \) is a global minimum of \( f \) iff \( f'(x) = 0 \).

**Proof.** Suppose \( f'(x) = 0 \). Then for all \( y \), we have
\[
f(y) \overset{(a)}{=} f(x) + \langle f'(x), y - x \rangle \overset{(b)}{=} f(x).
\]
Here (a) is convexity (i.e., lemma 27.1). (b) is because \( f'(x) = 0 \).

Conversely, suppose \( x \) is the global minimum. For all \( u \), we have
\[
\langle f'(x), u \rangle = \lim_{t \to 0} \frac{f(x + tu) - f(x)}{t} \overset{(c)}{=} 0.
\]
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 \( f'(x) = 0 \). \( \square \)
27.3 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

$$|f(x) - f(y)| \leq L\|x - y\|$$

for all $x, y$. The derivative of $f$ is $L$-Lipschitz continuous if we have

$$\|f'(x) - f'(y)\| \leq L\|x - y\|.$$

In the convex optimization literature, $f$ is called $L$-smooth if its derivative is $L$-Lipschitz.

**Lemma 27.3.** 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

$$f(x_1) - f(x_0) - \langle f'(x_0), x_1 - x_0 \rangle \overset{(a)}{=} \int_0^1 \frac{d}{dt}f(x_t)\, dt - \langle f'(x_0), x_1 - x_0 \rangle$$

$$\overset{(b)}{=} \int_0^1 \langle f'(x_t) - f'(x_0), x_1 - x_0 \rangle, dt$$

$$\overset{(c)}{=} \int_0^1 \|f'(x_t) - f'(x_0)\|\|x_1 - x_0\| dt$$

$$\overset{(d)}{=} \int_0^1 \|x_t - x_0\|\|x_1 - x_0\| dt$$

$$\overset{(e)}{=} \|x_1 - x_0\|^2 \int_0^1 t\, dt = \frac{1}{2}\|x_1 - x_0\|^2.$$

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 $x_t - x_0 = t(x_1 - x_0)$.

**27.3.1 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

$$x^+ = \arg \min_y f(x) + \langle f'(x), y - x \rangle + \frac{L}{2} \|y - x\|^2.$$  \hspace{1cm} (27.2)

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 27.4.** Let $f$ be $L$-smooth and $x \in \mathbb{R}^n$. Let $x^+$ be defined by (27.2). Then

$$x^+ = x - \frac{f'(x)}{L}.$$  

Moreover,

$$f(x^+) \leq f(x) - \frac{1}{2L} \|f'(x)\|^2.$$  

**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

$$f(x^+) \leq f(x) + \langle f'(x), x^+ - x \rangle + \frac{L}{2} \|x^+ - x\|^2 \quad \overset{(a)}{=} f(x) - \frac{1}{L} \|f'(x)\|^2 + \frac{1}{2L} \|f'(x)\|^2 \quad \overset{(b)}{=} f(x) - \frac{1}{2L} \|f'(x)\|^2,$$

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 $f'(x) = 0$. 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 27.5.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be an $L$-smooth function. Let $\epsilon > 0$ and $x_0$ be given. Let $x^*$ 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 (27.2) above). Then for all $T \in \mathbb{N}$,

$$\min_{t=1, ..., T} \|f'(x_t)\|^2 \leq \frac{2L(f(x_0) - f(x^*))}{T}.$$  

391
Proof. We have
\[ f(x_0) - f(x^*) \geq f(x_0) - f(x_T) = \sum_{t=1}^{T} f(x_{t-1}) - f(x_t) \]
\[ \geq \sum_{t=1}^{T} \frac{\|f'(x_t)\|^2}{2L} \geq \frac{T}{2L} \min_{t=1,...,T} \|f'(x_t)\|^2, \]
as desired. Here (a) is by telescoping series. (b) is by lemma 27.4.

27.3.2 Gradient step 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 (27.2) above. In addition to being smooth, we assume \( f \) is convex. Let \( x^* \) 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, \( f(x) - f(x^*) \). The second is in the Euclidean distance in the input space, \( \|x - x^*\| \). For both we will show that \( x^+ \) improves \( x \).

The first lemma shows that \( x^+ \) is closer to \( x^* \) than \( x \).

Lemma 27.6. For all \( x \),
\[ \|x^+ - x^*\|^2 \leq \|x - x^*\|^2. \]

Proof. First we have
\[ \frac{1}{2L} \|f'(x)\|^2 \leq f(x) - f(x^+) \leq f(x) - f(x^*) \leq \langle f'(x), x - x^* \rangle \]

(27.3)

Here (a) is by lemma 27.4. (b) is because \( x^* \) is the global minimum. (c) is by convexity of \( f \). Now, we have
\[ \|x^+ - x^*\|^2 = \|x - f'(x)/L - x^*\|^2 \]
\[ = \|x - x^*\|^2 - \frac{2}{L} \langle f'(x), x - x^* \rangle + \frac{1}{L^2} \|f'(x)\|^2 \]
\[ \leq \|x - x^*\|^2. \]

Here (d) is by the inequality obtained in (27.3).
Lemma 27.7. For all $x$,
\[ f(x) - f(x^*) \geq \frac{1}{2L} \left( \frac{f(x) - f(x^*)}{\|x - x^*\|} \right)^2. \]

Proof. We have
\[ f(x) - f(x^*) \quad \overset{(a)}{\leq} \quad \langle f'(x), x - x^* \rangle \quad \overset{(b)}{\leq} \quad \|f'(x)\| \|x - x^*\| \quad \overset{(c)}{\leq} \quad \|x - x^*\| \sqrt{2L(f(x) - f(x^*)}). \]

Here (a) is by convexity. (b) is by Cauchy-Schwartz. (c) is by lemma 27.4. Squaring both sides (where we note that both sides are nonnegative) and rearranging gives the desired inequality.

27.3.3 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 $x^*$ 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.,
\[ x_{t+1} = x_t^* = \arg\min_y f(x_t) + \langle f'(x_t), y - x_t \rangle + \frac{L}{2} \|y - x_t\|^2 \]
\[ = x_t - \frac{1}{L} f'(x_t). \]

Then $f(x_t)$ converges to $f(x_0)$ at the following rate.

Theorem 27.8. For $T = O\left(\frac{L\|x_0 - x^*\|^2}{\epsilon}\right)$, we have
\[ f(x_T) - f(x^*) \leq \epsilon. \]

Proof. We have
\[ f(x_t) - f(x_{t+1}) \quad \overset{(a)}{\geq} \quad \frac{(f(x_t) - f(x^*))^2}{2L\|x_t - x^*\|^2} \quad \overset{(b)}{\geq} \quad \frac{(f(x_t) - f(x^*))^2}{2L\|x_0 - x^*\|^2}. \]

Here (a) is by lemma 27.7 and (b) is by lemma 27.6.

Now, fix $\delta > 0$. Suppose the error is bounded above by $f(x_t) - f(x^*) \leq 2\delta$, and consider the number of iterations until $f(x_t) - f(x^*) \leq \delta$. As long as $f(x_t) - f(x^*) \geq \delta$, the decrease in error is at least
\[ f(x_t) - f(x_{t+1}) \geq \frac{\delta^2}{2L\|x_0 - x^*\|^2}. \]
Thus there can be at most $2L\|x_0 - x^*\|^2 / \delta$ iterations before has decreased from at most $2\delta$ to below $\delta$.

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 $O\left( L\|x_0 - x^*\|^2 / \epsilon_i \right)$ iterations where the error, $f(x_t) - f(x^*)$, is between $\epsilon_i$ and $\epsilon_{i-1}$. Thus the total number of iterations to reach error $\epsilon = \epsilon_0$ is at most

$$\sum_{i=0}^{\infty} O\left( \frac{L\|x_0 - x^*\|^2}{\epsilon_i} \right) = O\left( \frac{L\|x_0 - x^*\|^2}{\epsilon} \right) \sum_{i=0}^{\infty} \frac{1}{2^i} \leq O\left( \frac{L\|x_0 - x^*\|^2}{\epsilon} \right),$$

as desired.

\[ \blacksquare \]

### 27.4 Exercises

**Exercise 27.1.** Prove that for all $x, y \in \mathbb{R}^n$,

$$\|x + y\|^2 = \|x\|^2 + \|y\|^2 + 2\langle x, y \rangle.$$

**Exercise 27.2.** Prove that, for all $a, b \in \mathbb{R}$,

$$2ab \leq a^2 + b^2.$$

**Exercise 27.3.** Prove the AM-GM inequality (for two variables): For two variables $a, b > 0$,

$$\sqrt{ab} \leq \frac{a + b}{2}.$$

**Exercise 27.4.** Prove the Cauchy-Schwartz inequality.

**Exercise 27.5.** Prove the triangle inequality.

**Exercise 27.6.** Prove the following inequality.

$$\|x\| - \|y\| \leq \|x - y\|$$

for all $x, y$.

**Exercise 27.7.** Let $f : \mathbb{R} \rightarrow \mathbb{R}$ 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 $x^*$ 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 $x^* - x \leq \epsilon$. (The running time should depend on $\epsilon$ (or rather $1/\epsilon$). As usual, the better the dependency on $\epsilon$, the better.)
**Exercise 27.8.** 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 $1/\text{poly}(n)$ and $\text{poly}(n)$. Design and analyze a variation of the gradient descent algorithm in section 27.3 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 27.9.** Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ be an invertible map such that $\|f(x)\| \geq \mu \|x\|$ for all $x$. Prove that

$$\|f^{-1}(y)\| \leq \frac{1}{\mu} \|y\|$$

for all $y$. 


Chapter 28

Greedy Approximations and LP Duality

28.1 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 \( f : 2^{[n]} \rightarrow \mathbb{R}_{\geq 0} \) by

\[
f(S) \overset{\text{def}}{=} (\# \text{ points covered by sets (w/ index) in } S) = \left| \bigcup_{e \in S} A_e \right|.
\]

Now, for \( i = 1, \ldots, k \), let \( e_i \in [n] \) be the index of the \( i \)th 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 28.1 below as saying that the greedy algorithm maintains the inequality

\[(\text{improvement}) \geq \frac{1}{k} (\text{room for improvement}),\]
on each iteration.

**Lemma 28.1.** For \(i = 1, \ldots, k\), \(f(S_i) - f(S_{i-1}) \geq \frac{1}{k} (\text{OPT} - f(S_{i-1})).\)

**Proof of lemma 28.1.** The proof is illustrated by the following diagram.

There are at least \(\text{OPT} - f(S_{i-1})\) 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 \((1/k)(\text{OPT} - f(S_{i-1}))\) to the total coverage.

**Overall analysis.**

**Theorem 28.2.** \(f(S_k) \geq (1 - \epsilon^{-3}) \text{OPT}.\)

**Proof.** We have shown that

\[(i\text{th improvement}) = f(S_i) - f(S_{i-1}) \geq \frac{1}{k} (\text{OPT} - f(S_{i-1})) = (\text{room for improvement after } i - 1 \text{ iterations})\]

Rearranging, we get

\[\text{OPT} - f(S_i) \leq \left(1 - \frac{1}{k}\right)(\text{OPT} - f(S_{i-1})).\]

That is,

\[\text{(room for improvement after } i \text{ iterations}) \leq \left(1 - \frac{1}{k}\right)(\text{room for improvement after } i - 1 \text{ iterations}).\]
We see that “room for improvement” decays exponentially. Unrolling the above, we have

\[
\begin{align*}
\text{OPT} - f(S_0) &= \text{OPT} \\
\text{OPT} - f(S_1) &\leq \left(1 - \frac{1}{k}\right) \text{OPT} \\
\text{OPT} - f(S_2) &\leq \left(1 - \frac{1}{k}\right)^2 \text{OPT} \\
\text{OPT} - f(S_3) &\leq \left(1 - \frac{1}{k}\right)^3 \text{OPT} \\
&\vdots \\
\text{OPT} - f(S_k) &\leq \left(1 - \frac{1}{k}\right)^k \text{OPT} \\
&\leq e^{-1} \text{OPT}.
\end{align*}
\]

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. ■

28.2 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

\[(Ax)_i = \sum_{j=1}^{n} A_{i,j} x_j\]

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 \(A \in \mathbb{R}^{n \times n}\) is the matrix \(A^T \in \mathbb{R}^{n \times m}\) uniquely defined by\(^1\)

\[
\langle y, Ax \rangle = \langle A^T y, x \rangle
\]

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

\[A^T_{i,j} = A_{j,i}\]

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

\[
\min/\max \langle b, x \rangle = \sum_{j=1}^{n} b_j x_j \text{ over } x \in \mathbb{R}^n \\
s.t. \quad A_1 x \leq c_1, \ A_2 x = c_2, \text{ and } A_3 x \geq c_3.
\]

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

\[
\max \langle b, x \rangle \text{ over } x \in \mathbb{R}^n_{\geq 0} \text{ s.t. } Ax \leq c. \quad (P)
\]

where \( A \in \mathbb{R}^{m \times n}_{\geq 0}, b \in \mathbb{R}^n_{> 0}, \text{ and } c \in \mathbb{R}^m_{> 0} \) all have nonnegative coefficients. We let \( \text{Opt}(P) \) denote the optimum value of the LP \( \text{Opt}(P) \).

Covering LPs. A covering LP is a linear program of the form

\[
\min \langle c, y \rangle \text{ over } y \in \mathbb{R}^m_{\geq 0} \text{ s.t. } A^T y \geq b, \quad (C)
\]

where\(^2\) \( A \in \mathbb{R}^{m \times n}_{\geq 0}, b \in \mathbb{R}^n_{> 0}, \text{ and } c \in \mathbb{R}^m_{> 0} \). Abusing notation, we also let \( \text{Opt}(C) \) denote the optimum value of the LP \( \text{Opt}(C) \).

LP duality. LP duality is about the relationship between the linear programs \( \text{Opt}(P) \) and \( \text{Opt}(C) \), 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

\[
\langle b, x \rangle \overset{(a)}{\leq} \langle A^T y, x \rangle \overset{(b)}{=} \langle y, Ax \rangle \overset{(c)}{\leq} \langle y, c \rangle.
\]

Here (a) is because \( x \geq 0 \) and \( A^T y \geq b \). (b) is by definition of transpose. (c) is because \( y \geq 0 \) and \( Ax \leq c \). Thus, for a packing problem \( (P) \) and a covering problem \( (C) \) linked by duality, we have

\[
\text{Opt}(P) \leq \text{Opt}(C).
\]

\(^2\)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 \( (C) \).
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 $\text{OPT}(P) = \text{OPT}(C)$? The LP duality theorem (for packing and covering problems) states that in fact they are always equal.

**Theorem 28.3** (LP Duality for packing and covering). $\text{OPT}(P) = \text{OPT}(C)$.

Our primary goal for the rest of the discussion is to prove theorem 28.3. Along the way, we will also develop polynomial time approximation algorithms for $(P)$.

### 28.3 An algorithmic proof of LP duality

#### 28.3.1 Preliminaries

**The partition function.** We will use a convex potential function $\pi : \mathbb{R}^m \rightarrow \mathbb{R}$, 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 28.4.** 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. $\blacksquare$

The second property, below, is about the derivative of $\pi(x)$ and how it is particularly easy to interpret.

**Lemma 28.5.** 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 $[m]$. 

400
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 28.6.** 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

$$
\pi'_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{m} e^{x_j}}.
$$

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,

$$
\text{maximize } \langle 1, x \rangle \text{ over } x \in \mathbb{R}^n_{\geq 0} \text{ s.t. } Ax \leq \mathbb{1} \quad (P_1)
$$

and the normalized covering problem,

$$
\text{minimize } \langle 1, y \rangle \text{ over } y \in \mathbb{R}^m_{\geq 0} \text{ s.t. } A^T y \geq 1 \quad (C_1)
$$

Note that the rescaling did not affect the optimum value of either LP: $\text{Opt}(P) = \text{Opt}(P_1)$, and $\text{Opt}(C) = \text{Opt}((C_1))$. For ease of notation we will use $\text{Opt}(P)$ to refer to the optimum value of $(P_1)$ and $\text{Opt}(C)$ to refer to the optimum value of $(C_1)$.
28.3.2 Lagrangian relaxations

We let $\Delta^m$ denote the set of probability vectors over $[m]$; i.e.,

$$\Delta^m = \{ p \in \mathbb{R}^m_{\geq 0} : p_1 + \cdots + p_m = 1 \}.$$

For a probability vector $p \in \Delta^m$, consider the packing problem obtained by using $p$ to collapse the packing constraints $Ax \leq 1$ into a single packing constraint $\langle p, Ax \rangle \leq 1$.

$$\max \langle 1, x \rangle \text{ over } x \in \mathbb{R}^n_{\geq 0} \text{ s.t. } \langle p, Ax \rangle \leq 1. \quad (L)$$

For fixed $p$, we let $\text{Opt}(L)$ denote the objective value to (L).

Observe that $\langle p, Ax \rangle$ is a convex combination of packing constraints ($\langle Ax \rangle_i \leq 1$ for all $i$) of $(P_1)$. As such, (L) is a relaxation of $(P_1)$ – every feasible solution to $(P_1)$ is a feasible solution to (L). It is sometimes called a Lagrangian relaxation of $(P_1)$. As a relaxation of $(P_1)$, we automatically have

$$\text{Opt}(L) \geq \text{Opt}(P).$$

Interestingly, we can also relate $\text{Opt}(L)$ to $\text{Opt}(C)$, as follows.

**Lemma 28.7.** For all $p \in \Delta^m$, $\text{Opt}(L) \geq \text{Opt}(C)$.

**Proof.** Let $\lambda > \text{Opt}(L)$. We claim that $\lambda p$ is a feasible solution to $(C_1)$. This claim implies that $\lambda \geq \text{Opt}(C)$ for all $\lambda > \text{Opt}(L)$. Taking the limit as $\lambda \downarrow \text{Opt}(L)$, we obtain $\text{Opt}(L) \geq \text{Opt}(C)$, desired.

Now, fix $i \in [m]$. Let $x = \lambda e_i$, where $e_i$ is the vector that is 0 everywhere except the $i$th coordinate is 1. Since $\langle 1, x \rangle = \lambda > \text{Opt}(L)$, we must have $\langle p, Ax \rangle > 1$. Expanding out $\langle A, p \rangle x$, we have

$$\langle p, Ax \rangle = \lambda (A^T p)_i = \left( A^T (\lambda p) \right)_i > 1.$$

Thus $(A^T (\lambda p))_i > 1$ for all $i$. That is, $\lambda p$ is a feasible solution to $(C_1)$, hence $\langle 1, \lambda p \rangle = \lambda > \text{Opt}(C)$. ■

28.3.3 Finally, the algorithm

We design an algorithm for $(P_1)$ based on the following potential function that uses $\pi$. Let $\epsilon > 0$ be given, and let $\eta = \log(m)/\epsilon$. Consider the function $f : \mathbb{R}^n_{\geq 0} \rightarrow \mathbb{R}$ defined by

$$f(x) \overset{\text{def}}{=} \frac{1}{\eta} \pi(\eta Ax). \quad (28.1)$$

We think of $f$ is a continuous approximation to $\max_i (Ax)_i$, as follows.
Lemma 28.8. For all $x$,
\[
\max_i (Ax)_i \leq f(x) \leq \max_i (Ax)_i + \epsilon.
\]

Proof. By lemma 28.4, we have
\[
\max_i \eta (Ax)_i \leq \eta f(x) \leq \max_i \eta (Ax)_i + \log(m).
\]
Dividing by $\eta$ 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 28.9. Let $\epsilon > 0$ and consider an LP of the form $(P_1)$. Suppose that, given any distribution $p \in \Delta_m$, one can compute a $(1 - \epsilon)$-approximation to the Lagrangian relaxation
\[
\text{maximize } \langle \mathbb{1}, x \rangle \text{ s.t. } \langle p, Ax \rangle \leq 1.
\]
Then in $O(m \log(n)/\epsilon^2)$ iterations, one can compute a point $x \in \mathbb{R}_{\geq 0}^n$ such that
\[
\langle \mathbb{1}, x \rangle \geq (1 - \epsilon)\text{Opt}(C) \text{ and } Ax \leq 1.
\]
Therefore, $\text{Opt}(P) \geq (1 - \epsilon)\text{Opt}(C)$ for all $\epsilon > 0$ sufficiently small, hence $\text{Opt}(P) = \text{Opt}(C)$.

Proof. For simplicity, we will directly show how to obtain a $(1 - ce\epsilon)$-approximation for some constant $c > 0$. A $(1 - \epsilon)$-approximation is obtained by decreasing $\epsilon$ appropriately.

We will analyze a discrete algorithm that iteratively solves different Lagrangian relaxations of $(P_1)$. 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 $k$th 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 $k$th iteration. We start from an empty solution, $x(t_1) = x(0) = 0$. 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 $t$. 

\[403\]
Fix an iteration \( k \). Consider the optimization problem,

\[
\text{maximize } \langle 1, z \rangle \text{ s.t. } f'(x(t_k)), z \leq 1.
\]

Expanding out \( f'(x(t_k)) = A^T\pi'(\eta Ax(t_k)) \) and transposing, the above problem is the same as the following Lagrangian relaxation of \((P_\epsilon)\):

\[
\text{maximize } \langle 1, z \rangle \text{ s.t. } \langle \pi'(\eta Ax(t_k)), Ax(t_k) \rangle \leq 1.
\]

Here we recall that \( \pi'(\eta Ax(t_k)) \in \Delta^m \), by lemma 28.5. Now, by lemma 28.7, this relaxation has optimum value at least \( \text{Opt}(C) \).

We approximately solve the Lagrangian relaxation and obtain a point \( z_k \) such that

\[
\langle 1, z_k \rangle \geq (1 - \epsilon) \text{Opt}(C) \text{ and } \langle \pi'(\eta Ax(t_k)), Az_k \rangle \leq 1.
\]

Let \( \delta_k \) large as possible such that \( t_k + \delta_k \leq 1 \) and

\[
\delta_k \eta Az_k \leq \epsilon. \tag{a}
\]

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 \( t \in [0,1] \),

\[
\frac{d}{dt} \langle 1, x(t) \rangle \geq (1 - \epsilon) \text{Opt}(C).
\]

At any time \( t \), we have \( x'(t) = z_k \) for some \( k \), hence \( \frac{d}{dt} \langle 1, x(t) \rangle = \langle 1, x'(t) \rangle \geq (1 - \epsilon) \text{Opt}(C) \).

**Claim 2.** For all \( t \in [0,1] \), we have

\[
\frac{d}{dt} f(x(t)) \leq \epsilon.
\]

Fix \( t \). Then \( t \in [t_k, t_{k+1}] \) for some iteration \( k \). We have

\[
\eta Ax(t_k) \leq \eta Ax(t) \leq \eta Ax(t_{k+1}) = \eta Ax(t_k) + \delta_k \eta Az_k \leq \eta Ax(t_k) + \epsilon \tag{a}
\]

by \( \text{(a)} \) choice of \( \delta_k \). By lemma 28.6, the inequalities above imply that \( f'(x(t)) \leq \epsilon f'(x(t_k)) \). Thus

\[
\frac{d}{dt} f(x(t)) = \langle f'(x(t)), z_k \rangle \leq \epsilon \langle f'(x(t_k)), z_k \rangle \leq \epsilon,
\]

as desired.

