Contents

Chapter 1

Preamble and Syllabus

1.1 About These Notes

This document collects course materials from the Summer 2024 offering of Stanford CS 161: Design and Analysis of Algorithms. Many different materials are included:

- The syllabus for the course.
- Daily lecture notes that I prepared for my lectures (these exclude handwritten figures that I prepared for whiteboarding; if you'd like access to these, contact me).
- Post-lecture quiz questions (graded for completion) that spot-check student understanding of the content covered that day.
- Post-lecture notes that I posted after almost every lecture for students who were interested in learning more about the material (we didn't test on any of these post-lecture notes).
- Homework assignments.
- Study guides we posted for each of the three midterm exams.
- An appendix containing material we didn't give to the students, but would if we taught the class again. Currently this consists of a crash-course defining the logarithm and a number of important facts (such as the asymptotic value of H_n and the limit of $\left(\frac{n-1}{n}\right)^n$) that showed up more frequently in the course than I originally expected them to. Notably, most of these facts can be proved in a surprisingly rigorous way even without the explicit use of calculus, if the proper definition of the logarithm is used.
- C source code for almost all of the algorithms described in this class is available at the following link: `https://lair.masot.net/git/161-code.git/`. (This code is designed to follow the pseudocode as closely as possible; these should not be used as reference production implementations.)

1.2 Copyright

1.3 About Textbooks

These notes are not intended to replace a textbook, but they are sufficient for a reasonably complete introduction to algorithms. One major benefit of lecture notes as a supplement to a textbook is the selection of topics, in addition to homeworks (many of the homework questions are new).

There is a discussion of relevant textbooks in the syllabus, which tries to give a balanced view on their pros and cons. But, after teaching this course, you should know the following fact: **Sedgewick and Wayne is the best introductory algorithms textbook.** CLRS has more breadth, but the explanations and motivations for the algorithms in CLRS are nowhere near the quality of SW. Knuth was invaluable to me in preparing these lecture notes, but proved itself basically unreadable for most of our introductory algorithms students. Erickson has far too sparse coverage (especially a complete lack of search data structures).

1.4 Idiosyncrasies and Mistakes

I tried to keep the lecture content accurate and use standard terminology in the field. While I think we mostly succeeded in these two goals (especially thanks to some very adept bug-finders in the class!), there were some choices that we insisted on that may break from standard algorithm classes and terminology:

- I tried to consistently differentiate between "average-case time" (expected value over a random distribution of *inputs* to the algorithm) and "expected time" (expected value over a random distribution of *choices* made by the algorithm, e.g., pivot or hash function). This confused the students, and I don't think it's as standard a distinction as I originally thought it was.
- Following the literature, CLRS gives multiple definitions for 'amortized time' (e.g., 'aggregate time' vs. 'operation time plus change in potential'). We standardized on one definition (the 'aggregate time') and treat the potential method as a lemma that helps you prove things about the aggregate time.
- We don't teach the master theorem, as it's neither insightful (vs. the recurrence tree method) nor useful at all for most of the truly interesting recurrences we reason about in this course (e.g., quicksort expected time or average-case BST insertion).
- Regarding dynamic programming, we heavily focus on the top-down approach (although the bottom-up approach is discussed briefly).
- We tried to make a distinction between "black-box" and "white-box" worst-case times for, e.g., hash tables. I'm not sure the terminology we landed on was very standard, and it didn't seem to help the students.

1.5 Contacting Me

I am certain that the notes have many mistakes, both big and small. If you find mistakes in these notes, I would appreciate a short email (`matthew@masot.net`) so I can correct them.

Many of the homework questions have extended hints available, but we did not provide them in these collected notes. Most students found the questions were only approachable with the hints. If you would like a hint for any question, I'm happy to engage in (brief) email discussions about any of the problems — please start by sending me what you've tried so far.

1.6 Acknowledgements

Thanks to many students in 161 for helping to debug these notes! Happy to add your names here if you reach out, but I've kept it anonymous out of privacy concerns until then.

1.7 Syllabus

Course Logistics

Lectures:	MWF 10:30 AM – 12:15 PM, Skilling Auditorium
Prerequisites:	106B or 106X; 103 or 103B; 109 or STATS 116
Website:	`https://cs161.stanford.edu`
Discussion Forum:	[closed]
Office Hours:	Link to calendar on website
Instructor:	Matthew Sotoudeh, Gates 478 (extra OHs by appointment—email me!)
Email:	sotoudeh@stanford.edu
Course Assistants:	Havin Hosgur, Nolan Miranda (he/him/his), Anjiang Wei

About the Course

Informally, an *algorithm* is a precise description of a method to accomplish a particular goal. In this class, we will ask (and answer!) "*what is the best way to accomplish X?*" for many different goals X. I love the study of algorithms because it is a unique blend of practically useful, theoretically insightful, and fun! I hope by the end of this quarter you'll agree.

My learning goals for you after taking this course include:

- **Speak the lingo:** be able to converse comfortably with other computer scientists about "table stakes" concepts such as Dijkstra's algorithm, balanced trees, and quicksort.
- **Predict behavior:** given an algorithm and a setting, be able to analyze the algorithm's best, average, expected, and worst-case performance characteristics.
- **Solve problems:** given a problem and setting, be able to design and implement an algorithm to solve that problem efficiently and effectively.
- **Communicate your thoughts:** convince others of the efficiency and efficacy of algorithms.

Many of these can be summarized: *"Think critically about what the best tool for the job is, and communicate those thoughts to others."*

Textbooks

This course does not have a required textbook, but there are a number of very good algorithms textbooks available. These include:

- Robert Sedgewick and Kevin Wayne; *Algorithms* (SW)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein; *Algorithms* (CLRS)
- Donald E. Knuth; *The Art of Computer Programming* (TAOCP)
- Jeff Erickson; *Algorithms* (JE)

I suggest you purchase at least one of them and use it as a reference throughout the course. If you are new to algorithms and programming, I would suggest either CLRS or SW. If you already know what mergesort and red-black trees are, I would suggest picking up volume 3 (and maybe 1) of TAOCP. CLRS and TAOCP are available for free online from Stanford Libraries. JE is available for free online from the author. All four books have been placed on course reserves at the Terman Engineering Library.

Homeworks

Homework problems will be handed out after each Monday lecture and uploaded to Canvas. By 11:59 PM Pacific time on the Tuesday of the following week, you must turn in a LaTeX-generated document with the following contents:

1. How did lecture go for you this week (too fast, too slow)? How did you watch lectures (in-person, 2x video, etc.)? What topics did you feel confident in, and which did you feel you need more practice with?

2. How did the homework go for you this week? Which problems were harder vs. easier?

3. At least three written-up answers to homework questions that you want our feedback on (doesn't have to be correct, or complete). The homework problems will be grouped by lecture and you must submit an attempt for at least one question relating to each lecture (hence, at least 3 problems).

As long as you make an earnest attempt at the homework and turn in this weekly update document, you will get 100% on the homework portion of the grade. If you feel super confident in the material and don't want to spend too much time on the homeworks, we're giving you that flexibility. **But beware!** It is very easy to overestimate your familiarity with the material, and homeworks are your best opportunity to calibrate your understanding with our expectations before the exams. So I *strongly* suggest spending lots of time on the homework, both for your own learning and so that you can get feedback from us to better align your expectations with ours.

Exams

We will have 3 quizzes and one final. The quizzes will focus on material from the preceeding two weeks, but everything covered so far in the class is fair game. The quizzes will be long, around 10 questions, with the goal of reducing the impact of any one question on your overall grade. The final exam is cumulative. Non-SCPD students must take the exams (including quizzes) in-person.

Grading

The breakdown is shown below. We will never curve grades down; if you get a 93% according to the below breakdown, you will get an A in the course. However, we will happily curve grades up, e.g., if the class average is low.

Homework:	15%
Quizzes ($3 \times 20\%$ each):	60%
Final:	25%
Lecture Participation:	(free points on the next quiz/exam score)
Total:	100%

'Lecture participation' includes a practice quiz question given and turned in at the end of most lectures. It will not be graded for correctness, but we will provide solutions. The dual goals for these practice questions are (i) to give us some insight into how comfortable people were with the material that day, and (ii) to give you some calibration for how well you understood the material. Students who regularly participate in these practice quizzes will get a few free points on the next 'real' exam (quiz or final). This is partially meant to incentivize in-person attendance; non-SCPD students must turn it in in-person and SCPD students must submit it to Gradescope at the end of class to get the points.

Course Policies, Expectations, and Guidelines

- Auditors are welcome at lectures, but due to the very limited course staff please understand that we cannot take OH or grading time away from enrolled students.

- SCPD students are welcome and encouraged to attend anything they want (lectures, OHs, quizzes, final exam) in person.

- If you must use a laptop or other electronic device during lecture, please consider sitting towards the back of the lecture hall so it does not distract those behind you.

- Please, please, please ask questions! This is a class of over 100 people and the lecture notes are brand new. If you are confused about something, I guarantee that many others are as well. I understand that many students are concerned about slowing down lecture too much. **Don't be:** keeping lecture on track is my job. If we don't have time to answer your question on-the-spot, I will let you know, continue lecturing, and post the answer on Ed afterwards.

- **Regrade requests are for when the grader made a factually incorrect comment, not for when you disagree with the amount of partial credit given to a wrong answer.** Regrade requests will be open for a period of one week after each assignment; no regrade requests will be accepted after that window closes. If you disagree with a regrade decision, you may bring it to me during OHs and I will regrade the entire assignment with you (possibly lowering the overall score).

Office Hours

You are encouraged to come to office hours for any questions about the course material (and beyond!). An office hour calendar will be posted on the website. We will use QueueStatus to manage the office hours queue: `https://queuestatus.com/queues/2775` so please sign in there when you arrive.

Honor Code

The Stanford Honor Code and CS Honor Code apply to this class. For the homeworks you may use any resource you like (including other students), although rules about plagiarism still apply (you *must* properly cite sources used). For the exams, no assistance is permitted (including other people, websites, notes, chatbots) unless explicitly approved by the instructor. If you have a question about these policies, please contact the instructor.

SCPD Recording Notice

Video cameras located in the back of the room will capture the instructor presentations in this course. For your convenience, you can access these recordings by logging into the course Canvas site. These recordings might be reused in other Stanford courses, viewed by other Stanford students, faculty, or staff, or used for other education and research purposes. Note that while the cameras are positioned with the intention of recording only the instructor, occasionally a part of your image or voice might be incidentally captured. If you have questions, please contact a member of the teaching team.

Access and Accommodations

Stanford is committed to providing equal educational opportunities for disabled students. Disabled students are a valued and essential part of the Stanford community. We welcome you to our class.

If you experience disability, please register with the Office of Accessible Education (OAE). Professional staff will evaluate your needs, support appropriate and reasonable accommodations, and prepare an Academic Accommodation Letter for faculty. To get started, or to re-initiate services, please visit oae.stanford.edu.

If you already have an Academic Accommodation Letter, we invite you to share your letter with us. Academic Accommodation Letters must be shared at the earliest possible opportunity so we may partner with you and OAE to identify any barriers to access and inclusion that might be encountered in your experience of this course.

Tentative Course Schedule

Quiz dates on the below calendar are fixed, but we may adjust the topics covered on other days as we go along.

Day	Topic(s) and/or deadlines
M, June 24st	Introduction to CS 161. Analysis of selection sort. Example of correctness proofs and best+worst case time analyses.
W, June 26th	Big-O, insertion sort, and average case analysis.
F, June 28th	Divide-and-conquer and merge sort. Examples of analyzing recurrences.
M, July 1st	Quicksort and quickselect. Examples of *expected* time analysis.
T, July 2nd	(Week 1 Homework Due)
W, July 3rd	Heaps and heapsort.
F, July 5th	Comparison sorting lower bounds. New problems: dynamic sorting and search. Then, BSTs and the fundamental BST operations: best, average, and worst-case analysis.
M, July 8th	Quiz 1: Sorting
T, July 9th	(Week 2 Homework Due)
W, July 10th	AVL trees: analysis and insertion.
F, July 12th	2–3 trees and red–black trees: analysis and insertion.
M, July 15th	Amortized analysis: aggregate analysis, the potential method, and dynamic arrays
T, July 16th	(Week 3 Homework Due)
W, July 17th	Splay trees and union-find (algorithms in class, some analyses on homework)
F, July 19th	Start hashing.
M, July 22nd	Hash maps: handling collisions.
T, July 23rd	(Week 4 Homework Due)
W, July 24th	Review Session
F, July 26th	Quiz 2: Searching (trees and hashing)
M, July 29th	Graphs, graph representations, and graph parameters (e.g., directedness, sparsity, acyclicity). Graph reachability: DFS and BFS. Connected components on static, undirected graphs. Topological sort.
T, July 30th	(Week 5 Homework Due)
W, July 31st	Shortest paths: Dijkstra, Bellman-Ford, Floyd-Warshall. Dynamic programming: fibonacci.
F, August 2nd	Spanning trees: Prim and Kruskal. Discussion of greedy algorithms more generally.
M, August 5th	Max flow/min cut
T, August 6th	(Week 6 Homework Due)
W, August 7th	More examples of dynamic programming and greedy algorithms. Hopefully knapsack, independent set, Floyd-Warshall in more detail, etc.
F, August 9th	Quiz 3: Graphs
M, August 12th	Other things you should know exist, but we will not test on the final: (subset of?) linear programming, smoothed analysis, MILP, multiplication algorithms (numbers/matrices/polynomials), FFT, CDCL, local search, string matching, primality testing, prime number generators, stable marriage, A^*, optimizing submodular functions.
T, August 13th	(Week 7 Homework Due)
W, August 14th	Review for Final Exam
Sa, August 17	Final Exam

Chapter 2

Sorting

2.1 Lecture 6/24 Notes

2.1.1 Short Intro

Definition 1. *"Algorithm" (informally): a precise description of a method to solve a particular problem.*

Examples of algorithms:

- When I ask Axess to sort my list of students by name, it uses some particular algorithm under the hood to do that.
- Your computer's operating system has an algorithm (the "scheduler") to determine which program will get the nice slice of CPU time.
- Google Maps has an algorithm to determine the best way to get from my apartment to McDonald's.