We have now analyze the rate of change of both the objective, \( \langle 1, x(t) \rangle \), 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. $\langle \mathbb{1}, x(1) \rangle \geq \text{Opt}(C)$.

We have

$$\langle \mathbb{1}, x(1) \rangle = \langle \mathbb{1}, x(1) - x(0) \rangle \overset{(b)}{=} \int_0^1 \frac{d}{dt} \langle \mathbb{1}, x(t) \rangle \, dt \overset{(c)}{=} \int_0^1 (1 - \epsilon) \text{Opt}(C) \, dt = (1 - \epsilon) \text{Opt}(C).$$

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

$$\max_i A(x(1)) - \epsilon \overset{(d)}{=} f(x(1)) - f(x(0)) = \int_0^1 \frac{d}{dt} f(x(t)) \, dt$$

$$= \int_0^1 \langle f'(x(t)), x(t) \rangle \, dt \leq \int_0^1 e^\epsilon \, dt = e^\epsilon.$$

(d) is by lemma 28.8. (e) is the fundamental theorem of calculus. (f) is the chain 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 $\langle \mathbb{1}, x \rangle \geq (1 - \epsilon) \text{Opt}(C)$ 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 $(P_{\mathbb{1}})$, with objective value

$$\left\langle \mathbb{1}, \left( \frac{1}{1 + 3\epsilon} \right) x(\mathbb{1}) \right\rangle \geq \frac{1 - \epsilon}{1 + 3\epsilon} \text{Opt}(C).$$

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 $(P_{\mathbb{1}})$. 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 $\delta_k$ 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 $\epsilon/\eta$ 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 29

Randomized rounding

29.1 Max-flow Min-cut via LP duality

Recall the \((s, t)\)-max flow problem, which we present in terms of its path packing. The input consists of a graph \(G = (V, E)\) with positive edge capacities \(c : E \to \mathbb{R}_{\geq 0}\), a source vertex \(s\), and a sink vertex \(t\). The \((s, t)\)-max flow problem asks for a maximum size fractional packing of \((s, t)\) paths into \(G\). To formalize this problem as a linear program, let \(\mathcal{P}_{s,t}\) denote the collection of all \(s \to t\) paths. Then the maximum flow is given by the following packing LP.

\[
\begin{align*}
\text{maximize} & \quad \sum_{p \in \mathcal{P}_{s,t}} x_p \\
\text{over} & \quad x : \mathcal{P}_{s,t} \to \mathbb{R}_{\geq 0} \\
\text{subject to} & \quad \sum_{p \ni e} x_p \leq c(e) \quad \text{for all } e \in E.
\end{align*}
\]

(Max-Flow)

Note that \(\mathcal{P}_{s,t}\) has exponentially many variables, so we could not write out the LP in full in polynomial time.

Recall also the \((s, t)\)-minimum cut problem. Given the same input as \((s, t)\)-max flow, we interpret the capacities \(c : E \to \mathbb{R}_{\geq 0}\) as costs. The \((s, t)\)-minimum cut is the minimum cost set of edges whose removal disconnects \(s\) from \(t\). We can think of this as a (discrete) covering problem, where we want to choose “edges covering paths”, as follows.

\[
\begin{align*}
\text{minimize} & \quad \sum_{e \in C} c(e) \\
\text{over} & \quad C \subseteq E \text{ s.t. } p \cap C \neq \emptyset \text{ for all } p \in \mathcal{P}_{s,t}.
\end{align*}
\]

(Min-Cut)

We might also consider the fractional relaxation of the \((s, t)\)-minimum cut, where we want the minimum cost

\[
\begin{align*}
\text{minimize} & \quad \sum_{e \in E} y_e \\
\text{over} & \quad y : E \to \mathbb{R}_{\geq 0} \\
\text{subject to} & \quad \sum_{e \in p} y_e \geq 1 \quad \text{for all } p \in \mathcal{P}_{s,t}.
\end{align*}
\]

(Frac.-Min-Cut)
Abusing notation, we let \((\text{Min-Cut})\) denote the value of the minimum discrete \((s,t)\)-cut and we let \((\text{Frac.-Min-Cut})\) denote the value of the linear program given in \((\text{Frac.-Min-Cut})\). \textit{A priori}, we know that \((\text{Frac.-Min-Cut}) \leq (\text{Min-Cut})\) because the former is a relaxation of the latter.

**LP duality.** Interestingly, we can relate \((\text{Frac.-Min-Cut})\) and \((\text{Max-Flow})\) via LP duality. Recall the LP-duality theorem, which we restate below.

**Theorem 29.1** (LP duality theorem). Let \(A \in \mathbb{R}^{m \times n}, \ b \in \mathbb{R}^n, \ \text{and} \ c \in \mathbb{R}^{m \times n}\). Consider the packing LP

\[
\max \langle b, x \rangle \ \text{over} \ x \in \mathbb{R}^n_{\geq 0} \ \text{s.t.} \ Ax \leq c \quad (P)
\]

and the dual covering LP

\[
\min \langle c, y \rangle \ \text{over} \ y \in \mathbb{R}^m_{\geq 0} \ \text{s.t.} \ A^T y \geq b. \quad (C)
\]

Then \(\text{OPT}(P) = \text{OPT}(C)\).

Now, the maximum flow problem \((\text{Max-Flow})\) is a packing LP \((P)\). Here the constraint matrix \(A \in \{0,1\}^{E \times E_{s,t}}\) is the incidence matrix between edges and paths. For each edge \(e \in E\) and path \(p \in P_{s,t}\), we have \(A(e,p) = 1\) if \(e \in p\), and 0 otherwise. We also have \(b = 1\), and \(c\) (in \((P)\)) equal to the edge capacities in the graph (again \(c\)).

Likewise, the fractional minimum cut problem \((\text{Frac.-Min-Cut})\) is a covering LP of the form \((C)\). The covering constraint \(A^T \in \{0,1\}^{P_{s,t} \times E}\) is the transpose of the edge-path incidence matrix mentioned. The costs \(c\) (in \((C)\)) correspond to the capacities in \(G\) and the.

All put together, we have

\((\text{Max-Flow}) \overset{(a)}{=} (\text{Frac.-Min-Cut}) \overset{(b)}{\leq} (\text{Min-Cut}).\)

To recap, (a) is by LP duality. (b) is because \((\text{Frac.-Min-Cut})\) is a relaxation of \((\text{Min-Cut})\).

**Theorem 29.2** (Ford and Fulkerson [FF56] and Menger [Men27]). \((\text{Max-Flow}) = (\text{Min-Cut}).\)

We already proved this combinatorially, via the Ford-Fulkerson algorithm. But here we given an alternative proof based on LP duality\(^1\). We will directly prove that \((\text{Min-Cut}) = (\text{Frac.-Min-Cut}).\)

\(^1\)The proof is colorfully illustrated in a video by the author available at https://youtu.be/J4yUdABv1TE
Let \( y \in \mathbb{R}^E_\geq 0 \) be an optimum solution to the minimum cut LP, (Frac.-Min-Cut). We claim that, given \( y \), we can find an \((s, t)\)-cut \( C \subseteq E \) with total capacity, \( \sum_{e \in C} c(e) \) equal to the fractional capacity of \( y \), \( \langle e, y \rangle \).

**Line embeddings and sweep cuts.** We embed the vertices \( V \) on the line by assigning values \( \alpha : V \to [0, +\infty) \) as follows.

For each vertex \( v \), let \( \alpha_v \) be the length of the shortest \( s \leadsto v \) path w.r.t the edge lengths \( y \in \mathbb{R}^E_{\geq 0} \). We have \( \alpha_s = 0 \). We also have

\[
\alpha_t = \min_{p \in \mathcal{P}_{s,t}} \sum_{e \in p} y_e \geq 1
\]

because of the covering constraints in (Frac.-Min-Cut). Consider the following random cut. We pick a value \( \theta \in (0, 1) \) uniformly at random. Let \( S = \{ v : x_v \leq \theta \} \), and let \( \tilde{S} = V \setminus S \). Since \( \alpha_s = 0 \) and \( \alpha_t \geq 1 \), \((S, \tilde{S})\) is always an \((s, t)\)-cut.

Let us bounded the cost of the directed cut from \( S \) to \( \tilde{S} \), in expectation. We have

\[
E \left[ \sum_{e \in \partial^+(S)} c(e) \right] \stackrel{(d)}{=} \sum_{e \in E} c(e) \mathbf{P} \left[ e \in \partial^+(S) \right] \stackrel{(d)}{=} \sum_{e \in E} c(e) y_e = \text{(fractional min cut)}.
\]

Here (c) is by linearity of expectation. (d) is by the following argument.

\[
\text{(29.1)}
\]
29. Randomized rounding  

29.2 Set cover

For an edge $e = \{u, v\}$, $e \in \partial^+(S)$ iff $\alpha_u \leq \theta \leq \alpha_v$, which happens with probability $\leq \alpha_v - \alpha_u$. But concatenating the shortest path from $s$ to $u$ with the edge $e$,

$$s \xrightarrow{\alpha_u} u \xrightarrow{y_e} v,$$

gives a walk of length $\alpha_u + y_e$, so $\alpha_u \leq \alpha_u + y_e$.

Consider now the inequality obtained in (29.1),

$$E \left[ \sum_{e \in \partial^+(S)} c(e) \right] \leq \text{(fractional min cut)}.$$

We have generated a randomized (discrete) cut that is on average no worse than the minimum fractional minimum cut. By the probabilistic method, there exists a value $\theta$ where the $(s, t)$-cut has value at most this average. If not, then the average would have to be higher. This establishes the existence of a minimum cut with value equal to the minimum fractional minimum cut, hence the max flow min cut theorem. To extract the cut, one can simply scan $\theta$ over the interval $(0, 1)$ and check all $n - 1$ possible cuts. (In fact any $\theta \in (0, 1)$ will work, see exercise 29.1.)

A second part of the maximum flow minimum cut theorem asserts that when the capacities are integral, so is the maximum flow. The line embedding proof above does not capture this. However, knowledge of the fact that there is always a discrete cut with the same size as the maximum flow is enough to prove integrality (see exercise 29.2).

29.2 Set cover

In the set cover problem, we have $m$ elements $[m] = \{1, \ldots, m\}$, and $n$ sets $A_1, \ldots, A_n \subseteq [m]$. A set cover is a collection of sets that covers all of the elements:

$$|A_{e_1} \cup A_{e_2} \cup \cdots \cup A_{e_k}| = [m].$$

The minimum set cover problem is to find the set cover of minimum cardinality. We let $\text{Opt}$ denote the size of the minimum set cover. Unsurprisingly, set cover is NP-Hard (see exercise 29.3).

Previously we studied the closely related maximum coverage problem, where one wants to cover as many elements as possible subject to a cardinality constraint. There we showed that the greedy algorithm is a $(1 - 1/e)$-approximation algorithm. For set cover, we take a pursue a different approach based on the LP. Consider the following LP relaxation of the set cover problem.

$$\text{minimize } \langle 1, y \rangle \text{ over } y : [n] \to \mathbb{R}_{\geq 0} \text{ s.t. } \sum_{j : A_j \ni i} y_j \geq 1 \text{ for all } i \in [m]. \quad (29.2)$$
Here we have one variable $y_j$ for each variable $A_j$, which reflects how much of $A_j$ we take. For each $i$, the constraint $\sum_{j : A_j \ni i} y_j \geq 1$ says that we have to take at least one (fractional) set covering the $i$th point.

We can solve the LP (29.2) in polynomial time, producing a fractional set cover $y$. Note that since (29.2) is a relaxation of the discrete set cover problem, $\langle 1, y \rangle$ is at most $\Opt$ (the optimal value of the set cover).

The only deficiency of $y$, of course, is that it is only a fractional set cover. To say we cover a point with “half of one set and half of another set” is nonsense, in the sense that “half a set” is nonsense. A different interpretation of those fractional values is to say that we cover a point “half the time with this set, and half the time with that set”. This makes more sense, and inspires us to use $y$ to generate a random distribution of collection of sets $S \subseteq [n]$, where the probability of a set $A_j$ appearing in $S$ reflects $y_j$. However it is generally impossible to generate a random set $S$ where the marginal probabilities match $y$ exactly. So instead we oversample.

Let $\alpha = c \ln(m)$ for a suitably large constant $c > 0$. For each set $A_j$, independently with probability $p_j = \min\{1, \alpha y_j\}$, we add the set $A_j$ to $S$. This produces a randomized collection of sets $S$ that we will proceed to analyze. To recap, the overall algorithm is as follows.

1. solve the LP (29.2) to obtain an optimum solution $y$.
2. $S \leftarrow \emptyset$ and $\alpha \leftarrow c \log(m)$ for a sufficiently large constant $c > 0$.
3. for $j = 1, \ldots, n$, independently with probability $p_j = \min\{1, \alpha y_j\}$ add $j$ to $S$.
4. return $S$.

The expected number of sets we take is

$$E[|S|] \overset{(a)}{=} \sum_j p_j \leq \alpha \sum_j y_j \overset{(b)}{\leq} \alpha \Opt.$$  

Here (a) is by linearity of expectation. (b) is because $\sum_j y_j \leq \Opt$. It remains to show that $S$ is a set cover (most of the time). Fix a point $i$.

$$\mathbb{P}[i \text{ not covered by } S] \overset{(c)}{=} \prod_{j : A_j \ni i} \mathbb{P}[A_j \text{ not sampled}] = \prod_{j : A_j \ni i} \max\{0, 1 - \alpha y_j\} \overset{(d)}{\leq} \prod_{j : A_j \ni i} e^{-\alpha y_j} = e^{-\alpha \sum_{j : A_j \ni i} y_j} \overset{(e)}{\leq} e^{-\alpha} = \frac{1}{2^m},$$


where we recall that $c$ is a constant under our control. (c) invokes the fact that the sets are sampled independently. (d) is by the inequality $1 + x \leq e^x$ for all $x$. (e) is because $y$ is a feasible solution to (29.2). By the union bound,

$$
P[S \text{ not a set cover}] \leq \sum_{i=1}^{m} P[i \text{ not covered by } S] \leq m \cdot \frac{1}{mc}
$$

Thus for $c > 2$ (say), $S$ is a set cover with probability at least $1 - 1/m$.

**Theorem 29.3.** In randomized polynomial time, one can produce a randomized collection of sets with the following guarantees.

1. The expected number of sets is $O(\ln m)\text{Opt}$.

2. The collection of sets is a set cover with probability at least $1 - 1/m$.

### 29.3 Packing Multicommodity Paths

Recall the integral multicommodity flow problem, which extends the $(s, t)$-flow problem by allowing for multiple source-sink pairs.

Let $G = (V, E)$ be a directed graph with positive edge capacities $c : E \to \mathbb{N}$. Let $(s_1, t_1), \ldots, (s_k, t_k)$ be $k$ multicommodity pairs. We want to find $k$ paths $(p_1 : s_1 \to t_1, \ldots, p_k : s_k \to t_k)$, one for each $(s_i, t_i)$, that minimizes the congestion in the graph. Given a set of paths $p_1, \ldots, p_k$ and an edge $e$, the *congestion* on $e$ is defined as the ratio of the total flow on $e$ divided by the capacity of $e$:

$$
\text{(congestion of } p_1, \ldots, p_k \text{ on } e) = \sum_{i : e \in p_i} 1/c(e).
$$

The overall congestion of $p_1, \ldots, p_k$ is the maximum congestion over all edges $e \in E$.

We have previously discussed the integral multicommodity flow problem where we observed it is NP-Hard even for $k = 2$. But the fractional version of the problem is not NP-Hard, because it is an LP. Let $\mathcal{P}$ denote the collection of all paths in the graph. Consider the following linear program.

\[
\begin{align*}
\text{minimize } & \lambda \\
\text{over } & \lambda > 0, \ x : \mathcal{P} \to \mathbb{R}_{\geq 0} \\
\text{s.t. } & \sum_{p : e \in p} x_p \leq \lambda c(e) \text{ for all } e \in E \\
& \sum_{p : x_i \sim t_i} x_p \geq 1 \text{ for } i = 1, \ldots, k.
\end{align*}
\]
This LP has a variable $\lambda > 0$ that represents the congestion.

Although this LP is exponentially large, it can still be solved, producing a solution $\lambda > 0$ and $x : \mathcal{P} \to \mathbb{R}_{\geq 0}$ of polynomial size. We discuss this aspect below in section 29.3.1. For the moment, we instead focus on the issue of rounding a fractional solution $\lambda > 0$ and $x : \mathcal{P} \to \mathbb{R}_{\geq 0}$ to an integral selection of paths with congestion comparable to $\lambda$.

The algorithm is very simple. For each commodity $(s_i, t_i)$, randomly sample a single $(s_i, t_i)$-path $p$ in proportion to $x_p$. We return the sampled path.

We want to argue that, for each edge $e$, the congestion on $e$ w.r.t the sampled paths. Observe that for each path $p$, $p$ is randomly sampled with probability at most $x_p$. Thus for each edge $e$, the expected congestion is $\sum_{p: e} x_p \leq \lambda c(e)$.

From the applied viewpoint of routing, it also natural to assume that we are only interested in congestion when it is at least one. Increasing $\lambda$ if necessary, we assume that $\lambda \geq 1$. This also be convenient for technical reasons that we point out below.

**Lemma 29.4** (Multiplicative Chernoff inequality). Let $X_1, \ldots, X_n \in [0,1]$ be independent random variables, and let $\epsilon > 0$. Let $\mu \geq E[X_1 + \cdots + X_n]$ be an upper bound the expected sum. Then

$$
P\left[ \sum_{i=1}^{n} X_i \geq (1 + \epsilon)\mu \right] \leq e^{-\epsilon^2\mu/(1+\epsilon)}.
$$

We will not prove the multiplicative Chernoff bound due to time constraints. Perhaps the reader has seen it elsewhere as it is widely used in computer science. Proofs can be found in many places such as [Qua20a; Wikb].

Fix an edge $e \in E$. For each commodity $(s_i, t_i)$, $X_i \in \{0, 1\}$ be the random indicator variable that indicates whether the random path chosen for $(s_i, t_i)$ contains $e$. Note that the $X_i$’s are independent. We also have

$$
E\left[ \sum_{i=1}^{k} X_i \right] = \sum_{p \in \mathcal{P} : p \ni e} x_p \leq \lambda c(e).
$$

Thus,

$$
P[\text{congestion on } e > (1 + \epsilon)\lambda] = P\left[ \sum_{i=1}^{k} X_i > (1 + \epsilon)\lambda c(e) \right] P\left[ \sum_{i=1}^{k} X_i > (1 + \epsilon) \right] 
\leq e^{-\epsilon^2\lambda c(e)/(2+\epsilon)} \leq e^{-\epsilon^2/(2+\epsilon)}.
$$
29. Randomized rounding

29.3. Packing Multicommodity Paths

(a) is because, assuming integral capacities and $\lambda \geq 1$, we have $\lambda c(e) \geq 1$. For $\epsilon = O(\ln(m))$, the RHS is at most (say) $1/m^2$. Now we have

$$\Pr[\text{overall congestion } > O(\ln m)] \leq \sum_{e \in E} \Pr[\text{congestion on } e > O(\ln(m)) \lambda]$$

$$\leq m \cdot \frac{1}{m^2} = \frac{1}{m}$$

where (b) is by the union bound.

**Theorem 29.5.** One can compute a multicommodity flow with congestion at most

$$O(\ln m) \max\{\text{Opt}(29.3), 1\}$$

in polynomial time, where Opt(29.3) denotes the optimum value of LP (29.3).

With a refined Chernoff bound, one can improve the above bound to $O(\ln(m)/\ln \ln m)$. See [Che11] for these details.

### 29.3.1 Approximating the LP

While (29.3) would appear complicated, there are multiple ways to solve it. One way, which gives an exact solution, is via the Ellipsoid method\(^2\). Another approach is the approximation algorithm for pure packing based on Lagrangian relaxations that we discussed last time.

The first step is to convert (29.3) into a pure packing LP. For $i = 1, \ldots, k$, let $P_i$ denote the collection of paths from $s_i$ to $t_i$. Let $Q = P_1 \times \cdots \times P_k$. Each object in $Q$ consists of a bundle of paths, one for each commodity $(s_i, t_i)$. Consider the following LP for a given value of $\lambda$.

$$\max \sum_{q \in Q} x_q \text{ s.t. } \sum_{q = (p_1, \ldots, p_k) \in Q} x_q \sum_{1 \leq p_i \leq e} c(e) \leq \lambda c(e). \quad (29.4)$$

Observe that (29.4) is a packing LP. There is a fractional multicommodity flow with congestion at most $\lambda$ iff (29.4) has objective value at least 1. The best value of $\lambda$ can then be found by binary search. Given a solution $x$ to (29.4), we can easily convert it to the form of (29.3) by calculating, for each path, the fractional number of bundles that contains it.

\(^2\)We mentioned the Ellipsoid method (by Khachiyan) but have not really discussed it. While we will not use it today either, it is good to be aware of it.
To solve this LP (approximately) using the algorithm from our last discussion, we need to solve the Lagrangian relaxation. Let \( w \in \Delta^E \) be any probability distribution over the edges. The Lagrangian relaxation is to

\[
\text{maximize } \sum_{q \in \mathbb{Q}} x_q \text{ s.t. } \sum_{e} w(e) \sum_{q=(p_1, \ldots, p_k) \in \mathbb{Q}} x_q \sum_{i : p_i \ni e} 1 \leq \lambda \sum_{e} w(e)c(e).
\]

Consider the packing constraint. Rearranging sums

\[
\sum_{e} w(e) \sum_{q=(p_1, \ldots, p_k) \in \mathbb{Q}} x_q \sum_{i : p_i \ni e} 1 = \sum_{q=(p_1, \ldots, p_k)} \sum_{i=1}^k \sum_{e \in p_i} w(e).
\]

For fixed \( q = (p_1, \ldots, p_k) \), the quantity

\[
\sum_{i=1}^k \sum_{e \in p_i} w(e)
\]

(29.5)

can be understood to be the cost of a single bundle \( q \). Now, since every bundle has the same objective value (1), to solve the Lagrangian relaxation, it suffices to identify the bundle \( q \) of minimum cost. But (29.5) shows that the cost of a bundle \( q \) is the sum, over each path \( p_i \), of the length of \( p \) w/r/t the Lagrangian weights \( w \). Thus it suffices to compute, for each commodity \((s_i, t_i)\), the shortest \((s_i, t_i)\)-path w/r/t \( w \).

### 29.4 Exercises

**Exercise 29.1.** Consider the random cut, which was based on a threshold \( \theta \in (0, 1) \) chosen uniformly at random. Show that for all \( \theta \in (0, 1) \), the corresponding cut is a minimum \( \{s, t\} \)-cut.

**Exercise 29.2.** Using only the fact that the maximum \((s, t)\)-flow equals the minimum (discrete) cut, prove that if the capacities are integral, then the maximum flow can always be taken to be integral.