(Underline the problem in each one.)

This class will focus on the *study of algorithms:* given a problem, think about many different algorithms for solving it and try to determine which is best (or what the different tradeoffs are). If you think about it, this is basically the core of computer science!

Algorithms is a unique blend of

1. Practical: using the right algorithm can save you tons of money and time
2. Philosophical: tells us something about the world that one approach is truly better than another

About to dive in; a few things to touch on before we do so:

- Cadence of the class is generally very predictable: we'll introduce a problem, then we'll introduce an algorithm meant to solve the problem, then we'll spend time analyzing how good it is. Problem, solution, analysis. Problem, solution, analysis.
- Please, please, please ask questions! Keeping the lecture on pace is my responsibility, not yours. In return, I just ask that you be OK with me responding to some questions with a "let's talk offline" or "I'll post an extended answer to Ed." Not looking for "smart" questions, or insightful answers to think-pair-shares, etc. — just looking for engagement. Some bonus points for folks who engage, especially with the think-pair-shares. (Insert "Question Asking" between each of the three lecture steps above!)

2.1.2 Problem of the week: sorting!

Definition 2. *Given a list of objects, and some way to compare them, the* sorting *problem is to rearrange the objects so they are in ascending order.*

Examples:

- Numbers: simplest is sorting a list of numbers; $11, 100, 60, 70$ becomes $11, 60, 70, 100$.
- Words: sort lexicographically, like you're making a dictionary ("work, word, worm" becomes "word, work, worm")
- Can also sort rows in a table; Excel does this. E.g., students where the comparison compares grade. Note here it's a bit tricky: we're sorting the rows, but the comparison only takes into account grade.
- Can sort search results by how close they are to the user's query.

Big point: many different meanings to "comparison", not all of which are numeric! For the first set of sorting algorithms we see, the only thing we need is that there's some way to 'compare' two elements, i.e., ask which is smaller. E.g., the user of our sorting algorithm must provide a way to test whether "word ¡ work".

But if it helps, for now you can just assume we're always sorting numbers.

We're also being a bit vague about how the objects are stored: linked list? array? ?? For now, assume everything is stored in an array.

Why do we care about sorting?

At least three reasons:

- Practical: sorting is incredibly important in modern computer systems:

Project	Rough underapprox of sort calls
Pytorch	607
VSCode	764
LibreOffice (basically Office/Drive)	434
Chrome	1037

- Historical (`https://www.census.gov/history/www/innovations/technology/the_hollerith_tabulator.html`): In 1888 the Census bureau held a competition to see who could create a machine to process and sort their records the fastest. Herman Hollerith invented a machine based on punchcards and won: data that took other contests over 40 hours to sort took his machine just over 5 hours! His punchcard machine was one of the earliest ancestors of modern computers, and Hollerith later went on to create the predecessor of IBM.

  Ever since, sorting has been a core problem to computing machines; in fact, one of the first programs to be written for the first stored-program computer was a sorting routine!

  So studying sorting algorithms puts you 'in touch' with the core of our field.

- Pedagogical: sorting is a good example problem. There are many different algorithms with very different tradeoffs. The techniques needed to analyzing sorting algorithms will be useful for all the other algorithms we see in this class.

Think-Pair-Share

Now that we understand the goal, let's try to develop some sorting algorithms. Introduce yourself to your neighbor then try to come up with the simplest way you can to sort these five toys(?) by height(?). Note we're looking for a general procedure, that will work in many different scenarios not just this example one. So "move the red one to the end" isn't general enough; you'll have to say why you chose the red one to move.

Pull it back together, ask for ideas, record them on the board and try to link them to 'real' algorithms. Give praise to each one, e.g., selection sort (depending on the approach) leads to one of the best algos.

Maybe ask folks to guess which they think would be fastest? Point out it's not obvious, and implementing them is probably not super easy either.

Selection Sort

The first we'll study in detail is selection sort. Why? It is (to me) the most natural! And has some fun and pedagogically instructive problems that show up in its analysis.

If we were to write up the steps, it looks like:

1. Find the smallest element
2. Move it to the front
3. Repeat among the remaining items until none left

Example: with the physical objects.

And in more explicit pseudocode:

```
def selection_sort(A):
    i = 0
    while i < N: # 0, 1, ..., N-1
        # find the min among [i, N)
        m = i
        j = i + 1
        while j < N: # i+1, i+2, ..., N-1
            if A[m] > A[j]:
                m = j
            j = j + 1
        # swap it with A[i]
        swap A[i] <-> A[m]
        # continue on to the next element
        i = i + 1
    return A
```

Example: with $A = [17, 15, 9]$.

Talk briefly about how they might have invented this algorithm; one of the key ideas is "get one step closer to your goal on every iteration."

Analyzing Algorithms Now that we have an algorithm, we need to analyze it. What does that mean? It means asking and answering questions about its behavior!

There are at least two questions we can essentially always ask:

1. Is it *correct*?
2. Is it *efficient*?

There are many reasonable ways you could go about answering these questions. We usually think of two big attack directions:

1. The *empirical* approach: implement and test the proposed algorithm. See if it seems to work and seems to be fast.
2. The *analytical* approach: write a rigorous mathematical *proof* that the algorithm is correct, or meets some efficiency guarantee, etc.

Think-pair-share: where have you seen these two approaches in your prior coursework? What are some arguments for or against each one?

Example arguments:

- For empirical: might be easier. can run on real data, which might change the results. can run on the real machine, which has very difficult/intricate performance interactions.

- For analytical: could actually be easier for more complicated algorithms! can analyze *corner case* behavior that would only show up once pushed to prod for empirical approach.

Critical to have both tools in your toolbox, this class focuses only on the analytical approach because that's essentially the point of this class in the curriculum at Stanford (many other classes teach you how to use the empirical approach!).

The P Word The analytical method goes hand-in-hand with *proofs*. Just a rigorous argument that would be convincing to a skeptical reader. Some English essay-like subjectivity in grading. Sticking to the PWC is a good idea but not necessary for this class; think of it like the 5-para essay form.

Analyzing Selection Sort: Correctness Let's look at the first question: is selection sort *correct*, i.e., does it return a sorted version of the input list?

The answer is yes (unless I've made a typo!) but we need to *explain why* to a skeptical reader!

There are many different ways to prove algorithm correctness, but one is far-and-away the standard: *inductive loop invariants*. This is the standard we expect of rigorous correctness proofs in this class (though we won't ask for them too often: most commonly we'll ask you to *break* algorithms).

Two Rules of Correctness Proofs (in this class):

1. Your reader *can* understand and reason through the behavior of finitely sequences of basic operations on their own ...

2. But they *cannot* reason through the behavior of *loops* without your help.

That help should come in the form of a *loop invariant*, which we'll discuss next.

A *loop invariant* is just a statement that is true *every time the loop condition is checked*, and can be proved via induction on the number of loop iterations. The point of a loop invariant is to "summarize" the important behavior of the loop for the reader into one single claim. Key is that the reader only ever needs to reason about a single loop iteration at a time.

Theorem 1. *Selection sort correctly sorts any array (of numbers.)*

Proof. We have two loops, so we need two loop invariants.

Claim: every time the inner loop's condition is checked, m has the index of the minimal element among $A_i, \ldots, A_{j-1}$.

Proof by induction (have better diagrams in the handdrawn notes):

1. When the condition is first checked, we have $m = i, j = i + 1$ so the claim corresponds to A_i being minimal among $A_i, \ldots, A_i$, which is always true.

2. For the inductive case, we assume it's true at some point during execution, then need to show it's still true after one more loop iteration. Let i, j, m, A be the values of those variables right at the start of the iteration, and i', j', m', A' the values after the iteration. Notice $i' = i, A' = A, j' = j + 1$. The IH tells us m is minimal among $A_i, \ldots, A_{j-1}$. We must show that, after one more loop iteration, m' is minimal among $A'_{i'}, \ldots, A'_{j'-1}$, or in other words, m' is minimal among $A_i, \ldots, A_j$.. During the loop iteration, exactly one of two things could happen:

   - If $A_m > A_j$, then we set $m' = j$.
   - Otherwise, we must have $A_m \leq A_j$ and so we set $m' = m$.

   In the first case, $A_{m'} = A_j$ is smaller than A_m, which was (by IH) the smallest among $A_i, \ldots, A_{j-1}$. Hence, $A_{m'}$ is the smallest among $A_i, \ldots, A_{j-1}, A_j$, as desired.

   In the second case, $A_{m'} = A_m$ is smaller than A_j, and by IH is smaller than everything else in $A_i, \ldots, A_{j-1}$, hence $A_{m'}$ is smallest among $A_i, \ldots, A_j$, as desired.

(Notice after the very last iteration this implies when the loop ends it is minimal among $i, \ldots, N-1$. So, now the reader can ignore all the details of the inner loop and just summarize it as "the loop makes m the minimal element among $i, \ldots, N-1$.")

Claim: any time the loop condition on line 3 is checked, the indices $0, 1, \ldots, i-1$ contain the smallest i elements in sorted order.

Proof by induction (better pictures in handdrawn notes):

1. Vacuously true for the first time it's reached.

2. At the start of the loop we know $A_0, A_1, \ldots, A_{i-1}$ are the smallest i sorted; we need to check what happens after one more iteration. Well, we found the smallest thing among the remaining items (everything except for the i smallest), so it must be the $i' = (i+1)$th smallest element in A. We placed that in slot A'_i. Hence, now the i' smallest elements are in their proper places $A'_0, \ldots, A'_{i'-1}$.

(So now consider what happens when the outer loop ends: guarantees $0, 1, \ldots, N-1$ are sorted!)

(Note this is actually nto quite enough; we need to also argue that the resulting list is 'just' a rearrangement of the original list, i.e., we didn't just overwrite the whole thing with new, unrelated values. But this can be seen because the only operation we ever do to A is swapping values, hence the list always remains just a rearrangement of the original.) □

Take a step back to let the loop invariant/loop summary process sink in. It's a bit formal and tiring. We won't spend much time in lecture doing this, but we do want you to practice it. Key point is to be able to summarize for the reader the critical behavior of each loop in one statement.

Analyzing Selection Sort: Efficiency Efficiency is something we'll spend a lot more time thinking about in lectures.

The first question you often want to answer about efficiency is: how much time does this program take? It's impossible to answer this question without knowing the machine it will be run on and the inputs it will be run on. For now, as a rough estimate, we'll analyze the algorithm assuming every time a line is executed it costs 1 unit of time. We'll also look at the behavior on the *best* possible input, the *worst* possible input, and *average* (randomly chosen) inputs.

In theory, you can also use loop invariants to rigorously prove things about how much time a program takes, but we'll be a little less formal than that in this class.

The basic approach we'll use is to look at each line and count up how many times it's executed.

Think-pair-share: in terms of N, the length of the array, how many times total is each line of the selection sort pseudocode executed? Try to fill in this table:

Line	Count	Count (Asymptotic)
2	1	$\Theta(1)$
3	$N+1$	$\Theta(N)$
5	N	$\Theta(N)$
6	N	$\Theta(N)$
7	$N + (N-1) + \cdots + 1 = \frac{1}{2}N(N+1)$	$\Theta(N^2)$
8	$(N-1) + (N-2) + \cdots + 1 = \frac{1}{2}N(N-1)$	$\Theta(N^2)$
9	$0 \le T_9 \le \frac{1}{2}N(N-1)$	$O(N^2)$
10	$\frac{1}{2}N(N-1)$	$\Theta(N^2)$
12	N	$\Theta(N)$
14	N	$\Theta(N)$
Total (computed)	$1.5N^2 + 4.5N + 2 \le T \le 2N^2 + 4N + 2$	$\Theta(N^2)$

Line 9 is hard to get an exact count for: clearly it can't be executed fewer than 0 times or more than line 8 is.

So, without more analysis (we'll do it on the HW!), we can only get a rough bound. But that rough bound is pretty useful: it tells us that no matter what, we're going to be performing on the order of n^2 operations.

How much is that? Assuming each line takes 10^{-9} seconds to run...