**Exercise 29.3.** Prove that set cover is NP-Hard by reduction from SAT.\(^3\)

**Exercise 29.4.** Recall that one can compute a \((1 - 1/e)\)-approximation to maximum coverage via the Greedy algorithm. Use this to obtain a \(O(\ln(n))\)-approximation to set cover.

\(^3\)Hint: Given a SAT formula with \( n \) variables and \( m \) clauses, create a set cover problem such that the formula is satisfiable iff there is a set cover of size \( n \). It is natural to think of an assignment (e.g., \( x_1 = \text{True} \)) as “covering” all the clauses that it satisfies. Some additional ideas are required to make sure we don’t take the “sets” corresponding to two assignments for a single variable (e.g., both \( x_1 = \text{True} \) and \( x_2 = \text{False} \)).
Exercise 29.5. Given a \((1 - 1/e - o(1))\)-approximation to maximum coverage by solving a linear relaxation and randomly sampling \(k\) sets based on the LP solution.
Chapter 30
Random Sums and Graphs

30.1 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, say, 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 $\frac{1}{2^{100}}$. 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. Even though the event of getting any one of these counts should be low, being far from average, 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 $2.818 \times 10^{-7}$.

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 a (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 $1.364 \times 10^{-10}$. 


We generalize the discussion 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 [Wika].)

We write $B \sim \mathcal{B}(n,p)$ to denote a random variable $B \in \{0,\ldots,n\}$ drawn from the binomial distribution $\mathcal{B}(n,p)$. The expected value of $B$ is $E[B] = pn$. The following lemma bounds the probability of $B$ being a multiplicative factor smaller than its mean, $pn$. Note that the probability decays exponential fast in the mean.

**Lemma 30.1.** Let $B \sim \mathcal{B}(n,p)$ and $\epsilon \in (0,1)$. Then

$$
P[B \leq (1 - \epsilon)pn] \leq e^{-\epsilon^2 pn/2}.
$$

**Proof.** We have

$$
P[B \leq (1 - \epsilon)pn] = \mathbf{P}[e^{-\epsilon B} \geq e^{-\epsilon(1-\epsilon)pn}] \leq \mathbf{E}[e^{-\epsilon B}] e^{-\epsilon(1-\epsilon)pn} = e^{\epsilon(1-\epsilon)pn} \mathbf{E}[e^{-\epsilon B}]
$$

by (a) Markov’s inequality. It remains to analyze $\mathbf{E}[e^{\epsilon B}]$. 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

$$
\mathbf{E}[e^{-\epsilon B}] = \mathbf{E}[e^{-\epsilon X_1 - \epsilon X_2 - \cdots - \epsilon X_n}] \overset{\text{(b)}}{=} \mathbf{E}[e^{-\epsilon X_1}] \mathbf{E}[e^{-\epsilon X_2}] \cdots \mathbf{E}[e^{-\epsilon X_n}],
$$

where (b) is by independent of the $X_i$’s. For each $X_i$, we have

$$
\mathbf{E}[e^{-\epsilon X_i}] = pe^{-\epsilon} + (1-p) = 1 + p(e^{-\epsilon} - 1) \\
\overset{(c)}{\leq} p \left( 1 - \epsilon + e^{-\epsilon / 2} ight) + (1-p) = 1 - (\epsilon - e^{-\epsilon / 2})p \\
\overset{(d)}{\leq} e^{-(\epsilon - e^{-\epsilon / 2})p}.
$$

417
Figure 30.1: A random graph sampled from $G(n,.01)$ [Von].

Here (c) uses the inequality $e^{-x} \leq 1 - x + x^2/2$ for all $x > 0$. (d) is by the inequality $1 + x \leq e^x$ for all $x$. Thus,

$$E[e^{-\epsilon B}] = E[e^{-\epsilon X_1}] E[e^{-\epsilon X_2}] \cdots E[e^{-\epsilon X_n}] \leq e^{-(\epsilon-\epsilon^2/2)pn}, \quad (30.2)$$

Putting everything together, we have

$$P[B \leq (1 - \epsilon)pn] \leq e^{\epsilon(1-\epsilon)pn} E[e^{-\epsilon B}] \leq e^{\epsilon(1-\epsilon)pn-(\epsilon-\epsilon^2/2)pn} = e^{-\epsilon^2 pn/2},$$

by (e) inequality (30.1) and (f) inequality (30.2), 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 30.1 and left as exercise 30.1.

**Lemma 30.2.** Let $B \sim \mathcal{B}(n,p)$ and $\epsilon \in (0,1)$. Then

$$P[B \geq (1+\epsilon)pn] \leq e^{-\epsilon^2 pn/3}.$$

### 30.2 Random graphs

Paul Erdős, inspired by Ramsey [Ram30] before him, had a series of work analyzing **random graphs**, producing a large body of results that 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

\footnote{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 $y$.  

418
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 properties. Today we will study the $G(n,p)$ random graph, sometimes called Erdős-Rényi graphs based on work by Erdős and Rényi [ER59, ER60]. A random graph from $G(n,p)$ is an undirected graph over $n$ vertices, where every edge is sampled independently with probability $p$. By now there is a large 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, Erdős and Rényi showed that these properties can vary dramatically with very small changes in $p$. Consider the following theorem.

**Theorem 30.3.** Consider a random graph $G \sim G(n,p)$ 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 $G$ has 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 a line of research has obtained a much more refined and detailed understanding than stated in theorem 30.3. We refer the reader to [Bol98, Chapter 7] for further details and other results in this area.

### 30.2.1 Overview of the proof

We will prove part 1 of theorem 30.3 in roughly three parts.

**Part 1: the gap theorem.** Observe that in theorem 30.3 above, regardless of the value of $p$, there are simply no “medium”-size components, like a component of size $\sqrt{n}$ or of size $n/\log(n)$. The intermediate sizes are ruled out by the following “gap theorem”.

419
Lemma 30.4. 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 \( G(n,p) \), 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 30.4 theorem in section 30.3. The proof makes a surprising connection to our discussion on random sums in section 30.1.

Part 2: Existence of a large component. lemma 30.4 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 30.4 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 30.5. 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 30.5 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. In conjunction with the gap theorem (lemma 30.4), which rules out all intermediate sizes, this implies that there is at least one giant component of size \( \Omega(\epsilon^2 n) \).

By lemma 30.5, 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 \( G(n,p) \) 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. That is, we almost certainly connect $C_1$ and $C_2$. Since there is a limited number of 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 30.3 for $c > 1$.

c < 1. The proof for $c < 1$ is simpler and only requires the ideas underlying lemma 30.4. See section 30.3.1.

30.2.2 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 30.6.** 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 the $v$’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 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.

### 30.3 A gap in component size

In this section we prove lemma 30.4, 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 [Karp90]. His proof is also described in [Bol98]. We will also reuse some of the ideas in the proof to analyze the $p < 1/n$ in section 30.3.1. We first restate lemma 30.4 for the reader’s convenience.

**Lemma 30.4.** 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 $G(n,p)$, 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 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” $v$ 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 $i$th 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 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 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 $i$th iteration, is $B_{i-1} \neq A_{i-1}$, 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 $|A_i| \neq i$, $A_{i+1}$ 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 $x \not\in A_i$ if and only if $x$ 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., $|A_i| \sim \mathcal{B}(n-1, 1-(1-p)^i)$.

**Lemma 30.7.** Let $i \leq \frac{\epsilon n}{2((1+\epsilon))}$. Then $\mathbb{E}[|A_i|] \geq (1+\epsilon)i$.

**Proof.** We have

$$(1-p)^i \leq e^{-ip} \leq 1-itp + (tp)^2 \leq 1-itp + \epsilon tp/2 = 1-(1-\epsilon/2)ipt.$$  

where (a) is because $ipt = (1+\epsilon)i/n \leq \epsilon/4$. Thus

$$\mathbb{E}[|A_i|] = 1 + (1-(1-p)^i)(n-1) \geq (1-\epsilon/2)iptn \geq (1+\epsilon)i.$$ 

$\blacksquare$

**Lemma 30.8.** Let $i \leq \frac{\epsilon n}{2(1+\epsilon)}$. Then $\mathbb{P}[|A_i| \leq i] \leq e^{-\epsilon^2i/8}$.

**Proof.** We have

$$\mathbb{P}[|A_i| \leq i] \overset{(a)}{=} \mathbb{P}[|A_i| \leq (1-\epsilon/2)\mathbb{E}[|A_i|]] \overset{(b)}{\leq} e^{-\epsilon^2/8}.$$ 

Here (a) is by lemma 30.7. (b) is by the tail inequality on binomial distributions, lemma 30.1.

Let $I = \{i \in \mathbb{N} : 32 \ln(n)/\epsilon^2 \leq i \leq \epsilon n/2((1+\epsilon))\}$. For all $i \in I$, we have

$$\mathbb{P}[|A_i| \leq i] \leq 1/n^4.$$ 

By the union bound, we have

$$\mathbb{P}[|A_i| > i \text{ for all } i \in I] \geq 1 - \sum_{i \in I} \mathbb{P}[|A_i| \leq i] \geq 1 - 1/n^3.$$ 

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 30.4.

### 30.3.1 Probabilities $< 1$

Suppose instead that $p = (1-\epsilon)/n$. Then we have $\mathbb{E}[A_i] \leq (1-\epsilon/2)i$ unless $i$ is very close to $n$. In particular, for $i = O(\log(n)/\epsilon^2)$, the probability of $A_i > i$ is $1/poly(n)$. Thus we see that all components will have size $O(\log(n)/\epsilon^2)$ with high probabilities, establishing part 2 of lemma 30.4.
30.4 Galton-Watson branching processes

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 30.5, which we first restate for the reader’s convenience.

**Lemma 30.5.** 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 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 30.9.** 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. 30.2.

**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

\[
\begin{align*}
p_0 &= 1, \\
p_1 &= 1 - (1 - 1/k)^k \geq 1 - e^{-1} \geq 0.63, \\
p_2 &= 1 - (1 - 0.63/k)^k \geq 1 - e^{-0.63} \geq 0.467, \\
p_3 &\geq 1 - (1 - 0.467/k)^k \geq 1 - e^{-0.467} \geq 0.373 \geq 1/3.
\end{align*}
\]

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,

\[
p_{i+1} \geq 1 - (1 - 1/i k)^k \geq 1 - e^{-1/i} \quad \text{(a)}
\]

Here (a) is by induction. (b) applies the inequality \( e^x \leq 1 + x + \frac{1}{2}x^2 \) for \( x \leq 0 \).

\[\blacksquare\]

### 30.4.1 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 \( \leq h \), for any \( h \leq \epsilon n/(1 + \epsilon) \). Recall the sets \( B_0, B_1, B_2, \ldots \) from section 30.3. Given that \( |B_i| \leq h \), we can think of \( B_{i+1} - B_i \) as adding (at least) \( n/(1 + \epsilon) \) children each with probability \( p = (1 + \epsilon)/n \). Either we find new elements for all \( h \) rounds - which forces \( |B_i| \geq h \) - or we hit \( |B_i| = h \) at some point \( i < h \). Thus the odds of \( v \) acquiring \( h \) vertices in its connected component is at least the odds produced by theorem 30.9 for this value of \( h \); namely, \( 1/h \). This gives us lemma 30.5.

### 30.4.2 A more general analysis

One can consider a more general model as follows. Let \( X \in \mathbb{Z}_{\geq 0} \) be a random variable taking nonnegative integer values. For each node at a generation \( i \), we sample an independent copy of \( X \), and generate this many children in the next generation. Here we have the following remarkably precise theorem.

**Theorem 30.10.** Let \( n_i \) denote the number of children in the \( i \)th generation in the process described above.

1. If \( \mathbb{E}[X] < 1 \), then \( \lim_{i \to \infty} \mathbb{P}[n_i = 0] = 1 \).

2. If \( \mathbb{E}[X] > 1 \), then \( \lim_{i \to \infty} \mathbb{P}[n_i = 0] = q \), where \( q \) is the unique solution to \( x = \sum_{i \geq 0} \mathbb{P}[X = 0] x^i \).

See the lecture notes by Sinclair [Sin20] for a proof.
30.5 Exercises

Exercise 30.1. Prove lemma 30.2. (Here the important part is not the constant, $1/3$ – any constant $c > 0$ is already interesting.)
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

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 referred 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

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 Induction

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.

They keep 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 the 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.

Let us try to prove by induction that the instructions will scale to any staircase. Of course there is something a little awkward about proving something like this that's not as formal as a purely mathematical claim. This is meant as an
C. Mathematical Background

C.1. Induction

Kent Quanrud
Spring 2022

introductory example to develop intuition. We refer to [Che15; Eri] for a more formal presentation.

Besides, there is no shortage of mathematical or computational examples elsewhere in these notes. Recursion is induction, embodied in a computer. (At least, any recursion that is correct). Dynamic programming and divide-and-conquer are more specialized algorithmic forms of induction with considerations for both correctness and running time. Every exercise about these topics is also an exercise in induction.

With that said, we proceed to the proof.

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

In this proof we take for granted that the instruction of using one foot at a time works to go up one step. The only thing to prove is that repeating it will extend to long staircases, no matter how long.

We will prove the claim by induction on \( n \). The main issue is that our argument needs to be independent of \( n \); that is, we can’t write a separate proof for every value of \( n \). The motivation behind the principle of induction is that assuming the claim for \( n - 1 \) makes it easy to prove the claim for \( n \), with \( n \) still held generic. So as long as we can prove how to prove the claim for \( n \) given the claim for \( n - 1 \), where \( n \) is again generic, then we know automatically that iterating the proof for all \( n \) will eventually cover all positive integers.

The only exception is \( n = 1 \), since we have no “\( n - 1 \)” to rely on. So this is a base case that we prove from scratch. Although we do not leverage an induction hypothesis, \( n = 1 \) is obviously very simple. It is typically the case that base cases are simpler.

Base case: suppose \( n = 1 \). If you follow the instructions once, then you climb one stair. Since the staircase has one stair, this one step completes the staircase.

General case: Suppose \( n > 1 \). Note that we do not know the value of \( n \), although in our minds, we may sense that it shouldn’t really matter. We probably sense that it should behave just like \( n - 1 \) steps, with one extra step. Here’s how we put that intuition into words:

We follow the instructions and take one step onto the next stair. The remaining staircase is a staircase with \( n - 1 \) steps. By induction on \( n \), the instructions will work for any \((n - 1)\)-step staircase. We apply this to the \( n - 1 \) steps remaining from the \( n \)-step staircase, and conclude that the instructions will also scale an \( n \) step staircase as well.

That completes the proof. The principle of induction assures that by addressing the base cases, and explaining how to transition from one proven case to the next unproven case, we automatically address all cases. This might seem obvious for staircases but the point is that the principle applies generally to much more complicated and abstract situations.
C. Mathematical Background

C.1. Induction

C.1.1 Towers of Hanoi

So much for stairs. Hopefully it is a simple enough example that the point is clear, but my only concern is that it is too simple. Instead of understanding staircases by induction / recursion, one can also easily “see through” the induction and conceptualize the instructions iteratively as a loop, so to speak. So here is a famous puzzle that is too complicated to easily understand and solve without recursion and induction.

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. In doing so we must obey the rule that a ring can only be placed on top of another ring with greater radius.

The challenge is as follows: is it possible to move all the rings from post A to post B, subject to these rules? We encourage the reader to pause and try to solve this puzzle.

We prove by induction on \( n \) the following claim. Given a stack of \( n \) rings on post labeled post A, and two empty posts labeled posts B and C, one can move all \( n \) rings from post A to post B.

Take \( n = 0 \). Then the claim is satisfied vacuously. (All the rings are already moved to post B.)

Now take \( n \geq 1 \). We assume by induction on \( n \) that it is possible to move \( n - 1 \) rings from one post to another. By induction, we move the top \( n - 1 \) rings from post A to post C. We then move the bottom ring from post A to post B. By induction, we move the top \( n - 1 \) rings from post C to post B.

That’s it. To further elaborate on the induction step – moving the top \( n - 1 \) rings form A to C, and then from C to B – recall that the \( n \)th ring is allowed to be under any other ring and, when considering only the top \( n - 1 \) rings, can be completely ignored. What remains is an \( (n - 1) \)-ring towers of Hanoi problem that is already solved by induction.
C.1.2 Do I really need induction and recursion?

This style of thinking is abstract and naturally met with resistance. Why adopt new styles of thinking when you have already come so far? To be able to go farther, of course.

Some of our problems are simple enough that one can sort of “see the whole thing” without taking a top-down perspective. However the solutions that come out of this, even when correct, tend to be harder to understand and verify because the thinking process is not very structured.

A bigger issue is that at some point problems are sufficiently complicated that it’s harder to see everything all at once. It is helpful to break down the problem into smaller steps, having faith that repeating these steps eventually solves the entire problem. Moreover it is much easier to analyze and prove the correctness of a single step satisfying declared invariants, and then rely on induction to automatically scale that argument to the entire algorithm, then it is to analyze the whole algorithm all at once.

This is what induction really does for us. It is a sound principle for breaking down big problems into locally verifiable subproblems. It allows us to not only see solutions to harder problems, but also feel more confident that they are correct.

C.2 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.2.1 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.2.2 The integration trick.**

Take a sum such as \( \sum_{i=1}^{n} 1/i \). (The “\( n \)th 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.2.3 Geometric series.**

A **geometric sum** is any sum of the form
\[ \sum_{i} \alpha^i \]
for a fixed value \( \alpha \). 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

\[
\frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \frac{15}{16}, \frac{31}{32}, \cdots
\]

Each partial sum halves the gap between the previous partial sum and 1, and the whole sequence 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

\[
(1-\alpha) \sum_{i=0}^{n} \alpha^i = \sum_{i=0}^{n} \alpha^i - \alpha^{i+1} \overset{(a)}{=} 1 - \alpha^{n+1}.
\]

Here (a) is because we have a **telescoping series**: the $-\alpha^{i+1}$ term for a fixed $i$ cancels out with the $\alpha^i$ 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 $\alpha$ 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 simply 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

$$\begin{align*}
(\alpha - 1) \sum_{i=0}^{n} \alpha^i &= \sum_{i=0}^{n} \alpha^{i+1} - 1 = \alpha^{n+1} - 1. \\
\sum_{i=0}^{n} \alpha^i &= \frac{\alpha^{n+1} - 1}{\alpha - 1}.
\end{align*}$$

In particular, when $\alpha$ 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.

C.3 Additional notes and references

The following url, particularly figure 6 regarding some common sums, was recommended by a student in the class: https://everythingcomputerscience.com/algorithms/Algorithm_Analysis.html. You might try to re-derive Figure 6 using the ideas in appendix C.2 above.
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 $S \subseteq V$ 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 $3$CNF 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 $3$CNF 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
$3$CNF 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 $\bigcup_i S_i$. 

439
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


Bibliography


Bibliography


[Von] Vonfrisch. Erdos generated network-p0.01.jpg. URL: https://commons.wikimedia.org/wiki/File:Erdos_generated_network-p0.01.jpg.


Part II

Course Materials
Appendix E

Homework Assignments
E.1 Homework 1

- Due 12:00 PM on Monday, January 24.
- There are also multiple choice questions on Gradescope due at the same time.
- Please see appendix L.6 (in the syllabus) for homework policies.
- Recall that some student submissions will be selected and distributed to the class as solutions (see appendix L.6.7). If you would like to be excluded from consideration, please state so explicitly at the top of each problem.
- Please leave off your student ID (which Gradescope is already aware of).
- You may use standard data structures (e.g., stacks, queues and linked lists) and assume their standard running times per operation as a fact. You may also invoke algorithms from lecture, recitations, or previous homeworks (e.g., merge-sort) as a black box and assume as fact whatever bounds have already been proven for them.
- 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?

1. Exercise 1.4.
2. Exercise 1.5.
3. Exercise 2.8.
4. Exercise 2.9.

---

1You should refrain from using hash tables since technically your running time analysis would require additional probabilistic qualifications that we have not yet discussed. Keep in mind that it is usually not a bottleneck to re-number your input as a pre-processing step and then use an array instead.

2This problem was revised on the morning of Saturday, Jan 15. (While most people probably had not started anyway, you might check that your solution is for the latest version.)
E.2 Homework 2

- Due 3:00PM on Monday, January 31 (no exceptions).
- Please see appendix L.6 (in the syllabus) for homework policies.
- Recall that some student submissions will be selected and distributed to the class as solutions (see appendix L.6.7). If you would like to be excluded from consideration, please state so explicitly at the top of each problem.
- Please leave off your student ID (which Gradescope is already aware of).
- 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?

1. Exercise 3.5.
2. Exercise 4.2.
3. Exercise 4.7
4. Exercise 4.6.3

Exercise 3.5. 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 4.2. Recall that a palindrome is a string that reads the same forwards and backwards. For example,

\[ \text{mom}, \text{dad}, \text{racecar}, \text{and gohangasalamiimalasagnahog} \]

are all palindromes. Consider the problem where you are given a string \( A[1..n] \) as input, and the goal is to find the (length of the) longest palindrome that is a subsequence of \( A \). 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.

3This problem is tricky.
Exercise 4.7. Let $A[1..m]$ be a sequence of integers. Recalling that a subsequence is (strictly) increasing if each successive integer is strictly greater than the previous.

Here we consider two variants of the longest increasing subsequence problem. In the first problem, we want to find the longest increasing subsequence where the sum of integers in the subsequence is even. In the second problem, we want to find the longest increasing subsequence where the sum of integers in the subsequence is odd.

For both of these 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.

Exercise 4.6. 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.
E.3 Homework 3

- Due Tuesday, February 8, at 11:59PM.
- Please see appendix L.6 (in the syllabus) for homework policies.
- Recall that some student submissions will be selected and distributed to the class as solutions (see appendix L.6.7). If you would like to be excluded from consideration, please state so explicitly at the top of each problem.
- Please leave off your student ID (which Gradescope is already aware of.)
- 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?

1. Canceled: Exercise 5.4
2. Exercise 6.5.

Exercise 5.4. Recall the dominating set problem from section 5.3. 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.\(^5\)

\(^4\)Canceled because Kent is a dummy who accidentally posted a solution.

\(^5\)Of course, anyone who has already solved the general case automatically solves the special case where no two intervals are nested.
2. The minimum weight dominating set problem in trees.

**Exercise 6.5.** Let $G = (V, E)$ be 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: (red, white, blue), (white, blue, red), or (blue, red, white). We say that a vertex $s$ can **un-patriotically reach** a vertex $t$ if we can walk from $s$ to $t$ via an un-American walk.

Consider the problem of deciding whether a vertex $s$ can un-patriotically reach $t$. 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 6.7.** 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 6.8.** 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, $\pi : V \rightarrow [n]$ 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. Partial credit will be awarded to an algorithm that works in nearly linear time on DAGs.\(^6\)

---

\(^6\)Previously we mistakenly wrote linear time (without the “nearly”); but the solution we had in mind involves logarithmic factors.
E.4 Homework 4

- Due 3:00 PM on Monday, February 14.
- Please see appendix L.6 (in the syllabus) for homework policies.
- Recall that some student submissions will be selected and distributed to the class as solutions (see appendix L.6.7). If you would like to be excluded from consideration, please state so explicitly at the top of each problem.
- Please leave off your student ID (which Gradescope is already aware of.)
- 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?

1. Exercise 7.7.
2. Exercise 7.10.
3. Exercise 8.2.

(Just for fun; not to be submitted.) Exercise 8.3.

Exercise 7.7. 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 in $G$ for which the labels of the vertices are strictly increasing.

1. (7 points) 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. (3 points) 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.

Exercise 7.10. 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 $r : E \rightarrow \mathbb{R}_{>0}$. 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. You may assume for simplicity that $G$ has no sinks.

---

7Recall that a path is a walk that does not repeat vertices.
1. Given an integer \( k \leq n \), the goal is to compute a walk of length \( k \) maximizing the sum of rewards along that walk.\(^8\) (If you repeat an edge, you get the same reward each time you repeat the edge.) 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. Here we also incorporate a discount rate. Let \( \alpha \in (0, 1) \) be given. Given a walk with edges \( e_1, e_2, \ldots, e_k \), 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 \( k \) maximizing the discounted total reward of the walk. 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 8.2.** 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 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 8.3.** The racetrack problem in Chapter 5 of [Eri19].

\(^8\)There is always a \( k \)-edge walk under the assumption that \( G \) has no sinks.
E.5 Homework 5

- Due 3:00 PM on Monday, March 7.
- Please see appendix L.6 (in the syllabus) for homework policies.
- Recall that some student submissions will be selected and distributed to the class as solutions (see appendix L.6.7). If you would like to be excluded from consideration, please state so explicitly at the top of each problem.
- Please leave off your student ID (which Gradescope is already aware of.)
- 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?

1. Exercise 11.2.
2. Exercise 11.3.

Exercise 11.2. Here we assume the same model as exercise 11.1 – an array $A[1..n]$ of incomparable elements\(^9\) – 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.\(^{10}\)

\(^9\)In addition to disallowing sorting, no hash tables are allowed as well.

\(^{10}\)Your algorithm should be faster than $O(n^2)$ (which is the trivial running time) for, e.g., $k = n^{1/3}$.
Exercise 11.3. 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 $1 \leq i \leq j \leq n$, $i \leq A[i] \leq A[j] \leq j$, 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 $A$.\(^1\)

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.

Exercise 11.6. 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.\(^2\) You may assume that all the integers in $X$ are distinct (amongst themselves), and that all the integers in $Y$ are distinct (amongst themselves).\(^3\)

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 $Y$.\(^4\)

---

\(^1\)There may be more than one fixed point; any fixed point is fine.

\(^2\)That is, for fixed $z$, there is only one choice of $x \in X$ and one choice of $y \in Y$ such that $z = x + y$.

\(^3\)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.

\(^4\)It might be helpful to work through very simple examples such as $X = Y = \{1\}$ and $X = Y = \{0, 1\}$.
3. (2 pt.) Suppose we now that 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.\textsuperscript{15}

*Hint: For part 2, try “changing the input” to an algorithm from section 11.4 (that is not the FFT, at least directly).*

\textsuperscript{15}It might help to work through $k = 4$ to build your intuition.
E.6 Homework 6

- Due 3:00 PM on Monday, March 21.
- Please see appendix L.6 (in the syllabus) for homework policies.
- Recall that some student submissions will be selected and distributed to the class as solutions (see appendix L.6.7). If you would like to be excluded from consideration, please state so explicitly at the top of each problem.
- Please leave off your student ID (which Gradescope is already aware of.)
- 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?

1. Exercise 13.9
2. Exercise 13.11
3. Exercise 14.3

Exercise 13.9. 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.\(^\text{16}\)

Exercise 13.11. For $p > 0$, the $L_p$-norm of a vector $x \in \mathbb{R}^k$ is defined by

$$
\|x\|_p \overset{\text{def}}{=} \left( \sum_{i=1}^{k} |x_i|^p \right)^{1/p}.
$$

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

$$
\|x\|_\infty \overset{\text{def}}{=} \max_i |x_i|.
$$

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

\(^{16}\)It may be helpful to obtain a running time generically as a function of $k$, and then optimize $k$.\]
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 $w$,

$$
\|w\|_p \overset{\text{def}}{=} \left( \sum_{i=1}^{k} |\ell(e_i)|^p \right)^{1/p} .
$$

Analogously we define the $L_0$ and $L_\infty$ 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

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 $L_\infty$ -distance from $s$ to $t$.

**Exercise 14.3.** 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 computing a single $(a,b,c)$-path (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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.
E.7  Homework 7

- Due 3:00 PM on Monday, March 28.
- Please see appendix L.6 (in the syllabus) for homework policies.
- Recall that some student submissions will be selected and distributed to the class as solutions (see appendix L.6.7). If you would like to be excluded from consideration, please state so explicitly at the top of each problem.
- Please leave off your student ID (which Gradescope is already aware of.)
- 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?*

1. Exercise 16.2
2. Exercise 16.1
3. Exercise 15.4

**Exercise 16.2.** 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) $t \in [m]$, there is a subset of times $T_t \subseteq [n]$ of times when they can attend PSO.
2. For each time slot $j \in [n]$, there is a subset of rooms $R_j \subseteq [m]$ 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.
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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 16.1. Suppose we are given 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 15.4. 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 15.2.

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 \( \lambda / m \).

2. Describe a \( O(m + n \log(n)) \)-time algorithm to find the path described above.\(^{17}\)

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 \( \lambda \) denotes the value of the maximum flow.\(^{18,19}\) (The algorithm does not know the true value of \( \lambda \) \textit{a priori}.)

\(^{17}\)\(O(m \log n)\) 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.

\(^{18}\)This is polynomial with respect to the bit complexity of the input.

\(^{19}\)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).
E.8 Homework 8

- Due 3:00 PM on Monday, April 18.
- Please see appendix L.6 (in the syllabus) for homework policies.
- Recall that some student submissions will be selected and distributed to the class as solutions (see appendix L.6.7). If you would like to be excluded from consideration, please state so explicitly at the top of each problem.
- Please leave off your student ID (which Gradescope is already aware of.)
- 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?*

1. Exercise 19.2.
2. Exercise 19.3.
3. Exercise 20.2.

**Exercise 19.2.** 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.

$$\text{multipop}(\text{stack } S, k)$$

1. $L \leftarrow \text{empty linked list}$
2. For $i = 1, \ldots, k$ as long as $S$ is not empty:
   - Let $x = S.\text{pop}()$, 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 19.3.** Section 19.5.1 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 only two immutable stacks.

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.)

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 while the back stack, 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.

Analyze the double-ended queue data structure described above and prove that each of the four double-ended queue operations take $O(1)$ amortized time.

**Exercise 20.2.** In the push-relabel framework, we have seen that different strategies for selecting active vertices can lead to better running times. Here we will consider a variation of push-relabel that we will call greedy push-relabel. The idea is to repeatedly select the push of maximum total size. We will assume the capacities are integral (which will be important for the proof).

$\text{greedy-push-relabel}(G = (V, E), c : E \to \mathbb{Z}, s, t)$

1. Initialize a preflow $f$ and levels $\ell : V \to \mathbb{Z}_{\geq 0}$ (as usual).
2. Until $f$ is a flow:
   A. Let $v \in V - \{s, t\}$ have maximum excess $\hat{f}(v)$.
   B. Push or relabel from $v$ (as usual).
3. Return $f$.

This exercise asks you to analyze greedy-push-relabel as follows. We point out that the upper bounds from section 20.1.3 come for free since greedy-push-relabel is a special case of the generic push relabel algorithm. You may also want to look at section 20.3 for inspiration.
1. Suppose the maximum total excess over all non-terminals at a fixed point in time is $X$; i.e., $\sum_{v \in V - \{s, t\}} \hat{f}(v) = X$. Prove that there are at most $O(n^2)$ non-saturating pushes before the total excess over all non-terminals is less than $X/2$.

2. Prove an upper bound, as small as possible, on the total number of push-relabel operations made by greedy-push-relabel.

   You will probably want to use the previous part in your argument. The integrality of the capacities should play a role in your analysis.

   (Food for thought: A full implementation of greedy-push-relabel would require some data structures so you can quickly identify the next active vertex. How might you implement the algorithm? What is your actual running time?)
E.9 Homework 9

• Until this disclaimer is removed, these problems are only tentative.\(^{20}\)
• Due 3:00 PM on Monday, April 25.
• Please see appendix L.6 (in the syllabus) for homework policies.
• Recall that some student submissions will be selected and distributed to the class as solutions (see appendix L.6.7). If you would like to be excluded from consideration, please state so explicitly at the top of each problem.
• Please leave off your student ID (which Gradescope is already aware of.)
• 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?

Exercise 21.8. Let \( G = (V, E) \) be an undirected graph. For \( k \in \mathbb{N} \) a \( k \)-cut is a set of edges whose removal disconnects the graph into at least \( k \) connected components. Note that for \( k \geq 3 \), the minimum \( k \)-cut problem cannot easily be reduced to \( (s, t) \)-flow. In fact, the problem is NP-Hard when \( k \) is part of the input.

1. Design and analyze a randomized algorithm that runs in \( n^{ck} \) randomized time for a constant \( c \) and returns the minimum \( k \)-cut with high probability. (The smaller the constant \( c \), the better.)

2. How does your algorithm relate to the preceding statement that \( k \)-cut is NP-Hard when \( k \) is part of the input?

Exercise 22.4. The count-min-sketch data structure allows us to estimate the relative frequency of each element up to an \( \varepsilon \)-additive factor with probability of error \( \leq 1/\text{poly}(n) \) with \( O(\log(n)/\varepsilon) \) space. The original motivation, however was to also obtain a list of \( \varepsilon \)-heavy hitters. Explain how to adjust the count-min-sketch to also maintain a collection that contains all of the \( \varepsilon \)-heavy hitters while holding onto at most \( O(1/\varepsilon) \) items from the stream at any point in time, with probability of error \( \leq 1/n^2 \). Here your algorithm may also report a limited number of items with relative frequency in the range \([\varepsilon/2, \varepsilon)\), but no items with relative frequency less than \( \varepsilon/2 \).

Exercise 21.10. 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.

\(^{20}\)This draft is prepared for our weekly staff meeting where we discuss and revise the problems. Adjustments are also made based on what is actually covered in class.
For each light bulb $L_i$, there are 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 $j \in B_i$, 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, or (b) prove that a polynomial time algorithm implies a polynomial time algorithm for SAT.

1. (7 points) Decide if there exists a way to flip the switches to turn on all the light bulbs.

2. (3 points) Decide if there exists a way to flip the switches to turn on at least three-fourths of the light bulbs.
Appendix F

Recitation Problems
F. Recitation Problems

F.1. Recitation 1

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.¹

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. 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 1.3. 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 \( i \)th 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 \( (x_1, \ldots, x_n) \).

None of the recitations discussed any of the remaining questions and we plan to discuss them (in part) the following week.

Exercise 2.2. 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.

¹Some solutions are sketched in appendix G.
Exercise 2.7 for $k = 3$. Recall that the $k$-SAT problem is the special case of SAT where $f(x_1, \ldots, x_n)$ is a CNF with exactly $k$ variables per clause.

1. Show that a polynomial time algorithm for 3-SAT implies a polynomial time algorithm for 4-SAT.

2. Show that a polynomial time algorithm for 4-SAT implies a polynomial time algorithm for 3-SAT.
F.2 Recitation 2

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.

Exercise 2.7 for \( k = 3 \). Recall that the \( k \)-SAT problem is the special case of SAT where \( f(x_1, \ldots, x_n) \) is a CNF with exactly \( k \) variables per clause.

1. Show that a polynomial time algorithm for 3-SAT implies a polynomial time algorithm for 4-SAT.

2. (Time permitting) Show that a polynomial time algorithm for 4-SAT implies a polynomial time algorithm for 3-SAT.

Exercise 4.1. Let \( A[1..n] \) be an array of \( n \) numbers. Consider the problem of computing the length of the longest subsequence of \( A \) that is strictly increasing. Here a sequence of numbers \( x_1, \ldots, x_k \) is strictly increasing if \( x_i < x_{i+1} \) for \( i = 1, \ldots, k - 1 \). 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 3.4. Consider the following variation of subset sum which allows for an additive error of 1. We’ll call it \textit{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 2.2. Suppose one had access to a polynomial time algorithm that solves the decision version of Boolean satisfiability (outputting \texttt{true} or \texttt{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 3.3. Consider the following special case of subset-sum called the \textit{partition problem}. The input consists of \( n \) numbers \( x_1, \ldots, x_n \in \mathbb{N} \). The goal is to partition the \( n \) numbers into two parts so that the sums of each part are equal. 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.

--

\(^2\)Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby

\(^3\)A solution is given on page 497.
F.3 Recitation 3

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.4

Exercise 6.4. Let $G = (V, E)$ be 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. We say that a vertex $s$ can patriotically reach a vertex $t$ if there is an American walk from $s$ to $t$.

Consider the problem of deciding whether a vertex $s$ can patriotically reach $t$. 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.5

Exercise 5.3. 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.
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 $k$.

---

4Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby
5Hint: don’t change the algorithm; change the graph!
F.4 Recitation 4

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.\(^6\)

**Exercise 7.5.** The following two 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. For both 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. Given a directed graph \(G\), compute a walk that visits the greatest number of distinct vertices.

2. Given a directed graph \(G\), compute a walk that traverses the greatest number of distinct edges.

**Exercise 8.1.** Let \(G = (V, E)\) be an unweighted, directed graph, and 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. 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 7.6.** Let \(G = (V, E)\) be a directed graph with vertex weights \(w : V \rightarrow \mathbb{R}\). 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.

\(^6\)Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby
F.5 Recitation 5

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.7

**Exercise 8.1.** Let $G = (V, E)$ be an unweighted, directed graph, and 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. 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 9.1.** Recall that in foreign exchange, different currencies can be exchanged at 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 arbitrary 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise F.1 ([Eri19, Exercise 3.6])**. A **shuffle** of two strings $X$ and $Y$ is formed by interspersing the characters into a new string, keeping the characters of $X$ and $Y$ in the same order. For example, the strings “PRODGYRNAMAMMIINCG” and “DYPRONGARMAMMICING” are both shuffles of $X = "DYNAMIC"$ and $Y = "PROGRAMMING"$.

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 7.8.** Let $G = (V, E)$ be a directed graph where each edge is colored red, white, or blue. An American path is a path where the edge colors alternate red, white, blue. The American Hamiltonian path problem is to decide if there is an

---

7Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby
American path that visits all the 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 8.4. 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. 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.
F.6 Recitation 6

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.  

Exercise 11.1. 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. Prove your algorithm is correct by induction, where the recursive spec is your induction hypothesis.
4. Analyze the running time of your algorithm.

Exercise 11.5. 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 $k$th largest element among the combined elements of $A$ and $B$. For simplicity you make assume that all the elements are distinct.

1. Define a recursive spec for a recursive algorithm to solve this problem.
2. Show that if either $m$ or $n$ is $\leq 5$ then we can find the $k$th 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.

---

8Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby
9There is also a well-known $O(n)$ time algorithm which we are not asking for.
10By 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.
F.7 Recitation 7

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.11

The following definitions and results from the notes may be helpful.

**Span.** Let $S \subset E$ and $e \in E$. We say $S$ spans $e$, or $e \in \text{span}(S)$, if the endpoints of $e$ are connected by (the edges in) $S$.

**Lemma 13.5.** Let $e \in E$. Then the following conditions are equivalent.

1. Every minimum weight spanning tree contains $e$.
2. For every set of edges $S \subseteq E - e$ that spans $e$ (and doesn’t include $e$),

   $$w(e) < \max_{f \in S} w(f).$$

**Good edges.** An edge $e$ is good if it satisfies condition (2) above.

**Lemma 13.7.** 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.

**Lemma 13.10.** Let $S \subset V$ be any set of vertices. Then the minimum weight edge in the cut induced by $S$ is a good edge.

**Exercise 13.2.** Consider the following algorithm that claims to compute the MST (in a connected graph).

\[
\text{decremental-MST}
\]

1. Sort and index $E = \{e_1, \ldots, e_n\}$ in decreasing order of weight.

2. For $i = 1, 2, \ldots, n$:
   1. Unless $E - e_i$ is disconnected:
      1. $E \leftarrow E - e_i$.

3. Return $E$.

Either

---

11Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby
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.

**Exercise 13.6.** Let $G = (V, E)$ be an undirected graph with distinct nonnegative edge lengths. The bottleneck length of a path in $G$ is the length of the longest edge in the path. Prove that the MST of $G$ contains the path of shortest bottleneck length between every pair of points.

**Exercise 13.7.** 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 $T$, $\max_{e \in T} w(e)$. Consider the problem of computing the minimum

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.)

**Exercise 14.1.** 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Exercise 14.2.** 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 $a, b, c \in V$.

---

12Here’s step 1: compute the median edge weight in $O(m)$ time.
F.8 Recitation 8

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.\textsuperscript{13}

The augmenting-path algorithm of Ford and Fulkerson [FF56] computes the maximum flow in an integer-capacitated graph in $O(m\lambda)$ time, where $\lambda$ is the value of the maximum flow. The shortest-augmenting-path algorithm of Edmonds and Karp [EK72] computes the maximum flow in general capacitated graphs in $O(m^2n)$ time.

There are better maximum flow running times (including some striking recent results), but to keep things simple, we will stick to the running times established in class. A few more running times may come up in class or homework at which point we will extend the list of available running times.

**Exercise 14.2.** 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 15.2.** 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 of $f$, 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 15.1.** 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.

\textsuperscript{13}Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby
F.9 Recitation 9

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.¹⁴

Review problems. Each of the following problems are from [Eri19] and are taken from the list of “Midterm 2 practice problems” (appendix I). Below we also state the kind of instructions one might expect on the exam.

For “identify the subproblem” type solutions, including all dynamic programming and divide-and-conquer algorithms (and recursive algorithms in general), we require the following parts explicitly.

1. Recursive spec (i.e., a sentence) definition of the subproblem.
2. How to extract the final solution from the subproblems.
3. Pseudocode implementation of the recursive spec.
4. The total running time of your algorithm. If you are using dynamic programming, explicit mention that you are using DP/caching solutions to obtain the claimed running time. For other recursive algorithms including divide-and-conquer, you should state the running time recurrence that you are solving.

We will treat your recursive spec as an induction hypothesis to justify the correctness of the algorithm, and no explicit proof is otherwise required. (Some justification / comments might help the grader’s understanding in the event of partial credit.)

For “change the input” type solutions, where a problem \( X \) is reduced to problem \( Y \), you should describe an algorithm where an algorithm for problem \( Y \) is used as a black box to solve problem \( X \). (This can be reducing a given problem to an established polynomial time algorithm, or reducing a known NPHard problem to a given problem.) The correctness often involves establishing an “if and only if”. (E.g., the answer to decision problem \( X \) is true iff the answer to the instance problem \( Y \) is true, or there is a bipartite matching of size \( k \) iff there are \( k \) edge-disjoint paths.) Justify both directions explicitly (even if you think its obvious). For polynomial time algorithms you also need to analyze the running time; when proving a problem is hard, indicate somewhere that you are proving that problem \( Y \) is NPHard.

Exercise 1.32 in [Eri19]. Suppose we are give an array \( A[1..n] \) with the special property that \( A[1] \geq A[2] \) and \( A[n-1] \leq A[n] \). We say that an element \( A[x] \) is a local minimum if its less than or equal to both its neighbors, or more formally, if \( A[x-1] \geq A[x] \) and \( A[x] \leq A[x+1] \). For example, there are six local minima in the following list (which are boldfaced):

¹⁴Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby
We can obviously find a local minimum in $O(n)$ time by scanning the array. Describe and analyze that finds a local minimum in $O(\log n)$ time. [Hint: With the given boundary conditions, the array must have at least one local minimum. Why?]

Additional note from the instructors: Your algorithm to the above should leverage some kind of induction hypothesis to argue it is correct. What is the induction hypothesis?

**Exercises 3.5.f in [Eri19]**. Call a sequence $X[1..n]$ of numbers *convex* $2X[i] < X[i-1] + X[i+1]$ for all $i$. Describe an efficient algorithm to compute the length of the longest convex subsequence of an arbitrary array $A$ of integers.

**Exercise 7.4.b in [Eri19]**. A *feedback edge set* of an undirected graph $G$ is a subset $F$ of the edges such that every cycle in $G$ contains at least one edge in $F$. In other words, removing every edge in $F$ makes the graph acyclic. Describe and analyze a fast algorithm to compute the minimum-weight feedback edge set of a given edge-weighted graph.15

**Exercise 11.8 in [Eri19]** Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominoes, just the right size to cover to squares of the checkerboard. Describe and analyze an algorithm to determine whether one can tile the board with dominoes – each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.

Your input is a boolean array $Deleted[1..n, 1..n]$, where $Deleted[i, j] = true$ iff the square in row $i$ and column $j$ has been deleted. Your output is a single boolean; you do not have to compute the actual placement of dominoes. For example, for the board shown below, your algorithm should return true.

![Figure F.1: Covering a partial checkerboard with dominos. (Figure 11.12 in [Eri19].)]](image.png)

---

15For simplicity you may assume the edge weights are positive. To think about later: what if the edge weights are allowed to be negative?
F.10  Recitation 10

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.\(^6\)

**Exercise 19.1.** Recall the array-backed lists from section 19.3, where we showed how to support \textbf{append} in \(O(1)\) amortized time via the “doubling trick”. As discussed at the end of section 19.3, one may also want to include a \textbf{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 than 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 \(n\) \textbf{append} / \textbf{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 \textbf{remove} so that \textbf{append} and \textbf{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.

   The following is exercise 5 in [Eri13a]. Try to do parts (a) and (b). Part (c) is harder but a good challenge to think about offline.\(^7\)

\(^6\)Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby

\(^7\)I (Kent) believe that for part (c), \(O(1)\) amortized-time \textbf{pop()} is achievable.
5. A multistack consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the $i$th stack $S_i$ can hold up to $3^i$ elements. The user always pushes and pops elements from the smallest stack $S_0$. However, before any element can be pushed onto any full stack $S_i$, we first pop all the elements off $S_i$ and push them onto stack $S_{i+1}$ to make room. (Thus, if $S_{i+1}$ is already full, we first recursively move all its members to $S_{i+2}$.) Similarly, before any element can be popped from any empty stack $S_i$, we first pop $3^i$ elements from $S_{i+1}$ and push them onto $S_i$ to make room. (Thus, if $S_{i+1}$ is already empty, we first recursively fill it by popping elements from $S_{i+2}$.) Moving a single element from one stack to another takes $O(1)$ time.

Here is pseudocode for the multistack operations $M\text{Push}$ and $M\text{Pop}$. The internal stacks are managed with the subroutines $\text{Push}$ and $\text{Pop}$.