N	Time to execute N^2 lines at 10^9 lines per second
100	10^{-5} seconds
1,000	0.001 seconds
1,000,000	$>$ 1000 seconds
3,000,000,000 (Facebook MAU)	$>$ 285 years

Hmm, not too great!

Side Quest on Asymptotic Notation **Think-pair-share:** what are some pros and cons of asymptotic notation compared to the exact counting?

Some possibilities:

- Asymptotic analysis: it's easier to work with, it makes it easier to compare algorithms based on what happens for big inputs, and it reflects our uncertainty about the exact constant running time of each line/operation.
- Exact analysis: often constant factors *do* matter, especially when comparing two algorithms with the same asymptotics. on real computers constant factors matter *a lot*. even though we're uncertain about the constant running times of operations, hypothetically once we fix a machine we can integrate that info into our analysis. Gives us more insight into how variable/unpredictable the running time is (i.e., difference between best and worst might be 10×, or it might be 1×).

In this class, and in real life, we'll mostly use asymptotic notation. But it is still important to be able to perform a more precise analysis when necessary. But you should probably expect it to be like 90%–10% in favor of asymptotic analysis for this class.

2.1.3 Syllabus and Logistics

Pass out syllabi, talk through it. Do some roadmapping based on the calendar. Take some questions. Plan for ≈15m.

2.1.4 Wrap Up for Today

So we've seen:

- Using loop invariants to explain why an algorithm is correct
- Best and worst case analysis of algorithm efficiency

Now, let's go on to the practice quiz!

2.1.5 Post-Lecture Ed Notes

There were many interesting questions today, one of which was why Knuth uses "go to"s throughout his textbooks.

If you also found this question interesting, note that he writes at length about 'go to' in the following paper: `https://dl.acm.org/doi/pdf/10.1145/356635.356640` and specifically about the use in his textbooks on page 272 (PDF page 12), section title "A Confession."

Enjoy!

Name: _____ **Stanford ID Number:** _____

2.1.6 Lecture 6/24 Practice Quiz Question (Extra Credit for Completion)

Recall the selection sort pseudocode from lecture (note the line numbers may have changed!):

```
1  def selection_sort(A, N): # list A of length N
2      i = 0
3      while i < N: # 0, 1, ..., N-1
4          # find the min among [i, N)
5          m = i
6          j = i + 1
7          while j < N: # i+1, i+2, ..., N-1
8              if A[m] > A[j]:
9                  m = j
10             j = j + 1
11         # swap it with A[i]
12         swap A[i] <-> A[m]
13         # continue on to the next element
14         i = i + 1
15     return A
```

For each of the following proposed modifications, answer with brief justification:

- Does the resulting algorithm still correctly sort?
- **If so,** describe how much running time the modification saves in terms of N, the size of the array. Assume every line takes one unit of time every time it executes. A tight asymptotic answer (e.g., of the form $\Theta(n)$) is OK.
- **If not,** provide a small (at most 4 objects) example that it fails to sort correctly.

1. **Change the loop condition on line 3 to** $i < N - 1$

   (a) Does it still sort?

   (b) Time improvement or failing input:

Part (2) on back →

2. **Change the loop condition on line 3 to** $i < N - 2$

   (a) Does it still sort?

   (b) Time improvement or failing input:

2.2 Lecture 6/26 Notes

2.2.1 Big-O Bootcamp

Going to spend a few minutes at the start talking about what big-O is.

The most important thing to know is that big-O really has no direct relationship to programming, or to algorithms. It's just a convenient way of talking about upper and lower bounds on functions.

It turns out that we talk about upper and lower bounds on functions *a lot* when doing algorithms (namely, where those functions are the running time of our programs). But the actual definition says nothing about programming.

Definition 3. *For any natural-valued function $f(n)$ we can define the set of functions $O(f(n))$ like so:*

$$O(f(n)) = \{f_0(n) \mid \exists C_{f_0}, N_{f_0}, \forall n \geq N_{f_0}, \quad 0 \leq f_0(n) \leq C_{f_0} f(n)\}.$$

In words, $O(f(n))$ means "the set of functions that are eventually upper-bounded by a constant multiple of $f(n)$." It turns out big-O notation is very good at capturing what we mean by things like "the dominant term," e.g., we get results like $n^2 + 2n + 5 \in O(n^2)$. This is the sort of thing you want to play around with the definition a lot to let it sink in.

But with this definition, you might be thinking, then the sort of stuff we wrote last week like:

$$n + 1 = O(n)$$

really makes no sense at all! And you would be right. But it's what computer scientists do anyways.

In particular, we have the following:

Convention 1. *When we have an equation with asymptotic notation on either (or both) sides of the equals sign, we mean the following: if every asymptotic set on the left hand side is replaced by* any *of its members, then there* exists *a member of each asymptotic set on the right-hand side that can replace it and make the equation true.*

Hence, for example,

$$O(f(n)) + O(g(n)) + h(n) = O(k(n))$$

means, for any *choice of $f_0(n) \in O(f(n))$ and for* any *choice of $g_0(n) \in O(g(n))$ there* exists *a choice of $k_0(n) \in O(k(n))$ such that*

$$f_0(n) + g_0(n) + h(n) = k_0(n).$$

Definitions for the related notions of Ω, Θ, ω, and o are available in the textbook. For this class, I think we'll only need O, Ω, and Θ; you don't need to learn the remaining ones. *Informally*, you can think of the following analogies:

1. $f_0(n) = O(f(n))$ is like $f_0(n) \leq f(n)$,
2. $f_0(n) = \Omega(f(n))$ is like $f_0(n) \geq f(n)$, and
3. $f_0(n) = \Theta(f(n))$ is like $f_0(n) = f(n)$,

but of course, like most analogies, it's not precisely accurate.

2.2.2 Recall Selection Sort

Selection sort:

1. Find smallest
2. Swap it with index 0
3. Repeat among the remaining items

It was pretty slow: $\Theta(n^2)$ *regardless* of the input.

Remember in this class we always want to ask: *is it possible to do better??*

Today we'll see the answer is yes!

In fact we'll see our first two "beautiful" algorithms: insertion sort and merge sort. Insertion sort will be beautiful because of how well it adapts to and makes use of the already sortedness of some arrays, while merge sort will be beautiful because of how well it does *regardless* of how well sorted the array starts out!

Insertion Sort

How many have played a card game in their life?

Insertion sort is known as the *playing card sort*, because it's how card players often sort their hands.

Let's demo it: as you pull each card from the deck, you look for its proper place and insert it there.

In pseudocode:

```
1 def insertion_sort(A, N):
2     i = 0
3     while i < N:
4         j = i
5         while j > 0 and A[j] < A[j-1]:
6             swap A[j] <-> A[j-1]
7             j = j - 1
8         i = i + 1
9     return A
```

Let's walk through an execution on the list $5, 17, 1, 20$.

Think-pair-share: Last week we saw that the *final* iteration of selection sort's loop was not really necessay. What about insert sort? Are any iterations unnecessary?

Analyzing Insertion Sort: Correctness

Theorem 2. *Insertion sort properly sorts an array of numbers.*

From here out we'll only give the invariants, not the explicit inductive proofs.

Proof. The outer loop maintains the invariant that, every time line 3 is reached, the subarray $A_0, \ldots, A_{i-1}$ is sorted.

To prove this, you need the following invariant for the inner loop: every time line 5 is reached, the following subarrays are all sorted (separately):

- $A_0, \ldots, A_{j-1}, A_{j+1}, \ldots, A_i$, and
- $A_j, \ldots, A_i$.

(You also need that it's a rearrangement of the original items; here again that comes "for free" because we only ever swap entries in the array.) □

Analyzing Insertion Sort: Efficiency For selection sort, no matter what the input looked like it took about n^2 operations.

Let's do an analysis to see if that's true for insertion sort as well.

Line	Number of Times Executed (Asymptotic)
2	$\Theta(1)$
3, 4, 8	$\Theta(n)$
5	$\Omega(n)$, $O(n^2)$
6, 7	$\Omega(0)$, $O(n^2)$
Total	$\Omega(n)$, $O(n^2)$

Just like last week, we see that some of the lines seem to be hard to count without knowing more about the input. Unlike last week, *that actually makes a difference*: our lower and upper bounds are very far away from each other!

Note: so far, we have no way to tell whether our counting was just too loose (too many approximations) or if the running time of the algorithm really *does* vary greatly depending on the exact input array used. So let's do a **think-pair-share**: Can you come up with any types of inputs where it takes $O(n)$ time? What about $O(n^2)$ time?

Hopefully folks find the following examples:

1. On $1, 2, \ldots, n$, it runs in $\Theta(n)$ time
2. On $n, n-1, \ldots, 1$, it runs in $\Theta(n^2)$ time

So now we know for sure: the running time of this algorithm fluctuates between $\Theta(n)$ and $\Theta(n^2)$ depending on the exact input provided! This is an instance of a very common thing with algorithm analysis: *the efficiency of an algorithm often depends on the input you provide it with.*

Leads to the theorem:

Theorem 3. *The* best-case *time of insertion sort is* $\Theta(n)$*, the* worst-case *time of insertion sort is* $\Theta(n^2)$*.*

But neither of these is a very useful analysis; it could be that both the best and worst case are encountered very infrequently. What we really want is an *average case* analysis.

Definition 4. *The* average case *running time of an algorithm with respect to a distribution is the expected value of the time it takes to run the algorithm on an input chosen the distribution.*

In the context of sorting, we often consider the distribution of inputs to be all the $n!$ permutations of $\{0, 1, \ldots, n-1\}$ with each permutation having equal probability $\frac{1}{n!}$. Another way to look at it in the context of sorting is to ask what the average running time of the algorithm would be on a *randomly shuffled* list of distinct items.

Theorem 4. *The average running time of insertion sort on an input of n distinct items is $\Theta(n^2)$.*

Proof. (This theorem has a beautiful proof in terms of inversions that you'll be able to reconstruct after doing the homework, but for now let's see a more direct proof.)

We know that every line is executed $\Theta(N)$ times *except for* the inner loop, so let's analyze how many times the inner loop iterates.

Every iteration of the inner loop moves the item that was originally at A_i one entry to the left until it finds it proper sorted spot among the first i entries. But all orderings were equally possible, so this entry has a $\frac{1}{i+1}$ chance of getting place at any index $0, 1, 2, \ldots, i$. So the expected number of inner loop iterations is:

$$\frac{1}{i+1}(0 + 1 + 2 + \cdots + i) = \frac{i(i+1)}{2(i+1)} = \frac{1}{2}i,$$

and so the total expected time spent on those lines is

$$\sum_{i=0}^{n-1} \frac{1}{2}i = \Theta(n^2).$$

Hence the total average case running time is

$$\Theta(n) + \Theta(n^2) = \Theta(n^2).$$

□

Not so good! Though this is sort of to be expected for an algorithm that makes use of already sortedness: random arrays are by definition very unsorted!

So how to we formalize the idea that it "takes advantage of the sortedness?" On the homework you'll see a characterization in terms of *inversions*, a very natural measure of how sorted or unsorted an array is. Briefly: pick two items at random, and check if they're out of order. The probability that this happens is directly correlated to the running time of insertion sort (draw picture).

Bigger Picture on Insertion Sort

Insertion sort is interesting for at least three reasons:

1. Performs *very* well when the list is already sorted, or near-sorted (the other algorithms we'll see don't have this property!).
2. Requires extremely little code to implement
3. A simple (but hard to analyze!) modification makes it into *Shell sort*: instead of performing insertion sort on the whole list, sort with a stride of $2^k - 1$ for decreasing k. Lets entries 'teleport' further around the list. Shell sort is surprisingly efficient, and still very easy to implement. Quote SW comment about Shell sort.

Design Corner

Insertion sort is another great example of both being inspired by a "real-world" process (card sorting) and a great example of the "make it a little better each time" design patterns.

2.2.3 Wrap Up

So we've seen:

- Insertion sort and average case analysis
- Merge sort and solving recurrences (pushed to F)
- Divide-and-conquer (pushed to F)

Now, let's go on to the practice quiz!

Name: _____ **Stanford ID Number:** _____

2.2.4 Lecture 6/26 Practice Quiz Question

Here is the insertion sort code from lecture:

```
def insertion_sort(A, N):
    i = 0
    while i < N:
        j = i
        while j > 0 and A[j] < A[j-1]:
            swap A[j] <-> A[j-1]
            j = j - 1
        i = i + 1
    return A
```

Problem 1: Define for each n the *mountaintop list* $M_n = [1, 2, \ldots, n-2, n-1, n, n-1, n-2, \ldots, 1]$. What is the asymptotic running time of insertion sort on the mountaintop list M_n? Justify your answer.

2.3 Lecture 6/28 Notes

(Briefly mention Shell sort.)

2.3.1 Divide-and-Conquer

Today is our big "divide and conquer" day. So far, the algorithm design technique we've seen work for selection and insertion sort is the "make it a little better each time" technique. Today, we're going to see a different approach that often gives very good algorithms (tho, the MIALBET approach will still be useful in the algorithms we see today!). The new technique is called *divide-and-conquer*, and it goes hand-in-hand with recursion.