- $M\text{Push}(x)$:
  
  ```
  i ← 0
  while $S_i$ is full
  i ← i + 1
  while i > 0
  i ← i - 1
  for j ← 1 to $3^i$
  Push($S_{i+1}$, Pop($S_i$))
  Push($S_0$, x)
  ```

- $M\text{Pop}(x)$:
  
  ```
  i ← 0
  while $S_i$ is empty
  i ← i + 1
  while i > 0
  i ← i - 1
  for j ← 1 to $3^i$
  Pop($S_i$, Pop($S_{i+1}$))
  return Pop($S_0$)
  ```

(a) In the worst case, how long does it take to push one more element onto a multistack containing $n$ elements?

(b) Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where $n$ is the maximum number of elements in the multistack during its lifetime.

(c) Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where $n$ is the maximum number of elements in the multistack during its lifetime.
F.11  Recitation 11

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.\footnote{Solutions are also posted at \url{https://www.overleaf.com/read/gmhtrxhsfcb}y}

Exercise 21.4. For \(k \in \mathbb{N}\), 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?\footnote{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 think about a fair coin, where \(p = 1/2\).} 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 21.6. Recall the selection problem. Given an input \(A[1..n]\) of \(n\) comparable elements, the goal is to find the rank \(k\) element (that is, the \(k\)th largest element) in \(A\). You may assume the elements are distinct for simplicity.