In divide-and-conquer, you imagine someone gives you an input of size n and asks you to solve problem X. But, crucially, you also imagine you have some magic box that solves problem X for all inputs *strictly smaller* than n. So your goal is to some how turn your one big input of size n into a few smaller inputs of size $< n$, stick them through the box, and then "stitch together" the solutions from those smaller problems into a solution for the bigger problem. Of course, the magic box isn't really magic: it just uses the same technique recursively to reduce the problem more and more until you get to an easy-to-solve base case! In other words, you should know how to implement the box for, say, < 1. And the MIALBET approach will still be helpful in implementing the parts on either side of the box.

2.3.2 Merge Sort

We've seen:

- Selection sort is $\Theta(n^2)$ on all inputs
- Insertion sort interpolates smoothly between $\Theta(n)$ on the best inputs and $\Theta(n^2)$ on the worst.

Our dream "Algorithm X" would have $\Theta(n)$ on *all* inputs. Is this possible? Unfortunately not in full generality; we'll understand why next week. But until then, it is possible to get *extremely close,* namely, $\Theta(n \log n)$! The first algorithm that we'll see that reaches this bound is called *merge sort.*

Merge sort is a divide-and-conquer algorithm. It splits up the input in half, pushing both halves through the magic box. Then it uses a special merge routine to merge the two halves together. So its basic form looks like:

1. Sort the left half of the array
2. Sort the right half of the array
3. "Merge" the now-sorted left and right halves

These algorithms will be slightly more complicated because many of the recursive calls are working on subarrays, not the entire array at once.

```
1 def merge_sort(A, l, h):
2     if (h - l) <= 1: return
3     m = l + (h - l)//2
4     merge_sort(A, l, m) # sort the left half of the list
5     merge_sort(A, m, h) # sort the right half of the list
6     merge(A, l, m, h) # merge the two halves together
7     return A
```

There are two "tricks" to this algorithm: first, the use of recursion. Second, the mysterious *merge* method that we call.

What does it do? *Given a list A where two adjacent regions are sorted, Merge will merge them into a single fully sorted region.*

How should we go about that? Do an example with cards! At each step, take the smallest card. But only have to look at two.

```
1  def merge(A, l, m, h):
2      # assume [l, m) sorted and [m, h) sorted
3      T = (scratch space of size >=N)
4      k, i, j = 0, l, m
5      while i < m and j < h:
6          if A[i] <= A[j]:
7              T[k++] = A[i++]
8          else:
9              T[k++] = A[j++]
10     while i < m:
11         T[k++] = A[i++]
12     while j < h:
13         T[k++] = A[j++]
14
15     i = 0
16     while i < h - l:
17         A[l + i] = T[i]
18         i = i + 1
```

2.3.3 Analysis of Merge

Theorem 5. *Merge correctly merges two sorted lists.*

Proof. The final loop copies T into A: one invariant is that $A_l, \ldots, A_{l+i-1}$ are identical to $T_0, \ldots, T_{i-1}$.

All the other loops maintain the invariant:

1. $T_0, \ldots, T_{k-1}$ contain the entries $A_l, \ldots, A_{i-1}$ and $A_m, \ldots, A_{j-1}$ in sorted order, and
2. Every one of $T_0, \ldots, T_{k-1}$ is smaller than or equal to every one of $A_i, \ldots, A_{m-1}$ and $A_j, \ldots, A_{h-1}$.
3. ($i \leq m$, $j \leq h$, $k = (i-l)+(j-m)$, and a few other "housekeeping" relations that aren't too interesting or important for you to keep track of.)

□

Merge is both surprisingly tricky and surprisingly easy to analyze the running time of. It's tricky, because we actually can't get tight bounds for many of the lines: pretty much every line can be executed between $\Omega(0)$ and $O(n)$ times! The trick is that there is some correlation between the line counts: if, say, line 7 is executed 0 times, then line 11 must be executed $\Theta(n)$ times. So overall it's still $\Theta(n)$.

Thankfully, we can also skip that whole analysis and use the fact that the very last loop runs in $\Theta(n)$ time, while everything else runs in $O(n)$ time, and $\Theta(n) + O(n) = \Theta(n)$ so we're good.

Theorem 6. *Letting $n = h - l$, Merge takes time $\Theta(n)$.*

Proof. The final loop takes time $\Theta(n)$ because it iterates over $i = 0, 1, \ldots, n-1$.

Every other loop iteration "eats up" exactly one thing from either the left half or the right half of the array, hence can run for at most n times before it runs out of things to "eat."

The lines outside of the loop iterations run at most once, hence $\Theta(n)$ in total. □

2.3.4 Analysis of Merge Sort

Theorem 7. *Merge sort is correct*

Proof. (For recursive methods we usually don't give explicit loop invariants, instead we'll just induct on the input size with the IH being that the program is correct for all smaller inputs. You can think of this as being a very simple loop invariant, where the "looping" is actually the recursive call.)

By induction on input size.

Base case: when A is empty, it correctly returns A.

Inductive case: by the IH we see that, before the merge call, $A_l, \ldots, A_{m-1}$ and $A_m, \ldots, A_{h-1}$ are both sorted. Hence, by the correctness of merge, after the merge call $A_l, \ldots, A_{h-1}$ are all sorted. □

How should we go about analyzing the runtime? We'll see a few different ways to approach this!

In all cases, we're going to assume $n = 2^k$, i.e., the input array is a power-of-two size. (This is a questionable assumption, and you should have questions about it. But it turns out to be OK in *most* cases, meaning, the resulting bound you find is *usually* true even if n is not a power of two. If we have extra time let me know at the end of class and I'll try to wing an explanation of one way to show that. The textbooks work through some different ways of proving this explicitly; we won't.)

Recursion Trees Approach to MergeSort Runtime

Let C be a constant so the work done on an input of size m is at most Cm.

Draw the recursion tree (will draw on whiteboard); imagine each node as a worker. The root worker only does Cn work (the merge). He offloads the recursive calls to two subworkers, each one given an input of size $n/2$.

Each one of those subworkers does only $C(n/2)$ work themself after farming two jobs of size $n/4$ to their subworkers.

This repeats; on level i from the root we have 2^i 'workers' personally doing $C(n/2^i)$ work each, for a total of

$$\sum_{w=1}^{2^i} C\frac{n}{2^i} = Cn$$

work being done by all the workers on level i together.

How many workers do we have? Well, it stops once $n/2^i = 1$, i.e., $n = 2^i$, i.e., $i = \log n$. So the total work done by all the workers across the $\log n$ levels is

$$\sum_{l=1}^{\log n} Cn = Cn \log n = \Theta(n \log n).$$

"Plug-and-Pattern-Match" Approach to MergeSort Runtime

Proof. (Alternate, more algebraic) Let $T(n)$ be the time taken by merge sort to sort n items and assume $n = 2^k$. Then we have the relation:

$$\begin{aligned} T(2^k) &\leq 2T(2^{k-1}) + C2^k \\ T(1) &\leq C \end{aligned}$$

where C is some constant.

Now, plugging in the inequality for $T(2^{k-1})$ into that for $T(2^k)$ and repeating this process gives us:

$$\begin{aligned} T(2^k) &\leq 2T(2^{k-1}) + C2^k \\ T(2^k) &\leq 2(2T(2^{k-2}) + C2^{k-1}) + C2^k \\ &= 2^2T(2^{k-2}) + 2(C2^k) \\ T(2^k) &\leq 2^2(2T(2^{k-3}) + C2^{k-2}) + 2(C2^k) \\ &= 2^3T(2^{k-3}) + 3(C2^k) \\ &\vdots \\ T(2^k) &\leq 2^jT(2^{k-j}) + j(C2^k) \\ T(2^k) &\leq 2^kT(2^0) + k(C2^k) \\ &\leq 2^kC + k(C2^k) = C(k+1)2^k \\ &= O(2^k k). \end{aligned}$$

Recall that $n = 2^k$ and $k = \log n$, hence

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

(Note that, pedantically, this only gives us $O(n \log n)$. But you could do the same argument by observing $T(2^k) \geq 2T(2^{k-1}) + D2^k$ for some constant D to get $\Theta(n \log n)$.) □

"Induction-from-the-Ansatz" Approach to MergeSort Runtime

When you already have a guess ("ansatz") for the runtime, you can often use induction to prove it more directly.

Proof. Same analysis as before, but now we prove via induction that:

$$T(2^k) \leq C(k+1)2^k.$$

Base case: for $k = 0$, it's true by definition.
Inductive case:

$$\begin{aligned} T(2^k) &\leq 2T(2^{k-1}) + C2^k \\ &\leq 2(Ck2^{k-1}) + C2^k \quad \text{(by IH)} \\ &= Ck2^k + C2^k \\ &= C(k+1)2^k. \end{aligned}$$

□

"The Master Theorem" Approach to MergeSort Runtime

There is a theorem (the "master theorem") that helps us automatically solve recurrence relations like this. You are welcome to memorize it and use it on tests + problem sheets, but we won't discuss it much in class.

Practical Uses of MergeSort

It has a very predictable runtime and is great for external sorting because you're accessing the two halves of the array sequentially (can read off from a tape quickly).

Design Corner

Merge sort is the first very clear example we're going to see for the *divide-and-conquer paradigm.* In divide-and-conquer, you find a way to break the problem down into smaller versions of the same problem (these will get solved by recursion) where you can quickly combine those subproblem solutions into a solution to the original problem (like "merge" does).

Question to ask yourself: if I could solve this problem for *any portion of the input independently*, can I stitch those solutions back into a solution for the whole input?

2.3.5 Sorts We Know So Far

1. Selection sort: $\Theta(N^2)$ in all cases
2. Insertion sort: $\Theta(N)$ best, $\Theta(N^2)$ average and worst
3. Merge sort: $\Theta(N \log N)$ in all cases, but it requires a lot of extra memory and data movement

Today we'll see one of the all-around best sorting algorithms: *quicksort.* Similar to merge sort, quicksort is a divide-and-conquer scheme. *Unlike* merge sort, it is *randomized.* We will see that quicksort has $\Theta(N \log N)$ behavior on all inputs with *very high probability*, i.e., you have to get very unlucky with your random decisions for quicksort to be slow. In return for this tiny chance of getting unlucky, quicksort does away with the need for temporary storage in merge sort.

2.3.6 Post-Lecture Ed Notes

One student asked about parallelizing mergesort. CLRS discusses parallel versions of merge sort in Chapter 26.3. Long story short, if you just parallelize the recursive calls the best you can get is $\Theta(n)$ because there's still a $\Theta(n)$ sequential bottleneck for doing the merge. But you can, in fact, parallelize the merge (in a nonobvious way) which gets you a much better bound of $\Theta((\log n)^3)$ in an idealized scenario with infinite number of CPUs (as far as I understand from skimming the chapter).

Another thing that is outside the scope of this class, but still interesting, is to observe that merge sort is very good for external sorting, i.e., sorting such a huge amount of data that it doesn't all fit in memory at the same time. This is in part because merge only requires looking at the "tips" of each half of the array, rather than the random access required by other algorithms. Interested students may read the section on external sorting in Knuth.

In class we saw three methods of solving recurrences: the recursion tree method, the handwavey algebraic substitution method, and the ansatz-and-induct method. There is a theorem called the "Master Theorem" that can sometimes be used to immediately solve the recurrences that we ran into while doing the latter two methods. You are welcome to study the master theorem on your own and use it on exams (if properly cited). But in my experience the master theorem is a lot of work to memorize, doesn't save you much time compared to the three methods we saw in class, and is usually useless anyways (e.g., as far as I know it is completely useless for the expected-time analysis of quicksort that we'll see on Monday).

It is possible to implement merge sort in a natural, nonrecursive way: first apply merge to every pair of 2 items, then every subsequent group of 4, etc. Implementing this nonrecursive version might be a fun task.

In class, we found it significantly easier to prove time bounds for merge sort restricted only to the case where n is a power of two. You are welcome to do the same on exams for solving such recurrences, as long as you explicitly state that's what you're doing. Some of you, however, may be uncomfortable with this restricted proof. And rightly so. If you are such a student, you may wish to convince yourself that our analyses also hold for inputs with non-power-of-two size. There are many different ways to do this; here are a few pointers:

- As far as I know, CLRS explicitly does not cover this (see the discussion at the very beginning of Section 4.6)
- The Erickson book discusses one approach on page 34 ("Ignoring Floors and Ceilings Is Okay, Honest")
- I'm not sure if Knuth or SW discuss this issue.
- For MergeSort in particular, I think you can adapt the recursion tree approach to see that it's true for all n. Sketch: every element in the original list is given to at most one "worker" on each level, so the total work is at most Cn on each level. And the argument that the depth is at most $O(\log n)$ still holds. So total time is still $CnO(\log n) = O(n \log n)$.
- Let $T(n)$ be the worst-case runtime of the algorithm on size n input. If you can convince yourself that $T(n)$ is increasing (or at least eventually increasing), then proving $T(n) = O(f(n))$ for only the cases $n = 2^k$ is actually sufficient (this is a nonobvious claim you might want to try proving: you'll need one more assumption, which is that $f(n)$ "doesn't grow too fast," specifically I think you need to assume $g(n) = O(f(n))$ where $g(n) = f(2n)$. That condition itself is kind of interesting; I think you can interpret it as a limited variant of Lipschitz continuity)
- For many algorithms, a variant of the following "padding technique" suffices to tell you at least there exists an algorithm with the desired complexity. Suppose you have an algorithm A that sorts any list of size 2^k and runs in time $O(f(2^k))$ on inputs of size 2^k, where $f(n)$ is a function that "doesn't grow too fast" (this can be made precise). Then there is an algorithm Y that sorts any input of size n (not necessarily a power of two) in $O(f(2^{\lceil \log n \rceil})) + O(n)$ time (where $\lceil \log n \rceil$ means $\log n$ rounded up to the nearest integer). In particular, if $f(n) = \Omega(n)$ and doesn't grow too fast, then algorithm Y will run in time $O(f(n))$, which is what we want. Algorithm Y does the following:

  1. First, find the maximum element M in the list (takes $O(n)$ time).

2. "Pad" the list, inserting $2^{\lceil \log n \rceil} - n$ extra copies of $M+1$ at the end of the list (depending on how your list is stored, takes about $2^{\lceil \log n \rceil} - n = O(n)$ time).
3. Run algorithm X to sort the padded list (takes time $O(f(2^{\lceil \log n \rceil})) = O(f(n))$, assuming $f(n)$ doesn't grow too fast.
4. The copies of $M+1$ you added are now still at the end of the list; remove them. (time $O(n)$).
5. Adding those up, the total time is $O(n) + O(n) + O(f(n)) + O(n) = O(f(n)) + O(n) = O(f(n))$, assuming $f(n) = \Omega(n)$.

Name: _____ **Stanford ID Number:** _____

2.3.7 Lecture 6/28 Practice Quiz Question

Here is the `Merge` code from class:

```
def merge(A, l, m, h):
    # assume [l, m) sorted and [m, h) sorted
    T = (scratch space of size >=N)
    k, i, j = 0, l, m
    while i < m and j < h:
        if A[i] <= A[j]:
            T[k++] = A[i++]
        else:
            T[k++] = A[j++]
    while i < m:
        T[k++] = A[i++]
    while j < h:
        T[k++] = A[j++]
    i = 0
    while i < h - l:
        A[l + i] = T[i]
```

Problem 2: Explain why it is OK to insert the following line at the start of the `merge` function:

```
if A[m-1] <= A[m]: return
```

2.4 Week 1 Homework

2.4.1 Lecture 6/24

Big-O

Problem 1: Review the definition of big-O in CLRS. Then prove carefully that:

1. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$,
2. $O(p(n)) = O(n^k)$ if $p(n)$ is a polynomial of degree k, and
3. $\sum_{i=1}^{n} f(n) = O(n^2)$ assuming $f(n) = O(n)$.

Hint: Hint(s) are available for this hproblem on Canvas.

Problem 2: (Do Knuth 1.2.11.1.6; not copying here for copyright reasons)

Selection Sort

Problem 3: Devise a sequence of inputs (one for each length n) that seems to maximize the number of times you update the running minimum (i.e., run `m=j` in the algorithm shown in the 6/24 lecture) in selection sort's inner loop. Compute (in terms of n) how many times it updates on this input. (No need to prove it does maximize, but if you want to try, consult Knuth exercise 5.2.3.6)

Algorithm Design

Problem 4: (Diagrams for this hproblem are available in Erickson, page 49.) Provide pseudocode for an algorithm that sorts a stack of pancakes by diameter. The only operation you are allowed to perform on the stack is to place a spatula under a pancake of your choice in the stack, then flip the substack of pancakes above your spatula. Your algorithm should take $O(n)$ flips. Can you adapt this algorithm to sort a list on a normal computer in $O(n)$ time? Why or why not?

Problem 5: (Prep for heapsort) Answer the following questions:

1. Provide pseudocode for a modified version of selection sort that takes time $O(n\sqrt{n})$.

   (Hint: First divide the n elements into $\sqrt{n}$ groups of $\sqrt{n}$. Feel free to assume that n is a square number in your time analysis.)

2. Describe how, for any integer k, you can generalize that construction to get a version that takes time $O(n \sqrt[k]{n})$.
3. Prove that $\sqrt[r]{n} = O(1)$ and $n \sqrt[r]{n} = O(n)$.
4. Explain why we **cannot** take $k = n$ in part (2) to get a version that takes time $O(n \sqrt[r]{n}) = O(n)$.
5. Show how to modify the construction to get time $O(n \log n)$.

For all parts, your construction may use $O(n)$ additional memory.

Hint: Hint(s) are available for this problem on Canvas.

2.4.2 Lecture 6/26

Insertion Sort

Problem 6: Notice that the left-hand portion of the array is always sorted during insertion sort. Show how to use binary search to find the insertion point in $O(\log n)$ time. Does this variant make insertion sort $O(n \log n)$ time? What if you use a linked list instead of an array to store the list?

Problem 7: An *inversion* in a list $A_0, \ldots, A_{n-1}$ is a pair (i, j) of indices with $i < j$ but $A_i > A_j$. Prove that the number of swaps performed by the insertion sort algorithm we saw in lecture is equal to the number of inversions in the original list.

Hint: Hint(s) are available for this problem on Canvas.

Merge Sort

Problem 8: (Erickson) Provide pseudocode for an $O(n \log n)$ time algorithm to count the number of inversions in a list.

Hint: Hint(s) are available for this problem on Canvas.

Average Time Analysis

Problem 9: Determine the average number of times you update the running minimum (i.e., run `m=j` in the algorithm shown in the 6/24 lecture) in selection sort, assuming every permutation of n distinct elements is equally likely for the input.

Hint: Hint(s) are available for this problem on Canvas.

2.4.3 Lecture 6/28

(Pushed to week 2!)

2.5 Lecture 7/1 Notes

2.5.1 Sorts We Know So Far

1. Selection sort: $\Theta(N^2)$ in all cases
2. Insertion sort: $\Theta(N)$ best, $\Theta(N^2)$ average and worst
3. Merge sort: $\Theta(N \log N)$ in all cases, but it requires a lot of extra memory and data movement

Today we'll see one of the all-around best sorting algorithms: *quicksort*. Similar to merge sort, quicksort is a divide-and-conquer scheme. *Unlike* merge sort, it is *randomized*. We will see that quicksort has $\Theta(N \log N)$ behavior on all inputs with *very high probability*, i.e., you have to get very unlucky with your random decisions for quicksort to be slow. In return for this tiny chance of getting unlucky, quicksort does away with the need for temporary storage in merge sort.

2.5.2 Quicksort

In merge sort the key operation we took advantage of was *merge*: take two sorted lists and make one big sorted list.

In quicksort the key operation we will take advantage of is *partition*: given a list and an element X in the list, rearrange that list so that everything *below* X is less than it and everything *above* X is greater than it. Note this does not require those two sides to be sorted within themselves.

Example: partition $[8, 10, 5, 20]$ on 8: $[5, 8, 20, 10]$ is a valid outcome, as is $[5, 8, 10, 20]$. In general we don't know which of those we'll get; it depends on the exact partitioning algorithm.

Partitioning algorithm adapted from SW:

```
def partition(A, lo, hi):
    # rearranges A[lo], ..., A[hi-1] and returns an index p where:
    # - A[lo],...,A[p-1] are all AT MOST A[p]
    # - A[p+1], ..., A[hi-1] are all AT LEAST A[p]
    swap A[lo] <-> A[random(lo, hi)]
    v = A[lo]

    i, j = lo, hi

    while True:
        while A[++i] < v:
            if (i+1) >= hi:
                break
        while A[--j] > v:
            pass
        if i >= j:
            swap A[lo] <-> A[j]
            return j
        swap A[i] <-> A[j]
```

Recall eventually we want the left "half" of the array to be all things less than v and the right "half" of the array to be all things greater than v. The basic insight of this pivoting algorithm is to find something "on the left" that is too big, and something "on the right" that is too small. We know those two must be out of place, so we swap them.

Show how it works on an example.

Theorem 8. *Partition is correct*

Proof. Outer loop maintains the invariant: $A_{lo}, \ldots, A_i$ are all at most v, and $A_j, \ldots, A_{hi}$ are all at least v.

First inner loop maintains this invariant as well, so when we exit the first inner loop we know $A_{lo}, \ldots, A_{i-1}$ are all at most v. There are two possibilities, depending on how we exited the loop:

1. Either $i = hi$, or

2. $A_i \geq v$.

The second inner loop maintains the invariant that $A_j, \ldots, A_{hi}$ are all at least v. So when we exit that loop we see that $A_{j+1}, \ldots, A_{hi}$ all at least v, while $A_j \leq v$.

Suppose we exit on line 18 (this includes the case $i = hi$). At that point we know that $A_0, \ldots, A_{i-1}$ are at most v, while $A_{j+1}, \ldots, A_{hi}$ are at least v, and $A_j \leq v$. Hence, swapping A_j with A_0 ensures that all of $A_j, \ldots, A_{hi}$ are at least v. But $j \leq i$, and we knew $A_0, \ldots, A_{i-1}$ were at most v, so we've successfully partitioned! □

Theorem 9. *Partition takes time* $\Theta(n)$

Proof. The distance $j - i$ decreases by one on each inner loop iteration, hence takes $\Theta(n)$ time before they meet. □

Let's stop and observe something interesting about partition: afterwards, A_p is in its final sorted position! So we no longer need to worry about moving it, and we can sort both remaining "halves" separately. This observation leads to the quicksort algorithm:

```
1 def quicksort(A, lo, hi):
2     if (hi - lo) <= 1: return
3     p = partition(A, lo, hi)
4     quicksort(A, lo, p)
5     quicksort(A, p+1, hi)
```

Theorem 10. *Quicksort takes worst-case* $\Theta(n^2)$ *time.*

Proof. Suppose we get extremely unlucky and always choose the smallest element in the array to pivot on. This will result in $p = 0$ so our recursive call will be one of size 0 and another of size $n - 1$. This repeats to get work $n + (n - 1) + \cdots + 1 = \Theta(n^2)$. □

Think-pair-share: what is the chance we get *that* super unlucky?

Theorem 11. *Quicksort takes best case* $\Theta(n \log n)$ *time.*

Proof. (Hand-wavey) If we get lucky and choose the median every time, then we end up splitting the array in half (minus one), and the recursion tree looks identical to that of merge sort.

Actually, it's a bit better than merge sort because we're taking one guy out every time. □

Theorem 12. *Quicksort takes* expected *time* $\Theta(n \log n)$ *when all elements are distinct.*

Proof. (Adapted from SW) We're going to count only *comparison* operations, i.e., lines 11 and 14 in the partition method. This is OK, because quicksort always performs a comparison within a bounded number of operations.

Let the expected number of comparisons be $C(n)$, and $n = hi - lo$. When all the values are distinct, the partition method makes exactly $n + 1$ comparisons: $n - 1$ "useful" comparisons, and then a final 2 comparisons when i and j hit their boundary. We then recurse on instances of size p and $n - p - 1$, where p is the offset of the pivot from lo. But we picked the pivot at random, so each of the options $p = 0, 1, \cdots, n - 1$

are equally likely. This tells us

$$\begin{aligned} C_n &= (n+1) + \frac{1}{n}\sum_{p=0}^{n-1}(C_p + C_{n-p-1}) \\ &= (n+1) + \frac{1}{n}\left(\sum_{p=0}^{n-1} C_p\right) + \frac{1}{n}\left(\sum_{p=0}^{n-1} C_{n-p-1}\right) \\ &= (n+1) + \frac{1}{n}\left(\sum_{i=0}^{n-1} C_i\right) + \frac{1}{n}\left(\sum_{i=0}^{n-1} C_i\right) \\ &= (n+1) + \frac{2}{n}\sum_{i=0}^{n-1} C_i \end{aligned}$$

At this point, we get a sum that is beyond what we would expect you to be able to solve on an exam. But it's fun to see how anyways!

When you see sums like this, you should always first try subtracting the value for C_{n-1} from the value for C_n. Unfortunately, in this case, I don't think that turns out to be very helpful (at least I couldn't figure out how to solve it from there). The trick is to *clear denominators* first; it turns out this makes the sums much, much easier to cancel out. In this case, let's look at:

$$\begin{aligned} nC_n - (n-1)C_{n-1} &= n(n+1) + 2\sum_{i=0}^{n-1} C_i - (n-1)n - 2\sum_{i=0}^{n-2} C_i \\ &= 2n + 2C_{n-1} \\ C_n &= 2 + \frac{n+1}{n} C_{n-1} \end{aligned}$$

Notice how clearing denominators on the sums made them cancel out nicely!

At this point we need one more trick: divide both sides by $n+1$ to get

$$\frac{C_n}{n+1} = \frac{2}{n+1} + \frac{C_{n-1}}{n}.$$

The key here is that there's now a clean recurrence; when you rename $S_n = \frac{C_n}{n+1}$, we see this becomes just $S_n = \frac{2}{n+1} + S_n$. Expanding that out gives us

$$\frac{C_n}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{3},$$

where notice we stop at $\frac{2}{3}$ because the term $\frac{C_1}{2} = 0$.