Previously we studied a deterministic, linear time algorithm called median-of-medians. The algorithm was rather non-trivial – it recursively computes an approximate median to be used as a pivot. Here we will study a much simpler algorithm called quick-select that uses randomization instead of divide-and-conquer. The idea is very simple and similar to quick-sort: select a pivot randomly. In practice, this simpler algorithm is faster than median-of-medians.

```
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 \(\ell\) of \(A[i]\).  
   \(\text{// } O(n)\)

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.
The goal of this exercise is to prove that quick-select takes $O(n)$ time in expectation. We ask you to prove this in two different ways which offer two different perspectives.

1. **Approach 1.** Analyze quick-select similarly to quick-sort, based on the sum of indicators $X_{ij}$.

   You may want to do 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 $O(n)$.\(^{20}\)

   Use this to obtain a $O(n)$ expected running time, overall.

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 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 $k$th element or throw out at least $1/4$ of the remaining elements.

   (b) 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 $[\left(\frac{4}{3}\right)^i, \left(\frac{4}{3}\right)^{i+1}]$, is $O(1)$.\(^{21}\)

   (c) Fix an integer $i$, and consider the amount of time spent by quick-select while the number of elements remaining is greater than $\left(\frac{4}{3}\right)^{i-1}$ and at most $\left(\frac{4}{3}\right)^i$. Show that that the expected amount of time is $\leq O\left(\left(\frac{4}{3}\right)^i\right)$

   (d) Finally, use the preceding part to show that the expected running time of quick-select is $O(n)$.

**Exercise 21.5.** Suppose you only have access to a coin that flips heads with a known probability $\hat{p}$, and tails with (remaining) probability $1 - \hat{p}$. Describe and

\(^{20}\)The trickiest is the family of $X_{ij}$ where $i < k < j$. As an intermediate step, you might show that expected sum is $O(k \log(n/k))$. Is this increasing or decreasing in $k$?

\(^{21}\)Hint: Exercise 21.4.
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 number of biased coin tosses you make may in expectation may depend on $p$.)
F.12 Recitation 12

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.\footnote{Solutions are also posted at \url{https://www.overleaf.com/read/gmhtrxhsfcby}}
F.13 Recitation 13

Please try to solve each of the following problems, and we encourage you to discuss with your neighbors. Solutions will be discussed together as a class.²³

²³Solutions are also posted at https://www.overleaf.com/read/gmhtrxhsfcby
Appendix G

Selected solutions

In this chapter we provide solutions or solution sketches for some of the exercises. Some of these descriptions may be a little sketchier then we would expect in a submitted homework solution, especially where we believe the reader could furnish the missing details.

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 \( T(1) = 11 \).

Solution. In the following, I will try to describe what the recursion tree should look like; in reality, I have drawn out the recursion tree on a piece of scratch paper and I encourage you to do the same.

1. We have a ternary tree (three children per node) with height \( \log_3(n) = O(\log n) \). So there is 1 problem at the top, 3 problems at the second level, 9 problems at the third level, and generally \( 3^{i-1} \) problems at the \( i \)th level. The root has size \( n \), the problems at the second level have size \( n/3 \), the problems at the third level have size \( n/9 \), and so forth. Each subproblem contributes work. In particular, at the \( i \)th level, we have \( 3^{i-1} \) subproblems each contributing \( O(n/3^{i-1}) \) work. So the total work at level \( i \) is \( 3^{i-1} \cdot O(n/3^{i-1}) = O(n) \). There are \( O(\log n) \) levels, so we get a total running time of \( O(n \log n) \).

2. We now have a binary tree, so there are \( 2^{i-1} \) problems on the \( i \)th level. The problem sizes at each level are the same as before. When we sum up the work on the \( i \)th level, we now have

\[
2^{i-1} \cdot n/3^{i-1} = (2/3)^{i-1} n.
\]
In particular the work is geometrically decreasing by level. Summing up (cf. appendix C.2), we have

\[ O(\log n) \sum_{i=1}^{\log_3 n} \left( \frac{2}{3} \right)^{i-1} n = O(n). \]

3. This difference now is that we have four children per node, so there are \(4^{i-1}\) subproblems on the \(i\)th level (each of the same size, and contributing the same amount of work, as before). The total amount of work at the \(i\)th level is now (up to constants) \((4/3)^{i-1} n\). In particular the work is geometrically increasing by level. Here we have to be more precise about the height.

\[
\left\lceil \log_3 n \right\rceil \sum_{i=0}^{\left\lceil \log_3 n \right\rceil} (4/3)^{i-1} n \leq (4/3)^{\log_3(4)+\text{(some constant)}} n \\
= O\left(n^{\log_3\left(\frac{4}{3}\right)} \cdot n\right) \\
\overset{(a)}{=} O\left(n^{\log_3(4)}\right).
\]

Here (a) is because \(\log_3(4/3) = \log_3(4) - 1\).

\[
\blacktriangleleft
\]

**Exercise 1.2.** 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 \(A\).

**Solution.** This is a very standard problem given as an example in many textbooks. (So in particular a more comprehensive answer can be found elsewhere.)

The naive approach is just to compare all pairs, in \(O(n^2)\) time. But one can do better with divide-and-conquer. The high level idea is to adapt the merge-sort algorithm; in particular, we will modify the merge part.

We will describe a divide-and-conquer that, given an array \(A[1..n]\) of comparable elements, both (a) sorts \(A\) and (b) counts the number of inversions in \(A\).

In the base case \(n \leq 1\) and to do this. Otherwise let \(n > 1\). Given an array \(A[1..n]\) of elements, consider the two halves \(A[1..[n/2]]\) and \(A[[n/2]+1,..,n]\). Recursively we can count the number of inversions within each half and obtain sorted arrays \(B[1..[n/2]]\) and \(C[[n/2]+1..n]\) of the two halves. Now, when merging \(B\) and \(C\), every time that an element of \(B\) is selected from \(C\), observe this...
element will be placed ahead of all the remaining elements in \( B \), even though it originated after all of these elements in \( A \). That is, we have an inversion for every element remaining in \( B \). So we maintain a count where, every time the merging selects an element from \( C \), we increase the count by the number of elements in \( B \). This counts the total number of inversions across the two halves. Now the total number of inversions comes from adding up the three counts – the number of inversions within the first half, the number of inversions within the second half, and the number of inversions across the halves counted by the merge.

(In a submitted homework, we would expect some pseudocode to ensure there is no ambiguity about the algorithm being described.)

The running time follows from essentially the calculations as merge sort (which you should repeat). To prove the algorithm is correct, one needs to show, by induction on the number of elements, that each recursive call does exactly what we claimed above:

“...that, given an array \( A[1..n] \) of comparable elements, both (a) sorts \( A \) and (b) counts the number of inversions in \( A \).”

(The description above is probably pretty close to a full proof. One should probably be more explicit about checking the base case, etc. In general it is better to be too explicit and unambiguous than possibly being unambiguous, particular since you are not really given the benefit of doubt when we are trying to assess and provide feedback on whether you have understood all the new material.) ■

Exercise 1.3. 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 \( i \)th 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 \((x_1, \ldots, x_n)\).

Solution. You can prove this by following the exact same structure as the proof for the sorting lower bound. The two key observations are:

1. With only \( k \) queries and 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. ■

**Exercise 4.1.** Let \( A[1..n] \) be an array of \( n \) numbers. Consider the problem of computing the length of the longest subsequence of \( A \) that is strictly increasing. Here a sequence of numbers \( x_1, \ldots, x_k \) is strictly increasing if \( x_i < x_{i+1} \) for \( i = 1, \ldots, k-1 \). 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.

**Solution.** We will use dynamic programming and describe our solution following the steps outlined in section 3.1.3.

1. **Recursive spec / induction hypothesis.** For \( i \in \{1, \ldots, n\} \), we define
   \[
   \text{LIS}(i) = \text{the length of the longest increasing subsequence of } A[i..n] \\
   \text{where the subsequence starts with } A[i].
   \]
   (Here \( \text{LIS} \) stands for longest increasing subsequence.)

2. **Recursive implementation.**
   \[
   \text{LIS}(i)
   \]
   1. If \( A[i] \geq A[j] \) for all \( j > i \) then return 1.
   2. Otherwise return the maximum of
      \[1 + \text{LIS}(j)\]
      over all indices \( j > i \) with \( A[j] > A[i] \).

3. **Solving the original problem.** The maximum of \( \text{LIS}(i) \) 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)).

5. **Proof.** Consider a subproblem \( \text{LIS}(i) \), where we want to longest increasing subsequence of \( A[i..n] \) where the subsequence starts with \( A[i] \). If \( A[i] \) is the largest remaining number then this subsequence must consist of only \( i \). Otherwise, there is at least one other number remaining bigger than \( A[i] \). Suppose that in the largest increasing subsequence starting with \( A[i] \), the next number in the subsequence \( A[j] \). Then the remaining part after \( A[i] \), is an increasing

\[\text{1} \text{A solution is given on page 497}\]
subsequence starting from \( A[j] \), and by induction the length of this subsequence is \( \text{LIS}(j) \). The maximization in (2) over all choices of \( j \) ensures the value we return is indeed \( 1 + \text{LIS}(j) \).

**Remark G.1.** We emphasize that the recursive spec essentially solved the entire problem.

**Remark G.2.** It is well known that there are faster algorithms but that is besides the point.

**Exercise 4.2.** Recall that a **palindrome** is a string that reads the same forwards and backwards. For example,

\[
\text{mom, dad, racecar, and gohangasalamitamasagnahog}
\]

are all palindromes. Consider the problem where you are given a string \( A[1..n] \) as input, and the goal is to find the (length of the) longest palindrome that is a subsequence of \( A \). 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 to exercise 4.2.** We give a polynomial time algorithm using dynamic programming.

1. **Recursive spec / induction hypothesis.** We define
   
   \[ \text{LPS}(i, j) = \text{the length of the longest palindrome that is a subsequence of } A[i..j]. \]

2. **Recursive implementation.**

   \[
   \text{LPS}(i, j)
   \]

   1. If \( i > \) then return 0.
   2. If \( i = j \) then return 1.
   
   /* Otherwise \( i < j \). */

   3. If \( A[i] = A[j] \) then return the maximum of \( 2 + \text{LPS}(i + 1, j - i), \text{LPS}(i, j - 1), \text{and LPS}(i + 1, j) \).
   4. Otherwise return the maximum of \( \text{LPS}(i + 1, j) \) and \( \text{LPS}(i, j - 1) \).

3. **Solving the original problem.** The original problem is solved by \( \text{LPS}(1, n) \).
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.

5. Proof. We prove by induction on \( j-i \) that the pseudocode above implements the recursive spec of \( \text{LPS}(i,j) \).

In the base case where \( i > j \), \( A[i..j] \) is an empty array so the solution is 0.
In the base case where \( i = j \), \( A[i..j] \) consists of a single letter, which gives the maximum length palindrome.

In the general case suppose \( j > i \). We have two cases depending on whether \( A[i] = A[j] \) or not. Let \( S \) refer to the (true) longest palindrome subsequence of \( A[i..j] \). We claim that \( \text{LPS}(i,j) \) returns the length of \( S \).

Suppose \( A[i] = A[j] \). Then \( S \) either includes both \( A[i] \) and \( A[j] \) or excludes at least one of them.

1. If it includes \( A[i] \) and \( A[j] \), then the remaining part of \( S \) (excluding \( A[i] \) and \( A[j] \)) is a palindrome subsequence of \( A[i+1..j-1] \). By induction, \( \text{LPS}(i+1,j-1) \) computes the length of the longest palindrome subsequence of this smaller subarray, so \( 2 + \text{LPS}(i+1,j-1) \) returns (at least) the length of \( S \).

2. If \( S \) excludes \( A[i] \), then it is a palindrome subsequence of \( A[i+1..j] \). By induction, \( \text{LPS}(i+1,j) \) computes the length of the longest palindrome subsequence of \( A[i+1..j] \), so \( \text{LPS}(i+1,j) \) is (at least) the length of \( S \).

3. If \( S \) excludes \( A[j] \), then it is a palindrome subsequence of \( A[i..j-1] \). By induction, \( \text{LPS}(i,j-1) \) computes the length of the longest palindrome subsequence of \( A[i..j-1] \), so \( \text{LPS}(i+1,j) \) is (at least) the length of \( S \).

Now, \( \text{LPS}(i,j) \) returns the maximum of the three values above, one of which will give us the length of \( S \).


1. If \( S \) excludes \( A[i] \), then it is a palindrome subsequence of \( A[i+1..j] \). By induction, \( \text{LPS}(i+1,j) \) computes the length of the longest palindrome subsequence of \( A[i+1..j] \), so \( \text{LPS}(i+1,j) \) is (at least) the length of \( S \).

2. If \( S \) excludes \( A[j] \), then it is a palindrome subsequence of \( A[i..j-1] \). By induction, \( \text{LPS}(i,j-1) \) computes the length of the longest palindrome subsequence of \( A[i..j-1] \), so \( \text{LPS}(i+1,j) \) is (at least) the length of \( S \).

Now, \( \text{LPS}(i,j) \) returns the maximum of the two values above, one of which will give us the length of \( S \).
Exercise 4.7. Let $A[1..m]$ be a sequence of integers. Recalling that a subsequence is (strictly) increasing if each successive integer is strictly greater than the previous.

Here we consider two variants of the longest increasing subsequence problem. In the first problem, we want to find the longest increasing subsequence where the sum of integers in the subsequence is even. In the second problem, we want to find the longest increasing subsequence where the sum of integers in the subsequence is odd.

For both of these 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.

Solution. We will use dynamic programming and describe our solution following the steps outlined in section 3.1.3.

1. Recursive specs. For $i \in \{1, \ldots, m\}$, we define

   (a) $\text{EvenLIS}(i)$ 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).

   (b) $\text{OddLIS}(i)$ 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 implementations.

   $\text{EvenLIS}(i)$

   1. If $A[i]$ is even:

   A. Return the maximum of 1 or the maximum of $1 + \text{EvenLIS}(j)$ 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 $1 + \text{OddLIS}(j)$ over all $j > i$ with $A[j] > A[i]$.
1. If $A[i]$ is odd:
   A. Return the maximum of 1 or the maximum of $1 + \text{EvenLIS}(j)$ 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 $1 + \text{OddLIS}(j)$ over all $j > i$ with $A[j] > A[i]$.

3. *Mention “dynamic programming”* We cache the solutions to each subproblem (EvenLIS([1..m]) and OddLIS([1..m])) above in a $O(m)$-size table.

4. *Running time.* We have $O(m)$ subproblems, each of which takes $O(m)$ time, giving $O(m^2)$ time overall.

5. *Space.* There are $O(m)$ subproblem solutions that are stored.

6. *Relating the subproblems to the original problem.* The length of the longest increasing subsequence with an even sum is obtained by the maximum of EvenLIS($i$) over all $i \in [m]$. The length of the longest increasing subsequence with an odd sum is obtained by the maximum of OddLIS($i$) over all $i \in [m]$.

7. *Proof.* Omitted out of instructor laziness. (It would be similar to LIS on page 497, except with appropriate modifications to account for the parity.)

**Remark G.3.** We emphasize that the recursive specs made the implementation very straightforward.

**Exercise 6.4.** Let $G = (V, E)$ be 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. We say that a vertex $s$ can **patriotically reach** a vertex $t$ if there is an American walk from $s$ to $t$.

Consider the problem of deciding whether a vertex $s$ can patriotically reach $t$. 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.\(^2\)

\(^2\)Hint: don’t change the algorithm; change the graph!
Solution.

American-walk\((G = (V, E), s, t)\)

1. Build the following graph \(G' = (V', E')\).
   
   A. For each \(v \in V\), we have three copies \(v_{\text{red}}, v_{\text{white}}, v_{\text{blue}} \in V'\).
   
   B. For each red edge \(e = (u, v) \in E\), we have the edge \((u_{\text{blue}}, v_{\text{red}}) \in E'\).
   
   C. For each white edge \(e = (u, v) \in E\), we have the edge \((u_{\text{red}}, v_{\text{white}}) \in E'\).
   
   D. For each blue edge \(e = (u, v) \in E\), we have the edge \((u_{\text{white}}, v_{\text{blue}}) \in E'\).
   
   E. We also have additional vertices \(s^*, t^* \in V'\) and the edges \((s^*, s_{\text{red}})^*, (s^*, s_{\text{white}})^*, (s^*, s_{\text{blue}})^*, (t_{\text{red}}, t^*)^*, (t_{\text{white}}, t^*)^*, (t_{\text{blue}}, t^*)^* \in E'\).

2. Run DFS from \(s^*\) in \(G'\) to see if \(s^*\) can reach \(t^*\), returning this result.

There is a 1-1 bijection between American \((s, t)\)-walks in \(G\) and \((s^*, t^*)\)-walks in \(G'\). To see this, for a colored edge \(e \in E\), let \(e'\) denote its (unique) copy in \(E'\). Given an American \((s, t)\)-walk with edges \(e_1, e_2, \ldots, e_k\), the sequence of edges \(e'_1, \ldots, e'_k\) forms a walk in \(E'\). Adding the edges from \(s^*\) to the initial point of \(e'_1\) and from the end point of \(e'_k\) to \(t^*\) gives an \((s^*, t^*)\)-walk in \(G'\). Conversely, given an \((s^*, t^*)\)-walk in \(G'\), we can omit the first and last edges (from \(s^*\) and to \(t^*\)) and map the remaining edges back to \(G\) and obtain an American walk. The transformations are inverse to one another.

We have already established that DFS\((s^*)\) visits \(t^*\) iff \(s^*\) can reach \(t^*\); by the bijection underlying our construction, the latter holds iff there is an American walk from \(s\) to \(t\) in \(G\).

The algorithm takes \(O(m + n)\) time, because \(G'\) has \(m' = O(m)\) edges, \(n' = O(n)\) edges, and the DFS takes \(O(m' + n') = O(m + n)\) time.

**Exercise 5.4.** Recall the dominating set problem from section 5.3. 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.\(^3\)

2. The minimum weight dominating set problem in trees.

**Solution.** We first make the simplifying assumption that there are no nested pairs of intervals. We sort the intervals by left endpoint and let \( I_i \) be the \( i \)th interval by increasing left endpoint.

1. **Recursive spec.**

\[ \text{MWOS}(i) : \text{The minimum weight of any set of intervals that overlaps with all of } I_i, \ldots, I_n. \]

2. **Recursive implementation.**

\[ \text{MWOS}(i) \]

1. If \( i > n \) then return 0.

2. Return the minimum, over all intervals \( I_j \) overlapping with \( I_i \), the sum of \( w(I_j) + \text{MWOS}(k + 1) \), where \( k \) is the index of the last interval covered by \( I_j \) (and is at least \( i \)).

\[ /* \text{It is possible that } j < i! \text{ This is subtle, and may lead to chaos.} */ \]

3. We return \( \text{MWOS}(1) \).

4. **Running time:** We apply dynamic programming. There are \( O(n) \) problems (where \( n \) is the number of intervals) and a naive implementation would have a two loops: one over the choices of \( I_j \), and to obtain the index \( k \). However, we can precompute, for every \( I_j \), the index of the last interval \( I_k \) that overlaps with \( I_j \). For each \( I_j \), this takes \( O(\log n) \) time with binary search, and thus \( O(n \log n) \) time to do this precomputation for all \( I_j \). With this choice of \( k \) precomputed, each subproblem takes \( O(n) \) time, and we obtain a \( O(n^2) \) running time overall.

\(^3\)Of course, anyone who has already solved the general case automatically solves the special case where no two intervals are nested.
5. Proof by induction. Consider $\text{MWOS}(i)$. If $i > n$ then of course there are no intervals left to cover, and the solution is of course 0. (Note that all weights are positive.)

Suppose $i \leq n$. Consider each interval $I_j$ considered in step (2). We know that $I_j$ covers both $I_i$ and $I_k$ (where $k \geq i$); the non-nested assumption implies that $I_j$ covers all the intervals between $I_i$ and $I_k$. Meanwhile, by induction on the number of intervals remaining, $\text{MWOS}(k + 1)$ is the weight of the MWOS of $I_{k+1}, \ldots, I_n$. Therefore, for each $j$, $w(I_j) + \text{MWOS}(k + 1)$ is the weight of some set of intervals overlapping all of $I_i, \ldots, I_n$.

Now consider the true MWOS of $I_i, \ldots, I_n$. The MWOS contains some interval $I_j$ overlapping $I_i$; we note that it is possible to have $j < i$. Let $k$ be the index of the last interval overlapping $I_j$; the remaining intervals in the MWOS overlap with all of $I_{k+1}, \ldots, I_n$. By induction on the number of intervals that need to be overlapped, $\text{MWOS}(k + 1)$ returns the weight of the MWOS of $I_{k+1}, \ldots, I_n$; together, $w(I_j) + \text{MWOS}(k + 1)$ gives the weight of the MWOS of $I_i, \ldots, I_n$. This is among the options considered in step (2), so $\text{MWOS}(i)$ indeed gives the MWOS of $I_i, \ldots, I_n$.

Remark G.4. From what I can tell, the following recursive spec (which is the most natural one) may lead to some subtle bugs:

$$\text{MWDS}(i) = \text{the minimum weight dominating set out of } I_i, \ldots, I_n$$

because it does not allow us, for fixed $I_i$, to choose an overlapping $I_j$ where $j < i$. (Perhaps one can get around this anyway.)

To remove the assumption of having no nested intervals, we do the following. Let $\mathcal{J}_1$ denote the original set of intervals, and let $\mathcal{J}_2$ be obtained by repeatedly removing any interval $I$ that contains another remaining interval $J$. Observe that any set of intervals overlapping with all the intervals in $\mathcal{J}_2$ is also overlapping all the intervals in $\mathcal{J}_1$, because we have only removed intervals that contain some interval in $\mathcal{J}_2$. Meanwhile $\mathcal{J}_2$ has no nested pairs.

The goal now is to compute the minimum weight subset of $\mathcal{J}_1$ that overlaps with every interval in $\mathcal{J}_2$. Now one can basically apply the same approach as above where we assumed that none of the intervals were nested (with appropriate modifications).

Solution to part 2. We root the tree arbitrarily and henceforth refer to subtrees, children, etc., accordingly.

1. Recursive spec. For each $v \in V$, we declare

$$\text{MWDS}(v) = \text{the minimum weight over all dominating sets of the subtree rooted at } v.$$
MWDS-except-root(v) = the minimum weight of any subsets of vertices of the subtree rooted at v among those that dominate all the vertices in the subtree besides v. (Such a set may also dominate v; the point is that this is optional.)

MWDS-with-root(v) = the minimum weight dominating set of the subtree rooted at v among those that include v.

2. Recursive implementation.

MWDS(v)

1. Return the minimum of:
   
   /* Include v, which also means that all the children are already dominated. */
   A. \( w(v) + \sum_{\text{child } c \text{ of } v} \text{MWDS-except-root}(c) \).
   
   /* Don’t include v, and instead choose a child to dominate v. */
   B. The minimum, over all children c of v, of the sum of:
      1. MWDS-with-root(c)
      2. The sum of MWDS(d) over all children d other than c.
   
   /* For sake of efficiency, when calculating the minimum as described above, we first compute the sum \( X \) of MWDS(d) over all children d other than c. Then, for each child c, we take \( X \), subtract MWDS(c), and add MWDS-with-root(c) to get the score for that choice of child. In this manner, the total time to evaluate all choices of child c is proportional to the number of children. */

MWDS-with-root(v)

1. Return \( w(v) + \sum_{\text{child } c \text{ of } v} \text{MWDS-except-root}(c) \)

MWDS-except-root(v)

1. Return the minimum of:
   
   /* Include v, which also means that all the children are already dominated. */
   A. \( w(v) + \sum_{\text{child } c \text{ of } v} \text{MWDS-except-root}(c) \).
   
   /* Don’t include v. */
   B. \( \sum_{\text{child } c \text{ of } v} \text{MWDS}(c) \).
3. To extract the original solution, we return MWDS($r$) where $r$ is the root.

4. “Dynamic programming”.

5. There are $O(n)$ subproblems. If one is careful (reusing sums in step (1.B) of MWDS, as described above), one can make sure that each subproblem takes time proportional to the number of children, and get $O(n)$ time overall.

**Exercise 7.7.** 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 in $G$ for which the labels of the vertices are strictly increasing.

1. (7 points) 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. (3 points) 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.

**Solution to part 1.** 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

   \[ \text{LIP}(v) = \text{the length of the longest path starting from } v \text{ with strictly increasing labels}. \]

2. **Recursive implementation.**

---

\(^*\text{Recall that a path is a walk that does not repeat vertices.}\)
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 

$$1 + \text{LIP}(w)$$

over all edges $(v,w)$ with $\ell(w) > \ell(v)$.

Solution to part 2. 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.

Exercise 7.10. 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. You may assume for simplicity that $G$ has no sinks.

1. Given an integer $k \leq n$, the goal is to compute a walk of length $k$ maximizing the sum of rewards along that walk.\(^5\) (If you repeat an edge, you get the same reward each time you repeat the edge.) 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. 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 $k$ maximizing the discounted total reward of the walk. 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.

\(^5\)There is always a $k$-edge walk under the assumption that $G$ has no sinks.
Solution to part 1. This problem is very similar to the longest path problem, except the starting point $s$ is fixed, and the edges are weighted. Given the similarity to both longest path and the discounted reward version after (imagine $\alpha = 1$), I will be brief for this part and skip the proof.

1. Recursive spec / induction hypothesis. For $v \in V$ and $i = 0, \ldots, k$, we define

$$MRW(v, i) = \text{the maximum reward of any length } i \text{ walk starting from } v.$$ 

2. Recursive implementation.

$$MRW(v, i)$$

1. If $i = 0$ return 0.

2. Otherwise return the maximum, over all edge $(v, w)$ leaving $v$, of the sum

$$r(v, w) + MRW(w, i - 1).$$ 

3. Solving the original problem. The original problem is solved by $MRW(s, k)$.

4. Running time with caching / dynamic programming. We apply dynamic programming. We have $O(nk)$ subproblems, defined by $v \in V$ and $i \in \{0, \ldots, k\}$. For fixed $(v, i)$, the corresponding subproblem takes time proportional to the out-degree of $v$. For fixed $i$, the total running time over all $v \in V$, is $O(m)$. Summing over $i$, the total running time over all $v$ and $i$ is $O(km)$.

Solution to part 2. This solution builds on the ideas from part 1.

1. Recursive spec / induction hypothesis. For $v \in V$ and $i = 0, \ldots, k$, we define

$$MRDW(v, i) = \text{the maximum discounted reward of any length } i \text{ walk starting from } v.$$ 

2. Recursive implementation.

$$MRDW(v, i)$$

1. If $i = 0$ return 0.

2. Otherwise return the maximum, over all edges $(v, w)$ leaving $v$, of the sum

$$r(v, w) + \alpha MRDW(w, i - 1).$$
3. Solving the original problem. The original problem is solved by MRDW(s, k).

4. Running time with caching / dynamic programming. We apply dynamic programming. We have $O(nk)$ subproblems, defined by $v \in V$ and $i \in \{0, \ldots, k\}$. For fixed $(v, i)$, the corresponding subproblem takes time proportional to the out-degree of $v$. For fixed $i$, the total running time over all $v \in V$, is $O(m)$. Summing over $i$, the total running time over all $v$ and $i$ is $O(km)$.

5. Proof. We prove correctness of MRDW($v, i$) by induction on $i$. For $i = 0$ there is only the empty walk where the reward is 0.