Hence,

$$C_n = 2(n+1)\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{3}\right) = 2(n+1)(H_{n+1} - H_2) = \Theta(2n\log n).$$

(Note the last step relies on the fact that $H_n = \Theta(\log n)$. There is a beautifully visual proof of this fact that I suggest calculus-inclined readers to try to find. Hint: recall the integral definition of $\log n$*.)* □

CLRS has an alternate solution looking at the probability any two values are compared. Much easier to solve the resulting algebra. If we have time, might show both. But the homework is already going to walk through yet another approach (involving the depth of the call stack) so might not be worth it.

Why Quicksort?

Relatively little data movement. In-place. Tight inner loop. Good cache behavior; just linear scans.

2.5.3 Selection

Let's finish by seeing one more application of the partition algorithm: quickselect.

Here's the problem: suppose you want to find the ith smallest element in a list. One thing you could do is just perform quicksort (or any other $O(n \log n)$ algorithm) and then look at the element in the resulting list at index $i-1$. But is there any way to do better? Clearly we can at least find the *min* quickly (linear scan; we did it in selection sort!).

One neat way to do it is to adapt the quicksort routine. But notice, after partitioning, we know exactly which partition the ith element will land in! So in fact, we only need to recurse into one partition. This is made explicit below:

```
def select(A, lo, hi, i):
    if (hi - lo) <= 1: return A[lo]
    p = partition(A, lo, hi)
    if p == i:
        return A[p]
    if i < p:
        select(A, lo, p, i)
    if i > p:
        select(A, p+1, hi, i)
```

Quickselect Efficiency

But wait! Quicksort takes $\Theta(n \log n)$ time anyways right? Are we really saving anything?

In fact we are! The key is we're only making one of the recursive calls, so we do half as much work each time. Unfortunately, it's not very easy to adapt our analysis of quicksort to reflect this. Let's try and see why. Our quicksort recurrence was

$$C_n = (n+1) + \frac{1}{n}\sum_{p=0}^{n-1}(C_p + C_{n-p-1})$$

It "feels" like we should be able to make an analogous recurrence for quickselect by just dropping one of the recursive calls:

$$C'_n = (n+1) + \frac{1}{n}\sum_{p=0}^{n-1} C'_p,$$

but in fact that isn't valid! Why? Because, depending on the value of i, all the possible sizes to recurse into aren't equally likely! E.g., if $i = n/2$, it's impossible to recurse into a sublist of size smaller than $n/2$.

We can get a rough guess at the time of quickselect by looking at an "extremely average" scenario: suppose the pivot is always chosen to be the median. In that case, we'll always recurse into an exactly $\frac{n}{2}$-sized list, and halt after $\log n$ recursions. This gives us time:

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 = n\sum_{i=0}^{\log n}\frac{1}{2^i} = \Theta(n).$$

In fact, this rough analysis gives us the right asymptotic answer: the expected time for quickselect is $\Theta(n)$. We'll see a rigorous proof of this on the homework!

Quickselect is a *really* clever and quite surprising algorithm. It is possible (but nontrivial) to derandomize it; see Erickson for worked details on 'MoMSelect.'

2.5.4 Wrap Up

So we've seen:

- QuickSort

- QuickSelect
- Expected time

Now, let's go on to the practice quiz!

2.5.5 Post-Lecture Ed Notes

- It is very tempting to try to adapt the recurrence relation proof we saw for the expected time of Quicksort to show the expected time of Quickselect. But doing that correctly is actually very hard, for a somewhat nonobvious reason. This is discussed in the lecture notes for today, which I suggest you read.
- There was a question about how we are covering neither the master theorem nor generating functions. FYI, as far as I can tell, neither is in the ExploreCourses blurb. If you want to research them on your own, see re: master theorem, (previous Ed post) Re: generating functions, Knuth 1.2.9.
- You can prove the expected time result for Quicksort in many different ways. I know of at least three:
  1. The recurrence relation approach we saw in lecture.
  2. A very slick approach based on computing the probability that any two elements of the array will be compared with each other. See CLRS 7.4.2. This approach is really pretty, but it's very specific to quicksort so not quite as useful more generally.
  3. It is also possible to turn the recursion tree method (which we used to show a best-case bound, if you always pick the median) into a rigorous argument for expected time. The way to do this is to prove that, with very high probability, the tree does not have depth larger than $k \log n$ for some large k. How should you do that? Well, it turns out that you have a good chance of picking your pivot in the middle 50% of the array each time, and if you do that, then the sublists you recurse on have size at most 75% the size of the original list. It turns out this is close enough to the merge-sort tree (drop to 50% of the size each time) to bound the depth with high probability.
     - You'll do something very similar to this approach on the homework, except for Quickselect. To adapt it to Quicksort, you'll probably need to look at probabilities instead of expected value and use "Hoeffding's inequality" along with the "union bound." If you're interested in this, send me a message and I can write up some guiding hints to work you through the proof. (In particular, you can pick a constant C large enough to make the following work:)
     - Use Hoeffding's inequality to bound the probability that, in a sequence of m independent coin flips, at most $\frac{m}{C}$ of them come up heads. (Nothing too specific to Hoeffding's here, e.g., should be able to use a Chernoff bound as well.)
     - Think about the recursion tree for Quicksort. Then revisit week 2 HW problem 2, but this time use part (1) above to bound the probability that some particular leaf of the recursion tree is deeper than $C \log_{4/3} n$.
     - Apply the union bound to your result from (2) with this count to get an upper bound on the chance that any node in the recursion tree is deeper than $C \log_{4/3} n$.
     - If the event described in (3) occurs, what's an upper bound on the worst possible runtime? What if the event described in (3) does not occur? You should be able to combine those two runtime bounds with the bound on the probability of that occurring from (3) to upper bound the overall expected time for Quicksort.
     - In fact, this analysis tells you a bit more than the one we did in lecture because it also naturally gives you the probability that any branch is very deep. (Though I'm told it's possible to extend the recurrence relation approach to a recurrence for the variance, so maybe that would be even better.)
- In practice, Quicksort is one of the fastest sorting algorithms we have. This is due to a number of reasons. One is that there is a lot less data movement than mergesort (you relatively rarely swap in

Quicksort, whereas mergesort moves items into temporary space and back every time you call merge). Also, the inner loops have very good spatial locality. Also, you always compare one memory read to a local variable, so each inner loop iteration only does one memory read, not two.

- Someone at the end of class asked why I said Quicksort uses $\Theta(\log n)$ expected additional memory, rather than $\Theta(1)$. The reason is that recursion isn't free: when you make a recursive call, the computer first has to save the value of all of your local variables before the call so that they can be restored once that call returns. Hence every call in the call stack requires its own constant amount of space. The recursive call stack is expected to have depth $\Theta(\log n)$, hence the expected space usage is $\Theta(1) \cdot \Theta(\log n) = \Theta(\log n)$.
- As mentioned, you can derandomize both Quicksort and Quickselect to get an $O(n \log n)$ truly worst-case algorithm. The trick is a very clever way to find the median in linear time. Erickson has an extended discussion of this algorithm, the "MoM" algorithm, in section 1.8.

Name: _____ **Stanford ID Number:** _____

2.5.6 Lecture 7/1 Practice Quiz Question

Here is the PARTITION code from lecture:

```
def partition(A, lo, hi):
    swap A[lo] <-> A[random(lo, hi)]
    v = A[lo]
    i, j = lo, hi
    while True:
        while A[++i] < v:
            if (i+1) >= hi:
                break
        while A[--j] > v:
            pass
        if i >= j:
            swap A[lo] <-> A[j]
            return j
        swap A[i] <-> A[j]
```

Imagine running the partition algorithm once on the array $[2n, 2n-1, \ldots, n+1, n, n-1, \ldots, 2, 1]$. Answer the following questions. No justification is necessary. You should try to get an *exact* (nonasymptotic) count, but it's OK if your counts are off by a small constant number. Feel free to use the back for scratch space.

Part 1: How many swaps would occur if the random pivot was chosen to be 5 (i.e., at index $2n-5$)?

Part 2: How many swaps would occur if the random pivot was chosen to be $2n-5$ (i.e., at index 5)?

Part 3: Devise a general formula for how many swaps occur if the pivot is chosen to be the element at index i. *(You might need a case split depending on whether i is in the first or second half of the list.)*

2.6 Lecture 7/3 Notes: Heaps, heapsort, and stability

2.6.1 Thoughts on Feedback from the First Homework

Thanks everyone for the great feedback! Some thoughts below:

- Difficulty seems about right. Median was about 10 hours, which is expected: Stanford says 5 units means 5h of lecture and 10h of extra work. If you do not have the prerequisites, of course, you should expect to spend significantly longer studying every week.
- LaTeX was hard. Unfortunately that's really something you have to practice, practice, practice. You're more than welcome to ask LaTeX help questions on Ed.
- A few people mentioned having to refer to textbooks or do extra research for the problems. **That's fine,** but please note that the questions are designed to be solvable (given enough time) using just material from lecture + the hints provided on Canvas. For example, one or two of the textbooks might give a proof for the inversions problem. But at this point in your career you should be able to read the definition given in the problem, understand what it means, and prove things like the lemma given in the hint. So, again, totally fine to find things in a textbook, but just note that there is a skill here (understanding and working with curt definitions) that it circumvents practice of.
- There was some concern that people could only do many of the problems after looking at the hints, but there won't be hints on the exam. Actually, it's the exact opposite! The only reason we hide the hints on a homework is because you have so much time; **on a timed exam the problems will be easier and we'll give more hints/guidance/structure.** More generally, it is expected that you can't solve many of the HW problems without hints (otherwise I wouldn't provide hints). But I think it's still useful to your learning to spend a few minutes thinking about each problem without hints.
- A few folks suggested having a per-HW pinned Ed post with suggested problems, difficulty ratings, corrections, etc. — we'll start doing that with HW2!

Lecture feedback:

- Seems like the pacing is pretty good; some of you think it's too fast and some think too slow, but the median seem to think it's just right. We'll reëvaluate as the quarter goes on, but I think this is a good pace. For folks worried about dropping topics, don't freak out yet. Currently the only thing we need to drop is noncomparison sorts, which aren't very useful anyways. And we have a few catchup lectures later in the quarter that we can use up. On net, we still cover significantly more than the normal quarter (because we have more frequent and longer lectures).
- I'll try to start writing the pseudocode up before lecture, to better manage font size, etc. If you like to copy it down into your notes, you may want to come by a bit early to do so.
- Someone found a typo in the lecture notes; thank you!!! Please be on the lookout for typos and report any typos you find. We're putting together lecture notes, homework problems, practice questions, exams, and solutions all in real time; that's 25+ pretty dense pages of content per week. Unfortunately we're not perfect and there are going to be typos and mistakes. We will try to be prompt in confirming + correcting any issues reported!

2.6.2 Sorts We Know So Far

For inputs stored in a contiguous array:

1. Selection sort: $\Theta(N^2)$ in all cases.
2. Insertion sort: $\Theta(N)$ best, $\Theta(N^2)$ average and worst.
3. Merge sort: $\Theta(N \log N)$ in all cases, but it requires a lot of extra memory and data movement.

4. Quicksort: $\Theta(N \log N)$ *expected*, but $O(N^2)$ worst case, and still uses $\Theta(\log n)$ extra space for the call stack.

The holy grail would seem to be an absolute $\Theta(N \log N)$ worst case, in-place sort. Is this possible? Yes, and we'll learn one (heap sort) today!

Surprisingly, heapsort is essentially just an implementation detail on top of selection sort. In some sense, that makes selection sort both our asymptotically *worst* algorithm and also our asymptotically *best* algorithm!

2.6.3 Dynamic Sorting, Heaps, and Priority Queues

Before we get there, though, we're going to take a little detour and learn about *priority queues* and the *max-heap data structure*.

So far in this class we've been assuming you start with n items and want to find out their exact sorted order with respect to each other.

Consider the following problem: we want to keep track of the tallest current Stanford student. (Draw four people with their heights.) How might we do this?

- Could keep track of an unsorted list of all the Stanford students out there, and every time we're curious about the tallest we scan the whole list to find out who it is. With doubly linked lists, this has $O(1)$ insertion/deletion but $O(n)$ queries.

- Could keep track of a *sorted* list. If it's stored as a doubly linked list that makes deletion $O(1)$, insertion $O(n)$, and queries $O(1)$.

We're kind of stuck here: no matter what, it seems like we have to cycle through everyone at Stanford very frequently, even if only one new student joins!

The classic data structure to solve this problem is called a *max-heap*. It lets us perform the following operations:

- *Insert* into the max heap in $O(\log n)$ time

- *Delete* from the max heap in $O(\log n)$ time

- *Find the maximum* of everything the max heap in $O(1)$ time

Even if everyone in the world enrolled at Stanford, this takes only about 33 operations (times the constant factor in the big-O) to insert/delete one student! Pretty good.

How does it work? Our good friend the *binary tree* shows up here. We're going to organize everyone into a binary tree. Who should be at the top? Let's put the tallest person! In fact, our heap will be *defined* by this property, that parents are bigger than their children:

Definition 5. *A* max-heap *is a binary tree with the following property: the value of every node is* at least *the value of its children.*

As with any kind of binary tree, one of the big things we'll need to think about is balance. Informally, the best balance we can hope for is for the tree to be *complete*:

Definition 6. *A binary tree is* complete *if layer i from the root has 2^i nodes, except possibly the last layer, and all nodes in the last layer are "as far left as possible."*

Walk through some example binary trees and ask (1) is it a max-heap? (2) is it a complete binary tree?

Important to clarify this is just a definition: we haven't yet seen how to put everyone into this form or how to insert/delete people.

Almost-Heaps and Heap Fixing

Consider inserting into a heap. One thing you could try to do is just add the new item as a leaf. But that might violate the heap property, if the new item is bigger than its parent.

The key operation that makes heaps possible is one we'll call *fixing*. Fixing is the process of turning an *almost-heap* into a *true heap*.

Definition 7. *An* almost heap broken at node x *is a binary tree that can be made into a heap by changing the value of* x*.*

(Draw some examples.)

How to fix an almost-heap? Well, suppose you have an almost-heap broken at x. There are two possibilities:

1. If x's value is bigger than its parent, swap it with the parent. Now you have either a heap, or if x's value is bigger than its new parent, an almost-heap broken at x again. Repeat until you have a heap!
2. If x's value is smaller than one of its children, swap it with the biggest of its children. Now you have either a heap, or an almost-heap broken at x again. Repeat until you have a heap!

The first process is called *swimming*, the latter is called *sinking*. You should convince yourself that either terminates in time $O(d)$, where d is the depth of the heap. If we make sure to always add new nodes in such a way to fill the tree maximally, then this takes time $O(\log n)$.

Pseudocode looks like:

```
# CONVENTION: if x is the root, then parent(x) = x. If x does not have the
# r/l_child, then r/l_child(x) = x.
def swim(x):
    while x.value > parent(x).value:
        swap x <-> parent(x)
def sink(x):
    while x.value < largest_child(x).value:
        swap x <-> largest_child(x)
def update(x, new_value): # won't need this for this class
    old_value = x.value
    x.value = new_value
    if new_value < old_value:   sink(x)
    else:                       swim(x)
```

(Quote SW workplace joke.)

Note this is pretty high-level pseudocode; we haven't yet talked about how to represent the heap, so this doesn't tell us much.

Heap Operations: Insertion, Deletion, Find-Max, Creation

Finding the maximum is easy: according to the heap property, it's at the root. Proving this carefully using the definition of the heap property is a nice exercise.

```
def find_max(heap):
    return root(heap).value
```

How should we insert into the heap? The idea is to insert the new node on the very last level. You now have an almost-heap broken at that node, so fix it.

```
def insert(heap, value):
    x = new node
    x.value = value
    insert x at the leftmost empty slot in the last unfilled layer of heap
    swim(x)
```

How should we delete from a heap? Well, it's easy to delete a *leaf* from the heap: just do it; there's no way for that operation to harm the heap property! To delete some *other* node from the heap, replace its value with that of a leaf, delete the leaf, then fix the heap at that node.

```
def delete(heap, x):
    l = rightmost leaf in last layer of heap
    if x is l:
        remove x from heap
    else:
        old_value = x.value
        swap x.value <-> l.value
        remove l from heap
        if x.value > old_value: swim(x)
        else:                   sink(x)
```

How can we turn a random tree into a heap? This operation is called *heapify*. Well, at least if it's complete, note that all the leaves (in the last level) are already heaps (vacuously). Then, every node on the penultimate level is an *almost-heap* broken at that node. You can turn *those* into heaps by sinking them. Repeating this up the tree, you eventually turn the root itself into a heap!

```
def heapify(heap):
    for level in heap.bottom_level(), ..., heap.top_level():
        for node in level:
            sink(node)
```

Theorem 13. *All of the heap operations above are correct. If the heap is a complete binary tree, all of them except for heapify take $O(\log n)$ time. Heapify takes time $O(n \log n)$.* (On the homework you'll prove a tighter bound for heapify!)

Proof. (Sketch) Sink/Swim maintain the invariant that the tree is always either a heap or an almost-heap, and if it starts off broken at a node that's too small (too large), sink (swim) will only halt once the tree is a heap.

All other correctness claims follow from correctness of sink/swim. Heapify maintains the invariant that every node in the previous level is the root node of a heap defined as the subtree rooted at that node.

For time complexity, it follows from the fact that such a complete tree never has depth more than $\Theta(\log n)$, along with the fact that Sink/Swim takes time (in the worst case) proportional to the depth of the tree. □

There is at least one nuance in the above, which is that we need to be able to quickly find the rightmost leaf on the final layer (for delete). We'll see soon how to do this.

Heap Sort

We're now ready to see pseudocode for heapsort. The basic idea is to do selection sort with two small modifications: First, we pick out the max every time rather than the min (this isn't critical). Second, we store the elements in a heap so that we can quickly find the max element.

```
def heapsort(A):
    heap = complete binary tree made of elements in A
    heapify(heap)
    for i = 0, 1, ..., n-1:
        delete root of heap and place at A[n-1-i]
```

Theorem 14. *Heapsort is correct and takes worst-case time $O(n \log n)$.*

Proof. Correctness follows from correctness of the heap operations and the invariant that, whenever the for loop is run, $A_{n-i}, \ldots, A_{n-1}$ are the largest i elements in sorted order.

Time follows from the time complexity of the heap operations. □

At this point, we have an $O(n \log n)$ worst-case time algorithm, but it seems like it should require $\Theta(n)$ extra space to store the heap. What gives?

The secret is a very clever way of *reusing the input array A itself to represent the heap!*

How to Represent a Heap?

In the above pseudocode we've been very tight-lipped about how exactly the heap is represented/stored in the computer.

In 106B you may have seen binary trees represented as *linked structures*, i.e., with pointers. Surprisingly, that's not strictly necessary! If we make sure the heap is a complete binary tree, we can store all the nodes contiguously in an array.

(Draw picture)

The basic idea is to store the tree level-by-level. Level 0 has $2^0 = 1$ node, which gets index 0. Level 1 has $2^1 = 2$ nodes, which get indices 1 and 2. Etc.

In our pseudocode, the main operations we need to do are:

1. Find your parent,
2. Find your left/right children, and
3. Find the rightmost leaf (to either delete it or insert after it)

The rightmost leaf is always the last thing in the flattened representation. The right child is always one past the left child. Finding your parent is going to be the inverse of the process to find your parent's child. So let's focus just on finding your left child.

Suppose you're looking at the first node on level i. Because each level has two children per node on the prior level, that node will have index $2^i - 1$. Its left child, in turn, will have index $2^{i+1} - 1$. So if $x = 2^i - 1$ is the index of the node you're starting with, its child is at position $2x + 1$.

What if you're not the first node on a level? Then your index can be written $x = 2^i - 1 + k$, where k is the number of nodes before you on that level. Well, your left child is going to be on the level below you (hence starting at $2^{i+1} - 1$) and there will be $2k$ nodes before it on level $i + 1$, two for each node before you on level i. Hence, your left child will be at $2^{i+1} - 1 + 2k$, which can *also* be written $2x + 1$.

So in general, ignoring the corner cases where you have no parent or child:

1. Left child is $2x + 1$,
2. Right child is $2x + 2$,
3. Parent is $\lfloor (x - 1)/2 \rfloor$,
4. Last leaf is $N - 1$.

Using those identities, you can implement all of the heap pseudocode we saw earlier with the heap stored as an array.

An alternate approach to the contiguous array heap indexing rules If you start numbering at 1 instead of 0, there's also a cute explanation for these rules discussed at `https://cs.stackexchange.com/questions/87154/why-does-the-formula-2n-1-find-the-child-node-in-a-binary-heap`: the binary encoding of the index of each node describes the path taken to get to it from the root. E.g., $5 = 101_2$ means, reading from msb to lsb, "start at the root, then move left, then move right."

With some work, you can convince yourself that this encoding is both dense for complete trees (there are no unused slots in the array) and also that nodes are placed in level order (first level, then second level, etc., and within a level from left-to-right). So it's the same as the encoding we wanted above, just starting at one instead of zero.

Now, the path to your left child is the path to you and then one move left. So your left child's index must be your index with a zero appended to its binary encoding, e.g., the left child of 101_2 should be 1010_2. But appending a zero to the binary encoding of a number just means multiplying it by two, hence in this encoding you get the left child c' of i' is at $c' = 2i'$. But letting i and c be the corresponding zero-based indices, we see $i' = i + 1$ and $c' = c + 1$, hence $c + 1 = 2(i + 1)$, i.e., $c = 2i + 1$ which is exactly what we saw before.

Heap Sort

So now we can think of heapsort in all its glory!

We first turn the entire array, *in place*, into a heap. This means the maximum element is at index 0, and the final leaf is at index $n - 1$. Then, we repeatedly "delete" the root from the heap. But think about what this deletion procedure does: it's first going to swap the root (maximum element) with the final leaf (at index $n - 1$), then forget about the leaf storing the max now at index $n - 1$, and then it's going to fix up the remainder of the heap from 0 to $n - 2$. In other words, it moved the maximum element to exactly the right place, and set up the remainder of the list in a perfect spot to repeat the whole process among the first $n - 2$ elements!

Let's see a miniature example. If we start with the array $[4, 3, 2, 1]$, the heapify routine will do nothing (it's already a heap!). Then the first deletion will turn the array into $[3, 1, 2, 4]$. Then the second deletion (now among the first 3 elements, not all four) will turn the array into $[2, 1, 3, 4]$. Then the third deletion (now among the first two elements, not all four) will turn the array into $[1, 2, 3, 4]$. And finally the fourth deletion will end up leaving it as-is: $[1, 2, 3, 4]$.

Theorem 15. *Properly implemented for arrays, heapsort takes worst-case time $O(n \log n)$ and $\Theta(1)$ additional memory.*

Proof. Time is the same, but now we don't need extra space to store the heap! □

Sorting Stability

Suppose we want to sort three students by height: Jackson (7 ft), Matthew (6 ft), and Annie (7 ft). What should the result be? Definitely Matthew should come first as the shortest, but there's some ambiguity between whether Jackson or Annie should come next since they both have the same height: one of MJA or MAJ seems acceptable.

In this particular scenario it might be preferable to 'tie-break' by sorting people of the same height by name, and oftentimes if you have ambiguity like this the true fix is in fact to make your comparison function more sensitive. But our algorithms aren't able to guess at how you wanted to compare otherwise-equal things, so what should they do instead? A reasonable desire is for the algorithm to be what we call *stable*:

Definition 8. *We say a sorting algorithm is* stable *if equivalent items are ordered in the sorted list exactly as they are ordered in the input list.*

For example, if the input was JMA with J and A the same height, the output of a stable sorting algorithm should be MJA. If the input was AJM, the output should be MAJ. You can see this as either a simple tiebreaking rule that makes the output predictable, or you might also interpret it as an assumption that the original ordering already means something and we should try not to violate it if possible.

2.6.4 Wrap Up

We're now pretty much done with our survey of sorting algorithms. We've "bottomed out" at a worst-case bound of $O(n \log n)$. Is that the best that can be done? We'll see next time!

Now, let's go on to the practice quiz!

2.6.5 Post-Lecture Ed Notes

You may also hear "sink" and "swim" called something like "percolate" or "bubble" or "promote." I think sink/swim are the best terms I've come across so far (I found them in SW) because they are short, use the same amount of space in a monospace font, and avoid ambiguity, e.g., with bubble sort. I made up the "almost-heap broken at x" term to write down invariants for sink/swim, but it's possible that there's some more-standardized terminology for this I'm not aware of.

There was a question about variants of heapsort that, like insertion sort, perform well when the heap is near-sorted. The one I've heard of being used in practice is smoothsort, which Wikipedia[1] describes like so: "Like heapsort, smoothsort is an in-place algorithm [...] The advantage of smoothsort is that it comes closer to $O(n)$ time if the input is already sorted to some degree, whereas heapsort averages $O(n \log n)$ regardless of the initial sorted state." Smoothsort is used by musl libc[2]. (Meanwhile, glibc[3] seems to use a combination of quicksort and insertion sort. Python[4] uses an algorithm called timsort[5] that apparently combines mergesort and insertion sort).

There was a question about the best-case behavior of heapsort. Knuth 5.2.3.32 is a relevant exercise: "Prove that the number of heapsort promotions, B, is always at least $\frac{1}{2}N \lg N + O(N)$, if the keys being sorted are distinct. Hint: Consider the movement of the largest $[N/2]$ keys." I haven't worked through this exercise yet, but if you do I'd be curious to hear about it! (There are also solutions in the back if you want a spoiler.)

[1] https://en.wikipedia.org/wiki/Smoothsort
[2] https://git.musl-libc.org/cgit/musl/tree/src/stdlib/qsort.c
[3] https://github.com/lattera/glibc/blob/master/stdlib/qsort.c
[4] https://github.com/python/cpython/blob/main/Objects/listsort.txt
[5] https://en.wikipedia.org/wiki/Timsort

Name: _____ **Stanford ID Number:** _____

2.6.6 Lecture 7/3 Practice Quiz Question

In lecture we saw how, when the input is stored in a contiguous array, one can implement heapsort with $\Theta(1)$ space and $O(n \log n)$ time overhead by using the input array itself to implicitly store the heap.