Fix $v$ and $i > 0$ and consider MRDW($v, i$). Consider the true MRDW from $v$ with $i$ edges, which we denote $v, v_2, v_3, \ldots, v_i$. Observe that the total discounted reward is

$$r(v, v_2) + \alpha r(v_2, v_3) + \alpha^2 r(v_3, v_4) + \cdots + \alpha^{k-1} r(v_{i-1}, v_i)$$

$$= r(v, v_2) + \alpha \left( r(v_2, v_3) + \alpha r(v_3, v_4) + \cdots + \alpha^{k-2} r(v_{i-1}, v_i) \right).$$

The parenthesized sum on the RHS is the discounted reward of the $i - 1$ edge walk $v_2, v_3, v_4, \ldots, v_{i-1}, v_i$. By induction on $i$, MRDW($v_2, i - 1$) returns (at least) the value of this walk, and $r(v, w) + \alpha$MRDW($v_2, i - 1$) is among the options considered by the algorithm.

Remark G.5. We are thankful to the Piazza posts regarding the existence of $k$-edge walks, sinks, etc., which improved this exercise. We plan to take the most favorable interpretation when grading.

Exercise 8.2. 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 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.

**Solution to part 1.** The problem is NP-Hard and we will argue that it can solve the Hamiltonian path problem.

To prove this, suppose we had a black box to solve the problem described above. Given an instance of Hamiltonian path defined by a graph $G$, we define an instance of the describe problem where we give every edge $e$ length $\ell(e) = 1$ and set $x_1, \ldots, x_k$ to be the set $V$ of all vertices. We ask the black box for the length of the shortest walk visiting $V$, and return true iff the black box outputs length $n - 1$.

Now suppose $G$ had a Hamiltonian path. Then this Hamiltonian path is a walk of length $n - 1$ visiting all of $V$ in our weighted graph, so we will return true. If $G$ does not have a Hamiltonian path, then any walk in $G$ visiting all of $V$ requires at least $n$ edges. Therefore, any such walk has length at least $n$ in our uniformly weighted version of $G$, and the algorithm returns false.

**Remark G.6.** One can also reduce from Traveling Salesman and Hamiltonian cycle though there should be something to address the difference between circuits/cycles and walks/paths.

**Remark G.7.** Minor getcha’s include remembering to give the graph edge weights since that is what the black box expects, and checking both directions. (We plan to be gentle on these points when grading since this is still one of our first times to do something like this, but it is good to be aware of these for the exam.)

**Solution to part 2.** The high-level idea is to reduce the problem to a sequence of shortest path problems for each consecutive pair. Here would be such an algorithm.

$\text{ShortestWalk}(G = (V,E), \ell : E \to \mathbb{R}_{\geq 0}, x_1, \ldots, x_k)$

1. For each vertex $x$ appearing (at least once) in the list $x_1, \ldots, x_k$, compute the distance from $x$ to every other vertex with Dijkstra’s algorithm.

   /* Above, one might be careful to avoid rerunning Dijkstra from the same x particularly when k is much larger than n. */

2. Return the sum $d(x_1, x_2) + d(x_2, x_3) + \cdots + d(x_{k-1}, x_k)$ where $d(x_i, x_{i+1})$ denotes the distance from $x_i$ to $x_{i+1}$ and was calculated in step 1.

The running time is $O((m + n \log n) \min\{n, k\})$, coming from running Dijkstra’s algorithm from each distinct vertex in the list $x_1, \ldots, x_n$.
The value \( d(x_1, x_2) + d(x_2, x_3) + \cdots + d(x_{k-1}, x_k) \) is the length of the walk obtained by concatenating the length of the shortest \((x_i, x_{i+1})\) walk/path for each \( i \). To show that it is optimal, we observe that any walk visiting \( x_1, \ldots, x_k \) in order must have length at least \( d(x_1, x_2) + d(x_2, x_3) + \cdots + d(x_{k-1}, x_k) \), since for each \( i \) the “leg” from \( x_i \) to \( x_{i+1} \), as an \((x_i, x_{i+1})\)-walk, has length at least \( d(x_i, x_{i+1}) \).

Remark G.8. It is a very minor point that when \( k \) is larger than \( n \), one can save some time by avoiding redundant shortest path computations.

Remark G.9. We are grateful to the Piazza posts whose questions improved this exercise.

Exercise 4.10. Recall that previously we computed the length of the longest increasing subsequence of an input sequence of numbers. Here, instead of increasing numbers, we are interested in “wavy” sequences of numbers that oscillate up and down. More specifically, we say that a sequence of numbers \( y_1, y_2, \ldots, y_k \) is:

1. **Up-down wavy** if \( y_i < y_{i+1} \) for all odd \( i \), and \( y_i > y_{i+1} \) for all even \( i \).

2. **Down-up wavy** if \( y_i > y_{i+1} \) for all odd \( i \), and \( y_i < y_{i+1} \) for all even \( i \).

3. **Wavy** if it is either up-down wavy or down-up wavy.

(“Up-down wavy” sequences go up, then down, at the beginning. “Down-up wavy” sequences go down, then up, at the beginning.) For example, the sequence 1, 9, 2, 8, 3, 7, 4, 6 is an up-down wavy sequence. The reverse sequence, 6, 4, 7, 3, 8, 2, 9, 1 is a down-up wavy sequence.

Given an array of integers \( X[1..n] \), the goal is to compute the length of the longest wavy subsequence of \( X \).\(^6\) 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.

1. **Recursive spec:** We \( i = 1, \ldots, n \), we define:

   \[ \text{DUW}(i) = \text{the maximum length of any down-up wavy subsequence of } X[i..n], \text{ starting with } X[i]. \]

   \[ \text{UDW}(i) = \text{the maximum length of any up-down wavy subsequence of } X[i..n], \text{ starting with } X[i]. \]

2. **Recursive implementation**

\(^6\)As usual, the subsequence does not have to be a consecutive subsequence of \( X \). It could be like \( X[1], X[4], X[9], \) and so forth, skipping over some entries.
UDW(i)
1. Return the maximum of
   A. 1
   B. The maximum of 1 + DUW(j) over all j > i with
      \(X[j] > X[i]\).

DUW(i)
1. Return the maximum of
   A. 1
   B. The maximum of 1 + UDW(j) over all j > i with
      \(X[j] < X[i]\).

3. The solution to the original problem is given by the maximum of UDW(i) or DUW(i) over \(i = 1, \ldots, n\).

4. With dynamic programming, the running time is \(O(n^2)\), where we have \(n\) subproblems each taking \(O(n)\) time.

Remark G.10. This was an “identify subproblems” type of problem, where writing out the recursive specs / induction hypotheses is very helpful. The motivation behind selecting it was that it closely resembles the even-odd longest increasing subsequences problems from the homework. It is essentially problem 5.d in [Eri19], except we also tried to supply additional structure by breaking it down explicitly into the “up-down” and “down-up” variants that can play off of each other in the implementation.

Remark G.11. There are other ways to devise a dynamic programming solution, just as there are different ways to go about the longest increasing subsequence problem. Also, rather than two functions, one can define one function for “wavy” subsequences and using an additional parameter to keep track of whether you are currently going up or down, so to speak.

Exercise 8.6. Let \(G = (V, E)\) be a directed graph with \(m\) edges, \(n\) vertices, positive edge lengths \(\ell : E \to \mathbb{R}_{>0}\). Given \(s, t \in V\), the goal is to compute the length of the shortest walk from \(s\) to \(t\) with an even number of edges (which may be \(+\infty\) if there is no way to get from \(s\) to \(t\) with an even number of 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.
Solution. We reduce the problem to shortest paths in an appropriate auxiliary graph.

\[
\text{shortest-even-walk}(G = (V, E), s, t)
\]

1. We build the weighted directed graph \(G' = (V', E')\) where:
   
   1. For each vertex \(v\), we have two copies \(v_{\text{even}}\) and \(v_{\text{odd}}\) in \(V'\).
   
   2. 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 \(E'\), each with the same edge length as 
      \((u, v)\).

2. We run Dijkstra's algorithm in \(G'\) from \(s_{\text{even}}\) to find the length of
   the shortest walk/path from \(s_{\text{even}}\) to \(t_{\text{even}}\) in \(G'\), and return this
   value.

There is a length-preserving bijection between even-edge walks between \(s\) and \(t\) and walks between \(s_{\text{even}}\) and \(t_{\text{even}}\). Consequently the minimum length walk from \(s_{\text{even}}\) to \(t_{\text{even}}\), computed by Dijkstra's algorithm, gives the minimum length
\((s, t)\)-walk with an even number of edges in \(G\).

Building \(G'\) takes \(O(m + n)\), and running Dijkstra's algorithm in \(G'\) takes
\(O(m + n \log n)\) time.

Remark G.12. This was a “change the graph” type of problem, analogous (e.g.)
to the American walk problem. Here the copies of vertices “remember” if we
have taken an even or odd number of edges so far.

Exercise 5.9. 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 \(\{1, \ldots, k\}\)) 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 \(i = 1, \ldots, k\), 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 for 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.

\(^7\)Technically this would be the “infimum” of all assignment costs of (valid) vertex \(k\)-colorings.
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.

**Part 1.** The problem is NP-Hard.

Suppose we had a black box to solve minimum cost graph coloring. Given an instance of a graph (i.e., vertex) coloring problem, defined by $G = (V, E)$ and $k \in \mathbb{N}$, we create an instance of minimum cost graph coloring with $G$, $k$, and uniform costs $c(i, v) = 1$ for all $v \in V$ and $i = 1, \ldots, k$. We submit the instance to the black box for minimum cost graph coloring. We return true if the minimum cost is finite. We return false if the minimum cost is $+\infty$.

If $G$ is $k$-colorable, then that coloring would have finite cost, and we would return true.

If $G$ is not $k$-colorable, then the infimum cost over all colorings by definition is $+\infty$, and so we would return false.

**Part 2.** We root $T$ arbitrarily and henceforth refer to the root, children, subtrees, etc., accordingly.

1. **Recursive spec / induction hypothesis.** For vertex $v \in V$ and color $i \in \{1, \ldots, k\}$, we define

   
   
   \[
   \text{MCGC}(v, i) = \text{the minimum cost of any graph } k\text{-coloring of the subtree rooted at } v, \text{ where } v \text{ is not assigned color } i.
   \]

2. **Recursive implementation.**

   \[
   \text{MCGC}(v, i)
   \]

   1. Return the minimum, over all $j = 1, \ldots, i - 1$ and $j = i + 1, \ldots, k$ the sum

   
   \[
   c(v, j) + \sum_{\text{children } d \text{ of } v} \text{MCGC}(d, j).
   \]

   3. **Solving the original problem.** The original problem is solved by the minimum, over all $i = 1, \ldots, k$, of $\text{MCGC}(r, i)$, where $r$ is the root of the tree.

4. **Running time with caching / dynamic programming.** We apply dynamic programming to the recursive algorithm. There is a subproblem for every vertex $v$ and each color $i$, and the running time for the subproblem at $v$ is proportional to the number of children of $v$. Summing over all $v$ and $i$ counts each vertex as a child $k$ times, and we obtain a $O(nk)$ running time.
Remark G.13. Part 1 was a “change the input” type of problem. (Here the change
is small, adding uniform costs to an “unweighted” instance.) Part 2 was an
“identify subproblems” type of problem, where the subproblems correspond to
subtrees. Overall this problem combines two problems we discussed: coloring
and partial $k$-coloring (from PSO), and maximum weight independent set (from
lecture).

There are other ways to design a dynamic programming solution to part 2.

**Exercise 11.1.** 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. Prove your algorithm is correct by induction, where the recursive spec is
   your induction hypothesis.
4. Analyze the running time of your algorithm.

**Solution to exercise 11.1.** 1. Recursive spec.

   \[
   \text{majority}(A[1..n]) = \text{the set of all majority elements in } A.
   \]

2. Recursive algorithm

   \[
   \text{majority}(A[1..n])
   \]
   1. If $n \leq 2$ then return all the elements in $A$ (removing duplicates).
      // $O(1)$.
   2. Let $m = \lceil n/2 \rceil$.
   3. Let $X = \text{majority}(A[1..m])$ and $Y = \text{majority}(A[m+1..n])$.
      // $O(2T(n/2))$.
      /* $|X|, |Y| \leq 2$. */

---

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

515
4. \( Z \leftarrow \text{empty list} \)

5. For each distinct element \( e \in X \cup Y \)
   \[
   \begin{align*}
   &\text{A. Count the frequency of } e \text{ among } A. & \quad \text{// } O(n).
   &\text{B. If the frequency is } \geq \frac{n}{2}, \text{ then add } e \text{ to } Z.
   \end{align*}
   \]

6. Return \( Z \).

3. Proof by induction. We prove the algorithm implements the recursive spec by induction on \( n \). In the base cases where \( n \leq 2 \), all elements appearing in \( A \) are majority elements; indeed, these are the elements that are returned.

Suppose \( n > 2 \). Clearly \( \text{majority}(A[1..n]) \) only returns majority elements, since it checks that the every element returned is at least \( n/2 \). It remains to show that \( \text{majority}(A[1..n]) \) returns all the majority elements.

Let \( m = \lceil n/2 \rceil \); note that \( 1 < m < n \) for \( n \geq 2 \). We claim that any majority element \( e \) in \( A[1..n] \) must be a majority element in either \( A[1..m] \) or \( A[m+1..n] \). Indeed, if an element \( e \) is not a majority element in either \( A[1..m] \) or \( A[m+1..n] \), then it has frequency less than \( m/2 \) and \((n − m)/2\) in the two halves, respectively, hence frequency less than
\[
\frac{n}{2} = \frac{m}{2} + \frac{n − m}{2}
\]
overall.

Thus, any majority element in \( A[1..n] \) is a majority element in either \( A[1..m] \) or \( A[m+1..n] \). By induction on \( n \), \( \text{majority}(A[1..n]) \) returns all the majority elements in \( A[1..m] \) and \( A[m+1..n] \). This implies that every majority element in \( A \) has its frequency counted, and is included in the output.

4. Running time. The running time is modeled recursive by \( T(n) = 2T(n/2) + O(n) \); here we note that we only spend \( O(n) \) time recomputing frequencies for at most 4 elements, since the recursive calls can only return up to two majority elements of their respective subproblems. This is the same recurrence as merge sort and is solved by \( T(n) = O(n \log n) \).

Exercise 11.5. 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 \( k \)th largest element among the combined elements of \( A \) and \( B \). For simplicity you make assume that all the elements are distinct.\(^9\)

1. Define a recursive spec for a recursive algorithm to solve this problem.

\(^9\)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.
2. Show that if either \( m \) or \( n \) is \( \leq 5 \) then we can find the \( k \)th 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.

**Solution to exercise 11.5. Recursive spec.**

\[
\text{search-2}(A[1..m], B[1..n], k) = \text{the } k\text{th smallest element out of } A \text{ and } B \text{ combined.}
\]

**Base case.**

If \( m \) and \( n \) are both \( \leq 5 \), then we can merge \( A \) and \( B \) and find the \( k \)th element directly in \( O(1) \) time.

If \( k \leq 5 \) then we can apply the strategy above to \( A[1..k] \) and \( B[1..k] \).

If \( m \leq 5 \) and \( n, k \geq 5 \), then we are looking for the \((k - m)\)th element in \( A[1..m] \) and \( B[k - m..\min(n, k)] \), which reduces to the case where \( m, n \leq 5 \)th.

Similarly if \( n \leq 5 \) and \( m, k \geq 5 \).

**Implementation.**

\[
\text{search-2}(A[1..m], B[1..n], k)
\]

1. If \( m \) or \( n \) is smaller than (say) 5 then find the \( k \)th element directly and return it. 
   
   \[// O(1) \]

   /* In the interesting case, \( m \) and \( n \) are both at least 5. */

2. Let \( m' = \lceil \frac{m}{2} \rceil \) and let \( n' = \lceil \frac{n}{2} \rceil \).

3. If \( A[m'] > B[n'] \):
   
   /* \( A[m'] \) has rank at least \((m' + n')\), and \( B[n'] \) has rank at most \((n' + m' - 1)\). */

   A. If \( k \leq m' + n' \):
      
      /* Then \( A[m' + 1..m] \) are all bigger than the target element. */
      
      1. Return \( \text{search-2}(A[1..m'], B[1..n], k) \).

   B. Otherwise \( k > m' + n' \), and elements \( B[1..n'] \) are all less than the desired element. /*

   1. Return \( \text{search-2}(A[1..m], B[n' + 1..n], k - n') \).
/* Otherwise \( A[m'] < B[n'] \), \( A[m'] \) has rank at most \( (m' + n' - 1) \), and 
\( B[n'] \) has rank at least \( (m' + n') \). */

4. If \( k \leq m' + n' \):
   */ Then all the elements \( B[n' + 1..n] \) are bigger than the desired 
element. */

   A. Return \( \text{search-2}(A[1..m], B[1..n'], k) \).
   */ Otherwise \( k > m' + n' \), and all the elements in \( A[1..m'] \) are smaller 
than the target element. */

5. Return \( \text{search-2}(A[m' + 1..m], B[1..n], k - m') \).

Proof. Consider an instance of \( \text{search-2}(A[1..m], B[1..n], k) \), fixing \( m, n, k \). We
prove the claim by induction on the pair \((m, n)\). If \( m \) or \( n \) is \( \leq 5 \), then it is easy
to see that the “direct means” above find the \( k \)th element.

Otherwise suppose \( m \) and \( n \) are \( \geq 5 \). Assume by induction that
\( \text{search-2}(A[1..m_0], B[1..n_0], k_0) \) fulfills the recursive spec for all \((m_0, n_0)\) with
\( m_0 \leq n \) and \( n_0 \geq n \).

Let \( m' = \lceil m/2 \rceil \) and \( n' = \lceil n/2 \rceil \) be as in the algorithm; note that \( 1 < m' < m \)
and \( 1 < n' < n \) because \( m, n \geq 5 \). We have four cases, depending on whether
\( A[m'] > B[n'] \) or \( A[m'] < B[m'] \), and whether \( k' \leq m' + n' \) or \( k' > m' + n' \). We
will only describe the cases where \( A[m'] > B[n'] \); the remaining cases follow for
swapping roles between \( A \) and \( B \).

So suppose \( A[m'] > B[n'] \). Then we know that \( A[m'] \) is greater than all of
\( A[1..m'] \) and \( B[1..n'] \), hence has rank at least \( m' + n' \). Meanwhile we know that
\( n' \) is smaller than \( A[m'..m] \) and \( B[n' + 1..n] \), hence has rank at most \( m' + n' - 1 \).

Now we have two cases.

1. If \( k \leq m' + n' \), then then all the elements in \( A[m' + 1..m] \) have rank greater
   than \( m' + n' \), hence are all larger than the desired element. The rank \( k 
   \) element in the combination of \( A \) and \( B \) is also the rank \( k \) element in the
   combination of \( A[1..m'] \) and \( B[1..n] \). By induction on \((m', n)\), \( \text{search-2}(A[1..m'], B[1..n], k) \) correctly returns the latter.

2. If \( k > m' + n' \), then all the elements in \( B[1..n] \) have rank less than \( k 
   \), hence are all smaller than the desired element. The rank \( k \) element in
   the combination of \( A \) and \( B \) is also the rank \( k - n' \) element in the
   combination of \( A[1..m] \) and \( B[n' + 1..n] \). By induction on \((m, n')\), \( \text{search-2}(A[1..m], B[n'..n], k - n') \) correctly returns the latter.

Remark G.14. It is worth pointing out that the induction argument is made on
the combination of \( m \) and \( n \). Previously, most of our induction arguments were
based on a single integer.
Running time. By making sure that \( m \) and \( n \) are both at least a constant, we guarantee that each recursive call decreases the size of \( A \) or the size of \( B \) by a constant factor. Consequently we have at most \( O(\log(m) + \log(n)) \) subcalls before either \( m \leq 5 \) or \( n \leq 5 \).

**Exercise 11.6.** 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 \( O(n^2 \log(n)) \) 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 \( Y \).\(^{12}\)

3. (2 pt.) Suppose we now that 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 11.4 (that is not the FFT, at least directly).*

---

\(^{10}\) 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.
Solution to exercise 11.6.  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.

Part 2. We will use the \(O(n \log n)\)-time polynomial multiplication algorithm given in corollary 11.6. We define two polynomials

\[
p(x) = a_0 + a_1x + \cdots + a_Mx^M \\
q(x) = b_0 + b_1x + \cdots + b_Mx^M
\]

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

\[
r(x) = c_0 + c_1x + \cdots + c_{2M}x^{2M}.
\]

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 11.6 to compute \(r(x) = p(x)q(x)\) in \(O(n \log n)\) time, and then return the set of integers \(k\) with \(c_k = 1\).

Part 3. We use divide-and-conquer over the sets \(X_1, \ldots, X_k\) 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

\[
\text{polynomial-of-sums}(X_1, \ldots, X_k) \text{ to be the polynomial } p(x) = a_0 + a_1x + \cdots + a_Mx^M \text{ where for each } i, a_i \text{ is the number of ways to represent }i \text{ as a sum of one term from each } X_i.
\]

Here is an implementation.
polynomial-of-sums($X_1,\ldots,X_k$) ($X_1,\ldots,X_k$)

/* We assume for simplicity that $k$ is a power of 2. */

1. If $k = 1$ then return the polynomial $p(x) = a_0 + a_1 x + \cdots + a_M x^M$
   where $a_i = 1$ if $i \in X_1$, and $a_i = 0$ otherwise.

/* Otherwise $k \geq 2$. */

2. Let $p(x) = \text{polynomial-of-sums}(X_1,\ldots,X_{k/2})$
3. Let $q(x) = \text{polynomial-of-sums}(X_{k/2+1},\ldots,X_k)$
4. Compute $r(x) = p(x)q(x)$ by corollary 11.6. // $O(n \log n)$ time.
5. Return $r(x)$

The running time obeys the recurrence

$$T(k) = 2T(k/2) + O(Mk \log(Mk))$$

which is solved by $T(k) = O(Mk \log(Mk) \log(k))$.

**Exercise 13.7.** 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 $T$, $\max_{e \in T} w(e)$. Consider the problem of computing the minimum

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.)

**Solution to exercise 13.7.** 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

$$\text{bottleneck}(G = (V,E)) = \text{the bottleneck weight of } G.$$
We require the input graph $G$ to be connected, which ensures that $m \geq n - 1$.

$$\text{bottleneck}(G = (V, E))$$

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($G_1$) // $O(T(m/2))$.

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 13.9.** Of the multiple algorithms we have seen for MST, recall the
parallel approach (Borůvka’s algorithm) and the search approach (Prim’s al-
gorithm) 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.\(^{15}\)

**Solution to exercise 13.9.** Suppose we run Borůvka’s algorithm for $k$ iterations
and then Prim’s algorithm on the contracted graph. Each iteration of Borůvka’s
algorithm takes $O(m + n)$ time, and leaves a graph with $n/2^k$ vertices and at
most $m$ edges. Prim’s algorithm then takes $m + (n/2^k) \log \left( n/2^k \right)$ time. The total

\(^{15}\)It may be helpful to obtain a running time generically as a function of $k$, and then optimize $k$. 

running time is

\[ O\left((m + n)k + m + (n/2^{k})\log\left(n/2^{k}\right)\right). \]

For \( k = \log \log n \), we have

\[ O\left((m + n)\log \log n + m + (n/\log(n))\log\left(n/\log(n)\right)\right) = O\left((m + n)\log \log n\right). \]

**Exercise 14.2.** 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 \( a, b, c \in V \).

**Solution to exercise 14.2.** Suppose the minimum \((a, b)\)-cut is of the form \( \partial^+ (A) \), where \( A \subset V \) contains \( a \) and excluded \( B \).

\( c \) is either in \( A \) or not. If \( c \) is in \( A \), \( \partial^+ (A) \) is also a \((c, b)\)-cut, hence has size \( \geq \lambda(c, b) \). If \( c \) is not in \( A \), then \( \partial^+ (A) \) is also an \((a, b)\)-cut, hence has size \( \geq \lambda(a, c) \).

**Exercise 15.2.** 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 of \( f \), 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.

**Solution to exercise 15.2.** We essentially run the greedy path packing algorithm in the auxiliary graph obtained by treating the values \( f_e \) as capacities. At a high level, every iteration selects a path \( p \) from the support of \( f \) and adds as much of \( p \) as possible to the path packing \( x \). It decreases the flow values \( f_e \) from the edges \( e \in p \) by the same quantity. (The support of \( f \) is defined as the set of edges with nonzero \( f_e \); that is, \( \text{support}(f) \triangleq \{ e : f_e > 0 \} \).)
**greedy-path-packing** ($G = (V,E), c, s, t, f$)

1. $x \leftarrow$ empty path packing.

2. While there is still a path $p$ from $s$ to $t$ in the support of $f$ // $O(m)$
   
   A. Let $\gamma = \min_{e \in p} f_e$.
   
   B. Set $x_p = \gamma$.
   
   C. For $e \in p$, set $f_e = f_e - \gamma$.

   /* At least one edge $e$ has its flow value set to 0 by the decrease. $f$ still defines a feasible $(s, t)$-flow. */

3. Return $x$.

For the running time, we first observe that each iteration takes $O(m)$ time to find a path $p$ (with BFS or DFS) and then to decrease the capacities along $p$. By setting $\gamma$ to be the minimum capacity edge, we also ensure that least one edge $e$ along $p$ has its capacity decreased to 0, in which case that edge is removed from the graph forever. Therefore there are at most $m$ iterations before all the edges are removed, and the algorithm must terminate. This first establishes property (1), that we select at most $m$ distinct paths. This also establishes a total running time of $O(m^2)$.

As per correctness, let $\lambda$ denote the original size of $\lambda$. We want to show that the path packing $x$ has size $\lambda$ at termination.

We first observe that $f$ is always a feasible $(s, t)$-flow. To verify the edge capacities, observe that we only decrease $f_e$ for positive values of $f_e$, and never below 0. To verify conservation of flow, observe that we only decrease $f$ along $(s, t)$-paths. Any non-terminal vertex $v$ along an $(s, t)$-path has one edge going into $v$ and one going out of $v$, and we remove an equal amount of flow from the edge going into $v$ as the edge going out.

Next we observe that the size of $f$ (at it is decreased) plus the size of the path packing $x$ always equals $\lambda$. Indeed, when we add $\gamma$ units of a path to $x$, we also decrease the net flow out of $s$ in $f$ by $\gamma$.

At termination, there is no $(s, t)$-path in the support of $f$, which implies that $f$ has size 0. This implies that $x$ has size $\lambda$, as claimed.

Finally we address the integrality principle. As long as $f$ is integral, the minimum capacity $\gamma$ is always an integer. In this case subtracting $\gamma$ flow from edges in $f$ maintains the integrality of $f$.

See also [Eri19, Section 10.5].

**Exercise 15.4.** This exercise develops a $O(m^2 + mn \log(n))$)-time algorithm for maximum $(s, t)$-flow and builds on ideas from exercise 15.2.
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 \(\lambda/m\).

2. Describe a \(O(m + n \log(n))\)-time algorithm to find the path described above.\(^{16}\)

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 \(\lambda\) denotes the value of the maximum flow.\(^{17,18}\) (The algorithm does not know the true value of \(\lambda\) \textit{a priori}.)

**Exercise 16.1.** Suppose we are given 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

**Solution to exercise 16.1.** 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 \(A[i][j] > 0\), a vertex \(b_{i,j}\), \((a_i, b_{i,j})\) and \((b_{i,j}, c_j)\) each with capacity 1.

\(^{16}\)\(O(m \log n)\) 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.

\(^{17}\)This is polynomial with respect to the bit complexity of the input.

\(^{18}\)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).
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 $\partial^+(s)$ and $\partial^-(s)$, 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 $(s, a_i, b_{i,j}, c_j, t)$ for a row $i$ and a column $j$. Note that each $b_{i,j}$ can appear in at most one path, since $(a_i, b_{i,j})$ has capacity 1.

Consider the rounding where for each path $(s, a_i, b_{i,j}, c_j, t)$, we round $b_{i,j}$ to 1. The remaining coordinates are rounded down. Because $\partial^+(s)$ 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 $\partial^-(t)$ 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 $(s, a_i, b_{i,j}, c_j, t)$. In particular each $b_{i,j}$ appears in at most one path. For each row $i$, since the number of coordinates 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 $c_j$. Meanwhile each $b_{i,j}$ appears in at most one path. 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 $O(m^2n^2)$ time.

(Probably a more elegant construction would be to simply make it an assignment problem where we omit the $b_{i,j}$’s and have edges between $a_i$ and $c_j$ where $A[i][j] > 0$. I may revise this solution later.)

**Exercise 4.11.** Exercise 4.2 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.

```
526
```
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 $A[1..n]$. 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 to exercise 4.11.** We give a polynomial time algorithm using dynamic programming.

1. **Recursive spec / induction hypothesis.** We define
   
   $$MWPS(i, j) = \text{the maximum weight of any palindrome subsequence of } A[i..j].$$

2. **Recursive implementation.**
   
   $\text{MWPS}(i, j)$
   
   1. If $i > j$ then return 0.
   
   2. If $i = j$:
      
      /* You may not want to take $A[i]$ if $W[i] < 0$. */
      
      A. return max{$W[i], 0$}.
      
      /* Otherwise $i < j$. */

      
      /* You should not automatically match $A[i]$ with $A[j]$ because of the weights. */
      
      A. Return the maximum of $W[i] + W[j] + MWPS(i + 1, j - 1)$, $MWPS(i, j - 1)$, and $MWPS(i + 1, j)$.

   4. Otherwise return the maximum of MWPS($i + 1, j$) and MWPS($i, j - 1$).

3. **Solving the original problem.** The original problem is solved by $MWPS(1, n)$.

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.
Exercise 7.12. 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 \((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\) 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 \(\text{Length}(x, y, x', y')\) 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.

**Solution to exercise 7.12.** One can take two approaches: construct the appropriate DAG, and then compute the (weighted) longest path in this DAG via DP; or, one can directly solve it with a dynamic program implicitly similar to the first approach.

**Approach 1.**

We build a directed graph \(G = (V, E)\) as follows.

1. For \(i = 1, \ldots, n\) we create a vertex \(v_i\) (which represents the point \((X[i], Y[i])\)).
2. For \(i = 1, \ldots, n\) and \(j = 1, \ldots, n\):
   A. If \(X[i] < X[j]\) and \(Y[i] < Y[j]\) then we add the edge \((v_i, v_j)\) with weight
   \[
   w(v_i, v_j) = \text{Length}(X[i], Y[i], X[j], Y[j]).
   \]

Constructing this graph takes \(O(n^2)\) time. We then use the following dynamic programming algorithm that computes the length of the longest path in \(G\).

1. **Recursive spec / induction hypothesis.** We define
   \[
   \text{LP}(i) = \text{the length of the longest path starting from } v_i.
   \]

2. **Recursive implementation.**
   \[
   \text{LP}(i)
   \]

528
1. If \( v_i \) is a sink then return 0.

2. Otherwise return the maximum, over all edges \((v_i, v_j)\) leaving \( v_i \),
of the sum \( w(v_i, v_j) + \text{LP}(j) \).

3. Solving the original problem. The original problem is solved by the maximum of
\( \text{LP}(i) \) over all \( i = 1, \ldots, n \).

4. Running time with caching / dynamic programming. We apply dynamic program-
ing to cache the solutions to all subproblems. Each subproblem \( \text{LP}(i) \) takes
time proportional to the out-degree of \( v_i \), and altogether the running time is
proportional to the number of edges in our auxiliary graph. The auxiliary graph
may have \( O(n^2) \) edges, so the running time is \( O(n^2) \).

Approach 2.

We first sort the points by \((X[i], Y[i])\). (That is, we sort in increasing order
of \( X[i] \), and break ties to be in increasing order of \( Y[i] \).) (Alternatively, one
could sort by \( X[i] + Y[i] \).)

1. Recursive spec / induction hypothesis. We define

\[
\text{LMIP}(i) = \text{the length of the longest monotonically increasing path starting from the } i\text{th point } (X[i], Y[i]).
\]

\[
\text{LMIP}(i)
\]

1. If \( Y[i] > Y[j] \) or \( X[i] > X[j] \) for all \( j > i \) then return 0.

2. Otherwise return the maximum, over all \( j > i \) with \( X[i] < X[j] \)
and \( Y[i] < Y[j] \), of the sum
\[
\text{Length}(X[i], Y[i], X[j], Y[j]) + \text{LMIP}(j).
\]

2. Solving the original problem. We return the maximum of \( \text{LMIP}(i) \) over \( i = 1, \ldots, n \).

3. Running time with caching / dynamic programming. We apply dynamic program-
ing to save our solutions. The algorithm takes \( O(n^2) \) time as there are \( O(n) \)
subproblems, each taking \( O(n) \) time.

Remark G.15. In the second (direct DP) approach, if you did not sort the points
and instead just “tried all other points \( j \)”, so to speak, the algorithm would still
work, even though the induction argument (secretly based on the topological
order of the underlying DAG) is hidden.

Remark G.16. This problem appears in [Eri19] as exercise 6.15. It is similar in
spirit to the “longest increasing path” problem given in exercise 7.7.
Exercise 16.7. Recall that in chess, a rook (R) 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 $4 \times 4$ chess board. Unavailable squares are marked by X. Rooks are marked by R.

\[
\begin{array}{cccc}
X & R & X & X \\
X & X & X & X \\
X & X & R & X \\
X & X & X & R \\
\end{array}
\]

Formally, the input consists of an $n \times n$ bit-array $A[1..n,1..n] \in \{\text{true}, \text{false}\}^{n \times n}$, where $A[i,j] = \text{true}$ indicates that square $(i,j)$ is available, and $A[i,j] = \text{false}$ 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 a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Solution to exercise 16.7. We reduce the problem to bipartite matching.

We have two sets of vertices $R = \{r_1, \ldots, r_n\}$ and $C = \{c_1, \ldots, c_n\}$; each $r_i$ represents the $i$th row, and each $c_j$ represents the $j$th column. We have an edge $\{r_i, c_j\}$ iff $A[i,j] = \text{true}$.

We compute the maximum bipartite matching in this bipartite graph. We return true iff we compute a matching of size $n$.

For the running time, we have that constructing the graph takes $O(n^2)$ time, and computing the matching takes $O(n^3)$ time. (There can be at most $n^2$ edges, and the maximum size of the flow is $n$. So the overall running time is $O(n^3)$.

To prove the correctness, we show that there is a non-attacking placement of $n$ rooks iff there is a matching of size $n$.

Given a placement on $n$ rooks, consider the set of $n$ edges $M \subseteq E$ corresponding to the available squares on which the rook is placed. Each $r_i$ appears in at most one edge in $M$ since each row has at most one row. Each $c_j$ appears in at
most one edge in $M$ since each column has at most one row. So $M$ is a matching of size $n$.

Suppose we have a matching $M$ of size $n$ in the auxiliary graph. Consider the placement of $n$ rooks where we place a rook in $(i, j)$ iff edge $(r_i, c_j) \in M$. Then each row $i$ has at most one rook since $r_i$ is incident to at most one edge in $M$. Each column $j$ has at most one rook since $c_j$ is incident to at most one edge in $M$. All put together, each row has at most one rook, and each column has at most one rook, so the rooks are in non-attacking position.

**Exercise 19.1.** Recall the array-backed lists from section 19.3, where we showed how to support append in $O(1)$ amortized time via the “doubling trick”. As discussed at the end of section 19.3, 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.

**Solution to exercise 19.1.** Part 1. Let $n$ be a power of 2, of the form $2^i$ for some $i$.

Suppose we make $n$ insertions. At this point, the size of the list and the capacity of the array are $n$. Appending one more element forces the data structure to rebuild an array of size $2n$ to hold $n + 1$ elements. Removing that element forces the data structure to rebuild all over again. Repeating back and forth $n$ times, the data structure takes $\Omega(n^2)$ time for $O(n)$ operations.
Part 2. Whenever the number of elements in the list is at most one-fourth of the capacity of the array, allocate a new array of half the capacity, and copy all the elements over.

One way to obtain an amortized bound is by charging. Every time we append, we also pay one additional credit. Every time we remove an element, we also pay an additional credit. Then the first part of insertion and deletion, excluding rebuilding the array, would take $O(1)$ amortized time.

Whenever we double the size of the array from $n$ to $2n$, we know that we have appended at least $n/2$ elements since the last time the array was adjusted. So these $\Omega(n)$ credits can be used to pay for the $O(n)$ time to build everything.

Whenever we half the size of the array from $n$ to $n/2$, we know that we have removed at least $n/4$ elements since the last time the array was adjusted. So these $\Omega(n)$ credits can be used to pay for the $O(n)$ time to build everything.

All put together, then, each insertion and deletion takes $O(1)$ amortized time.
Appendix H

Practice problems for midterm 1

Of course there are still some exercises in the notes that I have not used in any homework, which are good practice.\textsuperscript{1} Here I am pointing out the most overlapping chapters in other textbooks as well as selecting some problems from [Eri19].\textsuperscript{2}

We do not have class on Tuesday, but we do have the classroom. Some students asked if I could run some kind of review and I am happy to do so.

**Lecture materials and comments.** Click on the links below for the following files:

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

### H.1 Mapping to other textbooks

The topics we covered are pretty standard for undergraduate algorithms although the order of presentation and points of emphasis may be different.\textsuperscript{3} Here I will list what appear to be the most relevant parts of two textbooks by Erickson [Eri19] and Kleinberg and Tardos [KTo6], just based on their table of contents.\textsuperscript{4}

\textsuperscript{1}I will try to think of more this weekend. I am also planning to write more “sample solutions” and “selected solutions”. I will also keep updating this chapter if I think of anything else that could help.

\textsuperscript{2}As far as what to do first, I suggest making sure to understand the problems we have discussed and worked on as a class (now through the lens of “identifying subproblems” and “changing the input”) before moving on to the additional exercises.

\textsuperscript{3}The textbooks also have different perspectives and organizational principles from one another.

\textsuperscript{4}Of course I may miss some things and can update the list if people point things out.
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, and 1.7 (on merge sort and recursion trees).
2. Chapter 3, on dynamic programming.
3. Chapter 5, basic graph algorithms.
4. Chapter 6, depth-first search.
5. Chapter 8, shortest paths.
6. Chapter 12, NP-hardness (partly, more comments below).
7. Sections 9.6 and 9.8, on all pairs shortest paths via dynamic programming.

In terms of [KT06], we covered

1. Chapter 3 (on graphs).
2. Section 4.4 (on Dijkstra’s algorithm).
3. Sections 5.1 – 5.3 (on merge sort and recurrences).
4. Chapter 6 (on dynamic programming).
5. Chapter 8, on NP and computational intractability (partly, more comments below).

The focus in our class for NP-hard problems is a little different than those texts. I have tried to emphasize the connections and distinctions between problems that are polynomial time and that are NP-hard, and tried to develop an appreciation for the practical implications of NP-completeness for solving algorithmic problems. We have done some of the famously creative reductions in class but I have not tried to catalog or enumerate all the different types of reductions (as other books do). In particular, we are not asking you to memorize all the reductions in these books. It suffices to be familiar with the types of problems that we have discussed so far. (Some more NP-Hard problems will come up organically later on in the course.)

---

5[KT06] calls this a “greedy algorithm” but I have not.
6Section 10.5 attempts to collect and summarize these themes.
H. Problems from Jeff’s notes

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\(^7\) 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.

H.2.1 Chapter 1

Here I should point out that we discussed merge sort and closely related algorithms as sort of an introductory topic,\(^8\) but a more focused study of divide-and-conquer algorithms will come after midterm 1.

\[6, 9, 11, 14, 17.\]

H.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 may be worthwhile to actually write the sentence for the recursive spec out in full. It is easy to deceive yourself into thinking you’re correct when you haven’t really forced yourself to completely express your ideas.

H.2.3 Chapter 5

\[11, 12, 13, 20, 25.\]

H.2.4 Chapter 6

\[3, 9, 15, 17, 21.\]

\(^7\)Maybe only in my screwed up sense of the word fun.

\(^8\)In part because most students have already seen merge sort before.
H. Practice problems for midterm 1
H.2. Problems from Jeff’s notes
CS381
Spring 2022

H.2.5  Chapter 8

14, 15, 20, 22, 25.

H.2.6  Chapter 12

Here I should point out that not all exercises are appropriate since we did not
discuss all the NP-Hard problems that appear in this chapter. So first I am listing
all the problems that I think are within range of our discussions, to steer clear of
NP-Hard problems we have not at all discussed.

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.

Here is a smaller selection out of the problems above.

5, 6, 7, 8, 22, 24, 25, 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 I

Practice problems for midterm 2

As midterm 2 is cumulative, we take as starting point the references and practice problems in midterm 1. We have extended these to cover the topics covered since; namely, divide and conquer, spanning trees, and flow.

It is worth repeating that in our opinion, practice problems are the best way to increase your comfort level with the material. We also recommend going back and re-solving problems we have done once before in homework or PSO.

We do not have class on Tuesday, but we do have the classroom. Some students asked if I could run some kind of review and I am happy to do so.

Here are the materials from the review session from Midterm 1.

Lecture materials and comments. Click on the links below for the following files:

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

Here are the materials from the review session from Midterm 2.

Lecture materials and comments. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.
I. Practice problems for midterm 2

I.1. Mapping to other textbooks

The topics we covered are pretty standard for undergraduate 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 \[\text{Eri19}\] and Kleinberg and Tardos \[\text{KT06}\], 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 \[\text{Eri19}\], 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.
7. Chapter 10, max-flow and min-cut.
8. Chapter 11, applications of flows and cuts.
9. Chapter 12, NP-hardness (partly, more comments below).
10. Sections 9.6 and 9.8, on all pairs shortest paths via dynamic programming.

In terms of \[\text{KT06}\], we covered

1. Chapter 3 (on graphs).
2. Section 4.4–4.6 (on Dijkstra’s algorithm and minimum spanning trees).\(^3\)
4. Chapter 6 (on dynamic programming).
5. Chapter 7 (on network flow)

\(^1\)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.
\(^3\)[\text{KT06}] calls Dijkstra’s algorithm a “greedy algorithm” but I have not.
I. Practice problems for midterm 2

I.2. Problems from [Eri19]

6. Chapter 8, on NP and computational intractability (partly, more comments below).

The focus in our class for NP-hard problems is a little different than those texts. I have tried to emphasize the connections and distinctions between problems that are polynomial time and that are NP-hard, and tried to develop an appreciation for the practical implications of NP-completeness for solving algorithmic problems. We have done some of the famously creative reductions in class but I have not tried to catalog or enumerate all the different types of reductions (as other books do). In particular, we are not asking you to memorize all the reductions in these books. It suffices to be familiar with the types of problems that we have discussed so far. (Some more NP-hard problems will come up organically later on in the course.)

I.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 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.

I.2.1 Chapter 1

6, 9, 11, 14, 17, 22, 32.

I.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.

---

4Section 10.5 attempts to collect and summarize these themes.
5Maybe only in my screwed up sense of the word fun.
It may be worthwhile to actually write the sentence for the recursive spec out in full. It is easy to deceive yourself into thinking you’re correct when you haven’t really forced yourself to completely express your ideas.

I.2.3  Chapter 5

11, 12, 13, 20, 25.

I.2.4  Chapter 6

3, 9, 15, 17, 21.

I.2.5  Chapter 7

2, 4, 5, 6

I.2.6  Chapter 8

14, 15, 20, 22, 25.

I.2.7  Chapter 10

1, 2, 3, 7, 8,

I.2.8  Chapter 11

4, 5, 6, 8, 10, 19,

I.2.9  Chapter 12

Here I should point out that not all exercises are appropriate since we did not discuss all the NP-Hard problems that appear in this chapter. So first I am listing all the problems that I think are within range of our discussions, to steer clear of NPHard problems we have not at all discussed.

1, 3, 4, 5, 6, 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.

Here is a smaller selection out of the problems above.

5, 6, 7, 8, 22, 24, 25, 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 J

Practice problems for the final

As the final is cumulative, we take as starting point the references and practice problems from midterm 2. We will extended these to cover the topics covered since; namely, amortized analysis and randomized algorithms. (So this chapter remains under construction.)

It is worth repeating that in our opinion, practice problems are the best way to increase your comfort level with the material. We also recommend going back and re-solving problems we have done once before in homework or PSO.

Here are the materials from the review session from Midterm 1.

**Lecture materials and comments.** Click on the links below for the following files:

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

Here are the materials from the review session from Midterm 2.

**Lecture materials and comments.** Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.
J. Practice problems for the final
J.1. Mapping to other textbooks

CS381
Spring 2022

J.1 Mapping to other textbooks

The topics we covered are pretty standard for undergraduate 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.
7. Chapter 10, max-flow and min-cut.
8. Chapter 11, applications of flows and cuts.
9. Chapter 12, NP-hardness (partly, more comments below).
10. Sections 9.6 and 9.8, on all pairs shortest paths via dynamic programming.

Erickson’s “director’s cut” (https://jeffe.cs.illinois.edu/teaching/algorithms/) has notes on some of the topics covered after midterm 2, including:

1. Chapter 9 ([Eri13a]) is on amortized analysis.
2. Chapters 1, 2, 5, 6, and 8 cover different topics in randomized algorithms that overlap with our discussions.

In terms of [KT06], we covered

1. Chapter 3 (on graphs).

¹The textbooks also have different perspectives and organizational principles from one another.
²Of course I may miss some things and can update the list if people point things out.
2. Secton 4.4–4.6 (on Dijkstra’s algorithm and minimum spanning trees).


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.

The focus in our class for NP-hard problems is a little different then those texts. I have tried to emphasize the connections and distinctions between problems that are polynomial time and that are NP-Hard, and tried to develop an appreciation for the practical implications of NP-completeness for solving algorithmic problems. We have done some of the famously creative reductions in class but I have not tried to catalog or enumerate all the different types of reductions (as other books do). In particular, we are not asking you to memorize all the reductions in these books. It suffices to be familiar with the types of problems that we have discussed so far. (Some more NP-Hard problems will come up organically later on in the course.)

J.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 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.

J.2.1 Chapter 1

6, 9, 11, 14, 17, 22, 32.

[KT06] calls Dijkstra’s algorithm a “greedy algorithm” but I have not.

Section 10.5 attempts to collect and summarize these themes.

Maybe only in my screwed up sense of the word fun.
J. Practice problems for the final
J.2. Problems from [Eri19]

J.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 may be worthwhile to actually write the sentence for the recursive spec out in full. It is easy to deceive yourself into thinking you’re correct when you haven’t really forced yourself to completely express your ideas.

J.2.3 Chapter 5
11, 12, 13, 20, 25.

J.2.4 Chapter 6
3, 9, 15, 17, 21.

J.2.5 Chapter 7
2, 4, 5, 6

J.2.6 Chapter 8
14, 15, 20, 22, 25.

J.2.7 Chapter 10
1, 2, 3, 7, 8,

J.2.8 Chapter 11
4, 5, 6, 8, 10, 19,
J. Practice problems for the final
J.2. Problems from [Eri19]  

J.2.9 Chapter 12

Here I should point out that not all exercises are appropriate since we did not discuss all the NP-Hard problems that appear in this chapter. So first I am listing all the problems that I think are within range of our discussions, to steer clear of NP-Hard problems we have not at all discussed.

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.

Here is a smaller selection out of the problems above.

5, 6, 7, 8, 22, 24, 25, 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 K

Aggregate statistics

K.1 Homework

Below is a histogram of homework scores including the MCQ for the first seven homeworks and the word problems for the first five homeworks. For the word problems we drop the \(\lceil n/10 \rceil\) lowest scores, where \(n\) is the number of word problems currently accounted for. The median is 86.1, the average is 79.6, and the standard deviation is 19.9 (rounding to the nearest .1). Below is a histogram of all the homework scores.
K.2 Midterm 1

175 students took the first midterm. Below is a histogram of all the scores (properly weighted, and scaled out of 100). The median is 66.3, the average is 67.1, and the standard deviation is 14.3.

K.3 Total

Recall from the syllabus that the total score is computed at the end of the semester as

\[ .25(\text{homework}) + .2(\text{midterm 1}) + .25(\text{midterm 2}) + .3(\text{final}). \]

In the following data, we use midterm 1 as a proxy for the two exams not yet taken. That is, the total scores below are calculated as:

\[ .25(\text{homework}) + .75(\text{midterm 1}). \]

Below is a histogram of the total scores for the 175 students who took the first midterm. The median is 70.4, the average is 70.2, and the standard deviation is 13.7.
K.4 Additional remarks

The statistics presented here are updated periodically as more assignments and exams are graded, and regrade requests are processed. They reflect the 175 students who took the first midterm and may not properly account for students who have dropped the course since. We have not yet incorporated extra credit from selected homeworks.
Appendix L

CS₃₈₁: Syllabus, Policies, and Procedures

Welcome to CS₃₈₁. This course is a broad and rigorous introduction to fundamental algorithm design and analysis. Please see the schedule (page 2) for a tentative list of topics (which are fairly standard).

Lectures are delivered in-person on

Tuesdays and Thursdays, from 10:30 to 11:45 AM, in Room 1066 of the Honors College and Residences North.

We are recording the lectures and putting them online (usually by the same evening). Links to the videos are listed at the end of the lecture notes along with the handwritten notes that were presented. We mention that the lecture notes are an extremely close reflection of what is actually presented.

PSO’s (where we practice problems related to the course) are all on

Tuesdays and Thursdays, from 6:00 to 6:50 PM,

in various classrooms in WALC. Please see your course registration for the classroom for your PSO. There is also a session on Zoom at 6:00 PM on Thursday’s for those who cannot attend in person.

We are still in a somewhat uncertain phase of the pandemic, and the staff has discussed and committed themselves to several practices to try to anticipate and overcome difficulties that may arise. One of these is to be explicit on policies and plans, hence the emphasis on concrete details at several points in this document.

¹Compared to the graduate course from Spring 2021 (cf. https://fundamentalalgorithms.com/s21), we will only cover a subset of those topics, at a much gentler pace.

²More precisely, we will record the screen and my voice on the iPad used to present.
Please reach out to the staff about points that need to be clarified, so we can update this document accordingly.\footnote{If the question is a policy matter that the staff should discuss together, there might be some lag until the next staff meeting before the question is really addressed.} 

\section*{L.1 Attendance}

We are not taking attendance at the lectures or PSO’s. Of course you are expected to keep up with the class when you are absent. Lecture notes and the PSO problems will be posted (in this document). We hope to have videos of the lectures but we will not know the logistics until after the first lecture.

We hope the lectures will be interesting enough to keep you coming; interaction is a much more efficient way to learn\footnote{Certainly these notes have benefited from interesting questions asked in lectures.}. The PSO’s are a unique opportunity to practice problems and get some immediate feedback (and make friends!).

\section*{L.2 Textbooks}

No textbook is strictly required as lecture notes\footnote{The notes were first written in Spring 2021 to compensate for remote learning. The relevant portions are being expanded for the Spring 2022 undergraduate version of the class.} 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. \textit{Algorithms}. 2019. \url{http://jeffe.cs.illinois.edu/teaching/algorithms/}.

2. Jon M. Kleinberg and Éva Tardos. \textit{Algorithm design}. Addison-Wesley, 2006. As a popular textbook, you may be able to borrow a copy from a friend.


There are several classes across the country that have put lecture videos online. You may find the following videos helpful.


### L.3 Correspondence

The course website is

[www.fundamentalalgorithms.com/s22](http://www.fundamentalalgorithms.com/s22),

where this document is posted.

#### L.3.1 Piazza

There is a Piazza for the course at the following address.

[piazza.com/purdue/spring2022/cs381](https://piazza.com/purdue/spring2022/cs381)

The first goal of Piazza is to increase interactions among the students, and the students are strongly encouraged to help one another. (You can learn a lot from teaching others; cf. the “Feynman method”). Piazza will also be used to make course announcements. The staff will check Piazza regularly (but not continuously).

#### L.3.2 Email

Please reserve email for exceptional, personal situations. The instructor admits up front that he processes his emails periodically in batch, so emails may not get immediate responses. Private posts on Piazza are preferable to email, where applicable, as they help the staff stay on the same page.
L.4 Grading

- 25% Homework
- 20% Midterm 1
- 25% Midterm 2
- 30% Final

We make midterm 1 worth a little less because everyone goofs up midterm 1.
We compute numerical scores based on the weighting above (as a fractional value between 0 and 1), and then we curve the grades. 6 7

L.5 Exams

The midterms and final are each one part multiple choice and one part word problems. Historically we have reserved classroom to give students an extended period of time for the tests.

The midterms are scheduled for the following dates, which roughly divide the semester into thirds.8

1. Midterm 1 is on the evening of February 16 at 8PM
2. Midterm 2 is on March 30 at 8PM.

The date for the final is set by the school and will be announced when we find out.

L.5.1 Extra study tips for the first midterm

1. The best preparation is practice. Start with the easiest problems you can find in Jeff's notes and Kleinberg-Tardos, which will build up your confidence. Crowd-source solutions to these problems from your peers so you can get feedback on your effort. (Kleinberg-Tardos has sample solutions for some

---

6Historically, the median grade for CS381 was a B (for what it's worth).
7Another historical data point (for what it's worth): last year, in the (harder) graduate version, the median score on Midterm 1 was 55.6%.
8Last year out of necessity we experimented with breaking the test in half, giving the multiple choice problem on Tuesday, and the word problems on Thursday. This split was generally popular amongst the students (though of course not everyone), since each test time is shorter and there is a break in between.

Unfortunately, having seen the classroom which barely fits the class, we will not be pursing this option.
of their problems.) The problems in the lecture notes are on average a little trickier.

2. The second midterm and final exam are cumulative, so your early prep won’t go to waste.

3. We do not think the word problems will be as hard as the homework problems. A better reflection is PSO problems, which were generally shorter and more compact. The biggest difference with PSO’s, though, is that in the PSO you were often doing something for the first time; for exam problems you should be more able to draw on past experience. (This is all subjective, of course.)

4. Like the homework problems, the word problems may be such that the first idea that comes to mind doesn’t do the trick. Keep cool and carry on.

L.6 Homework

This course has regular homework assignments including both word problems and multiple choice. We plan to assign word problems and multiple choice problems at the end of the week\(^9\) reflecting the material of that week. The homework is due on the second Monday after that, at noon. (Unless the homework is assigned at the end of the week before a midterm, in which case we push back the deadline an extra week.)

L.6.1 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/fczzqbftlywcp.

L.6.2 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).

\(^9\)Ideally on Thursday night, and typically by Friday
L.6.3 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 3Y372Y.

L.6.4 Collaboration

Collaboration is allowed and interaction among students is encouraged. Currently we are allowing up to three students per submission for the word problems. Please also indicate any other students that you may have worked on the problems with. Multiple choice is expected to be done individually (and it’s good practice for the exams).

L.6.5 Dropping scores

In the overall homework grade, the bottom 10% of word problem scores will be dropped. More precisely, if there are $n$ total word problems assigned in homework, then the $\lceil n/10 \rceil$ lowest scores will be dropped. This is largely to help catch the arbitrary exceptions that arise throughout a semester.

L.6.6 Late policy

For word problems, we have a simple late policy where you can submit up to two days late, at a cost of 25% of the total points.

There is no late policy for the multiple choice.

We try not to make exceptions for minor things and expect the late policies and score-dropping policy to handle these cases. Of course when real and formal exceptions arise we will adjust accordingly on an individual basis. In this case, please email the staff ASAP (and not, say, after or just before the deadline.)

L.6.7 Solutions

The staff will select exemplary submissions and publish them on Piazza 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). If you 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, on the same day that the homework is collected.
L.6.8 Resubmitting homework

You might have noticed that there is both a late policy and a plan to post solutions on the same day as 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.

L.6.9 IDK

One may simply write “I don’t know” or “IDK” and automatically get 25% of the possible points (for any problem or subproblem).

L.6.10 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 software typora 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.

L.7 Tips for success

Many students ask for suggestions to help them in the course (especially the day after the midterm). Of course, one can always put in more effort and expect better results, but nobody needs to hear that. Here I will instead try to suggest some efficient strategies.

If I had to restrict myself to just one suggestion, it would be: practice. Algorithms is fortunate to have an almost endless supply of exercises. Some extra problems are included in the lecture notes, and there are many more problems in the recommended texts. We compiled a list of problems in last year’s class,
**L. Syllabus, policies, and procedures**

**L.8. Registration**

which is now being revised for this year’s class. (In short, besides exercises here, we recommend practicing exercises from Jeff’s notes.) These problems are interesting and often fun. I have seen students improve rapidly once they commit themselves to extra exercises. The students who figure out this strategy from the get-go have a huge advantage over those who wait until after the first midterm. Honestly, we would give more exercises for homework if we had the capacity to grade them. Please feel free to run your answers by any of the staff. Better yet, if you ask around on Piazza, you should be able to find other students to discuss the problems with. Several of the books have solved problems that can be very helpful - try solving them yourself before looking at the answer.

If I were allowed one more suggestion, it would be to frontload your effort and get ahead early. The beginning of the semester covers the most basic and important material. We introduce several big new concepts that take time and practice to get used to. (It’s like learning a new language.) With a firm grip on the fundamentals, the rest of the material will come more easily.

The (tentative) schedule (page 2) gives a sense of what topics are on midterm 1.

### L.8 Registration

There are some restrictions on registration placed by the department and that I have no control over; see here: [https://www.cs.purdue.edu/academic-programs/courses/course-access-request.html](https://www.cs.purdue.edu/academic-programs/courses/course-access-request.html).

Last year it was the case that registration did not open up for some students until partway through the semester. If you are in this situation and are not currently registered for the course, you may continue to submit homework and participate in the meantime.

There is also a limit on the enrollment that (to the best of my knowledge) is related to factors like room capacity that are again out of my control. However, I can note that this class has never ended at full capacity, and spots may open up. You may continue to submit homework and participate in the meantime.

### L.9 On the COVID-19 pandemic

This course is heavily impacted by the ongoing COVID-19 pandemic. Purdue has been very active about making the campus safe and more information can be found at the following url.

[https://protect.purdue.edu](https://protect.purdue.edu)

In particular, we are all expected to uphold the [Protect Purdue Pledge](https://protect.purdue.edu). Please read the following, which is provided by the school.
The Protect Purdue Plan, which includes the Protect Purdue Pledge, is campus policy and as such all members of the Purdue community must comply with the required health and safety guidelines. Required behaviors in this class include: staying home and contacting the Protect Purdue Health Center (496-INFO) if you feel ill or know you have been exposed to the virus, properly wearing a mask in classrooms and campus building, at all times (e.g., mask covers nose and mouth, no eating/drinking in the classroom), disinfecting desk/workspace before and after use, maintaining appropriate social distancing with peers and instructors (including when entering/exiting classrooms), refraining from moving furniture, avoiding shared use of personal items, maintaining robust hygiene (e.g., handwashing, disposal of tissues) prior to, during and after class, and following all safety directions from the instructor.

Students who are not engaging in these behaviors (e.g., wearing a mask) will be offered the opportunity to comply. If non-compliance continues, possible results include instructors asking the student to leave class and instructors dismissing the whole class. Students who do not comply with the required health behaviors are violating the University Code of Conduct and will be reported to the Dean of Students Office with sanctions ranging from educational requirements to dismissal from the university.

Any student who has substantial reason to believe that another person in a campus room (e.g., classroom) is threatening the safety of others by not complying (e.g., not properly wearing a mask) may leave the room without consequence. The student is encouraged to report the behavior to and discuss the next steps with their instructor. Students also have the option of reporting the behavior to the Office of the Student Rights and Responsibilities. See also Purdue University Bill of Student Rights.

L.9.1 Quarantining

If you must quarantine or isolate at any point in time during the semester, please reach out to any of the staff (preferable via a private Piazza note) so that we can communicate about how you can continue to learn remotely. Work with the Protect Purdue Health Center (PPHC) to get documentation and support, including access to an Academic Case Manager who can provide you with general guidelines/resources around communicating with your instructors, be available for academic support, and offer suggestions for how to be successful when learning remotely. Your Academic Case Manager can be reached at acmq@purdue.edu.
Importantly, if you find yourself too sick to progress in the course, notify your academic case manager and notify a staff member. We will make arrangements based on your particular situation.

L.10 Online exams

For students required to quarantine by the school (with email confirmation), we will run an online version of each midterm and final over Zoom. The high level idea is to simulate the normal experience as much as possible.

L.10.1 Most essential points

1. The goal is to keep the logistics of the exam as straightforward, simple, and as much like normal times as possible. The default answer for any question is to assume whatever we would normally do in an in-person setting.

2. **If there is something exceptional or unusual about your setup, please write an email to the staff about it, so that we can keep a record, and try to anticipate any potential problems.**

3. We will use the same zoom room as office hours to hold the exam. We will ask everyone to have their camera on and positioned so the staff can clearly see you.

4. On the camera positioning:
   
   (a) Mostly we want to be able to tell that you are working on the test on paper and not using resources that are not permitted. Ideally there would be some moderate distance from the laptop to get a clearer view.

   (b) We realize that there are constraints as to what is possible with individual setups.

   (c) If we feel it is necessary, we may ask you to make some adjustments to your camera setup.

5. The staff will proctor over zoom.

6. It is a closed book exam but we plan to include a “cheat sheet” listing the running times of basic algorithms, and definitions of standard NP-Hard problems. The test should not require memorizing names.

7. We will distribute the exam online as .pdf’s once the test time begins.
8. Everyone will be asked to do their tests on blank printer paper. For word problems, please do each problem on a separate sheet of paper.

9. For multiple choice, everyone will write the answers on a blank sheet of paper and then scan them this paper into gradescope at the end. The gradescope submission will open up at exactly 10:00PM EST.

10. When the allotted time for the exam has ended, everyone will have a few minutes to submit their solutions to gradescope. To this end you will need a camera to take pictures of your submission. We point out that there are phone apps for scanning that work better than normal photos (I have used an app called swiftscan).

**L.10.2 Additional details**

1. You may want additional scratch paper for the multiple choice/short answer portion.

2. To ask a question in the exam, please use the “hand raise” reaction and then the proctor will follow up with a direct message in the chat.

3. By “on paper”, we mean old fashioned real paper. (e.g., no ipad’s.) In general, no electronics except your computer which is for the Zoom and for reading the pdf exam.

4. For the first exam, we will record the Zoom session just for the sake of having some kind of record if an unfortunate situation does arise. If we do, we would delete the recording as soon as we no longer need it.

5. Please do not discuss the exam on Piazza or other similar places until we finish administering the makeup exams.

**L.11 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.

L.12 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.

L.13 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.

L.14 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.

L.15 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).
L.16 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.

L.17 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/

L.18 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.

L.19 Mental Health and Wellness

If you find yourself beginning to feel some stress, anxiety and/or feeling slightly overwhelmed, try WellTrack (https://purdue.welltrack.com). Sign in and find information and tools at your fingertips, available to you 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
L. Syllabus, policies, and procedures

L.20. 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/.

No one on the teaching staff is qualified to make any kind of diagnosis, and we rely on medical professionals to document cases of illness and provide us with instructions when applicable. We have recourse policies in place for documented illness.

L.21 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.
L.22 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.

L.23 Changes to the syllabus

This syllabus is subject to change and changes will be announced appropriately.