Explain why that approach can *not* be easily used if the input is stored as a linked list instead of an array, and upper bound the worst-case time complexity of the algorithm if you tried to use the same $\Theta(1)$-space approach but where the input is a linked list instead of an array.

2.7 Lecture 7/5 Notes: Comparison Lower Bounds and BSTs

2.7.1 Lower Bounds for Comparison Sorting

So far, all the sorts we've seen so far have one thing in common: the *only* thing they do with items is move them around and compare them. We call these algorithms *comparison sorting algorithms.*

Definition 9. *A* comparison sorting algorithm *is a sorting algorithm where the only operations it performs on its input are: (1) get the number of input elements, (2) compare two elements, and (3) rearrange the elements.*

On the one hand, this is quite powerful! We can use these algorithms to sort objects by weight given only a balance. We can sort by things that aren't numbers, such as sorting words in dictionary order.

On the other hand, it seems like we might be leaving some information on the table. Maybe if we knew for a fact that the objects were *numbers*, we could do something smarter? Numbers have prime factorizations, etc., that we could imagine trying to make use of. But comparison sorts have no way to 'introspect' on the contents of the keys. We won't look at any of these, but it turns out there are sorting algorithms that get better asymptotic behavior by using this information.

Sketch of the Proof

We'll defend two claims:

1. If there exists a comparison sorting algorithm that runs in time $f(n)$, then there exists a *decision tree* for sorting n-length arrays that has depth $f(n)$.
2. Every decision tree for sorting n-length arrays has depth at least $\Omega(n \log n)$.

Turning Algorithms Into Decision Trees

Let's think a bit more carefully about what a comparison sorting algorithm *is*. For a given input length n, the only way it can learn *anything* about the input is by asking to compare two items.

Notice an interesting fact: the very first two items chosen for comparison are *always the same*, regardless of the actual input! This fact follows because there's no way for it to learn anything about the input before the first comparison, so there's no way for it to base its first comparison on any other info about the list! In fact, the same argument tells us that, once we know the result of the first comparison, the next two items to be compared is *also* fixed, and not dependent on the rest of the input at all (only on what happened in the first comparison).

This leads us to think about comparison sorting algorithms as *decision trees.* Each node in the decision tree determines two indices to compare. Its children determine the next comparison to perform, depending on whether the result was 'less-than' or 'not less-than.' The leaves correspond to termination of the algorithm, at which point the algorithm must know exactly how to turn the list back into sorted order.

Definition 10. *A* decision tree *is a tree where internal nodes correspond to queries about the input and leaves give the solution to the problem for all inputs inputs taking that branch.*

Theorem 16. *If a comparison sorting algorithm runs in worst-case $f(n)$ time, then there exists for every n a decision tree of depth at most $f(n)$ that sorts n-length arrays.*

Proof. Start with an empty decision tree and simulate the comparison sorting algorithm. Every time it makes a comparison, add a node to the tree representing that comparison and then continue simulating both of the possible results of that comparison in different subtrees. Whenever the algorithm returns, add a leaf with the corresponding rearrangement. □

Let's think about the decision tree for *merge sort* running on 3 items, $[A_0, A_1, A_2]$. What are the first two items to be compared? It recurses on $[A_0]$ and $[A_1, A_2]$, hence the first two things to be compared are A_1 and A_2 during the merge call in the second recursive call. Now we've got two options: either $A_1 \leq A_2$ or not. We'll draw those as two subnodes here; let's leave the right one empty for now and start on the left

one. (Trace that branch down; draw the full tree; each node should include the current state of the array and what comparisons are made.)

Notice every possible input takes *exactly* one branch, and all the branches terminate knowing exactly how to sort the input. In some sense, these trees fully capture the behavior of the comparison sort algorithm. What's the depth? Well, it's the number of comparisons that the algorithm would have had to make on such an input. This is at worst the total time taken by the algorithm.

(Note we haven't yet talked about what to do if the algorithm is randomized; we'll mention it briefly at the end.)

Lower Bounds on Depth

Now let's ask an interesting question: how many leaves does the tree have? Well, notice that *every* $n!$ *permutation* of the list $1, 2, \ldots, n$ must take a different branch (why? if two took the same branch then the sorting algorithm would have to rearrange them in the same way, but then they wouldn't be sorted!). This means our tree must have *at least* $n!$ leaves.

We now need the following lemma

Lemma 1. *A binary tree with k nodes must have at least $\lfloor \log k \rfloor + 1$ many levels.*

Proof. By contradiction: if you had at most $\lfloor \log k \rfloor$ levels, the most nodes you could have is with the perfect binary tree, which would have $2^{\lfloor \log k \rfloor} - 1 < k - 1 < k$ nodes. □

OK, so we have a tree with $n!$ leaves so it has at least $\log n! = \Theta(n \log n)$ many levels. That looks familiar! In fact, this proves the following theorem:

Theorem 17. *Any comparison sorting algorithm requires $\Omega(n \log n)$ time to sort some list of size n.*

By corollary, then, merge sort and heap sort are *asymptotically optimal*: it's not possible to do better, except perhaps in terms of constant factors *or if you go outside the comparison sorting mindset.*

Randomized Algorithms

The above proof is a bit ambiguous about whether and how it applies to randomized algorithms like Quicksort. The basic idea is, during the simulation, every time you make a random decision *make an entirely new tree* for each possible random decision you could have made. Now you've turned the algorithm into many different decision trees, one for each of the possible choices of the random decisions you could have made. But, the earlier argument still applies to each one of these trees, i.e., no matter how the random decisions are chosen it will still take $\Omega(n \log n)$ time.

We won't ask you to do anything like that in this class.

Noncomparison Sorting

It's possible to break this bound *if you get more info from the items other than just pairwise comparisons.* We're not going to talk about those algorithms. In practice, algorithms like Quicksort run so fast, and the flexibility that comes from allowing general comparison functions, makes it relatively rare that you ever see anyone using noncomparison sorts.

2.7.2 Searching

Definition 11. *The* search *problem is the problem of storing a set of items with the following operations:*

- Insert *an item into the set,*
- Remove *an item from the set, and*
- Search *for whether or not an item exists in the set.*

In fact, for all the algorithms we'll see, they can be augmented to not just report if an item exists in the set, but also do things like associate a different value with that item (hence making something like a key-value map).

Applications include:

- In a compiler, when it comes across a variable name it needs to check that it's been defined before and, if so, where its value is stored.
- When you ask to open a certain file by name/path, your computer needs a way to make sure that file exists and to find its contents.

2.7.3 Binary Search Trees

Definition 12. *A binary search tree is a binary tree where each node is annotated with a value, and the value of every node is:*

1. *Greater than (or equal to) the value of all its left-descendants, and*
2. *Less than (or equal to) the value of all its right-descendants.*

Binary trees are usually stored with each node having a left- and right-pointer to its children. It is sometimes also convenient to store a pointer to your parent.

The following pseudocode searches through a binary tree for a value v, inserting it if it doesn't exist:

```
def search_or_insert(root, v):
    if root is empty: # NOT FOUND, insert it
        node = new node
        node.value = v
        replace root with node
        return node
    if v == root.value:
        return root
    if v < root.value:
        return search_or_insert(root.left, v)
    if v > root.value:
        return search_or_insert(root.right, v)
```

We can do other things with a binary tree, e.g., find the minimum:

```
def find_min(root):
    if root is empty: (error)

    if root.left is not empty:
        return find_min(root.left)

    return root.value
```

We can delete a node similar to how we did with heaps: deleting a node that doesn't have a child is easy, so find such a node to swap with, do the swap, and then delete that node. It turns out the node you should swap with is the *smallest thing in the tree greater than yourself*, because then the swap-and-delete doesn't violate the BST property. (There are many different ways to delete, e.g., you could swap with the largest thing in the tree greater than yourself instead.)

```
def delete(node):
    if node.left is empty:
        replace node with node.right and return
    if node.right is empty:
```

```
        replace node with node.left and return
    successor = find_min(node.right)
    swap node.value <-> successor.value
    replace successor with successor.right and return
```

Implementing Linked Binary Trees

In the preceeding pseduocode, we have tried to be very agnostic to the underlying storage method. In practice, we'll usually use linked structures (pointers). The below C code shows how you could translate, say, the search/insert pseudocode into C and how to use it:

```c
#include <stdlib.h>
#include <assert.h>

struct node {
    int value;
    struct node *left, *right;
};

struct node *search_or_insert(struct node **root, int v, int do_insert) {
    if (!(*root) && !do_insert) return 0;
    if (!(*root)) {
        *root = calloc(1, sizeof(**root));
        (*root)->value = v;
        return *root;
    }
    if (v == (*root)->value) return *root;
    if (v < (*root)->value)
        return search_or_insert(&((*root)->left), v, do_insert);
    return search_or_insert(&((*root)->right), v, do_insert);
}

void main() {
    struct node *tree = 0;
    search_or_insert(&tree, 4, 1);
    search_or_insert(&tree, 5, 1);
    search_or_insert(&tree, 3, 1);
    assert(!search_or_insert(&tree, 2, 0));
    assert(search_or_insert(&tree, 5, 0));
    assert(search_or_insert(&tree, 5, 0)->value == 5);
}
```

Of particular importance is that we always pass around a pointer to a pointer to a node; this is because if, say, the tree is empty, we need to insert a new root node by overwriting the pointer to the root. We've also added a flag that lets you control whether it does just a search, or a search and insertion. If you would like to create a map instead of just a set, you can add a separate "associated value" field to the node struct.

A worthwhile practice problem would be to turn the above pseudocode into a nonrecursive form.

Worst-Case Time Complexity of BST Operations

If you insert in the order $1, 2, 3, 4, \ldots, n$, the total time taken will be $\Theta(n^2)$. Not good!

Average-Case Time Complexity of BST Operations

Suppose you insert in a randomly chosen permutation of $1, 2, \ldots, n$. What is the expected time to do the insertion? To simplify things, just like in Quicksort, we're only going to count comparison operations. Whatever we insert first will be the root, requiring zero comparisons. Suppose it was i that got inserted first, as the root. Then, every subsequent insertion will start off being compared with i, followed by either recursing in the left subtree ($i-1$ of them do this) or in the right subtree ($n-i$ of them do this). This gives us the following natural recurrence for the expected number of comparisons:

$$T(n) = (n-1) + \frac{1}{n}\sum_{r=1}^{n}(T(r-1) + T(n-r))$$

which is, surprisingly, almost exactly the same recurrence we saw for Quicksort! We can solve and upper bound it like so, using the exact same tricks from the quicksort recurrence (clear denominators, subtract subsequent terms, and then divide to get a clean recurrence):

$$\begin{aligned}
T(n) &= (n-1) + \frac{1}{n}\sum_{r=1}^{n}(T(r-1) + T(n-r)) \\
&= (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \\
nT(n) - (n-1)T(n-1) &= n(n-1) + 2\sum_{i=0}^{n-1} T(i) - (n-1)(n-2) - 2\sum_{i=0}^{n-2} T(i) \\
&= (n-1)(n-(n-2)) + 2T(n-1) \\
&= 2n - 2 + 2T(n-1) \\
nT(n) &= 2n - 2 + (n+1)T(n-1) \\
T(n) &= 2 - \frac{2}{n} + \frac{(n+1)T(n-1)}{n} \\
\frac{T(n)}{n+1} &= \frac{2}{n+1} - \frac{2}{n(n+1)} + \frac{T(n-1)}{n} \\
S_n &= \frac{2}{n+1} - \frac{2}{n(n+1)} + S_{n-1} && \text{(Defining } S_n \coloneqq \frac{T(n)}{n+1}\text{)} \\
&= \sum_{i=2}^{n}\left(\frac{2}{i+1} - \frac{2}{i(i+1)}\right) \\
&= \sum_{i=3}^{n+1}\left(\frac{2}{i} - \frac{2}{(i-1)i}\right) \\
&= 2H_n - 2H_2 - \sum_{i=2}^{n}\frac{2}{i(i+1)} \\
&\leq 2H_n \\
T(n) &\leq 2H_n(n+1) = \Theta(n\log n),
\end{aligned}$$

Where we start the second sum at $i = 2$ because $T(1) = S_1 = 0$.

So it takes time $\Theta(n \log n)$ to do the n randomly-ordered insertions. Not too bad.

Why is it so similar to quicksort? Well, whatever you insert first is going to be the root. You can think of that like the first pivot you pick in quicksort. Then, any subsequent insertion smaller than that is going to be handled by the left subtree, while any insertion bigger than that is going to be handled by the right subtree, and the left/right subtrees don't interact. So it's exactly like what happens when you recurse in quicksort.

(Maybe draw the tree and compare it to a recursion tree for quicksort.)

In fact, an analysis similar to the one you'll do on your homework this week for Quickselect tells you that, with very high probability, you expect the depth of the tree to be at most, say, $100 \log n$.

Roadmap

We've seen that the standard BST insertion routine is actually not horrible, *if you insert in a random order.* But in practice that isn't a good assumption, and this naïve BST implementation will break down quickly. So we'll spend the next week or so learning about ways to modify this basic BST idea to *guarantee* that the tree doesn't get too deep. There are lots of ways to do this, all with their own tradeoffs (kind of like sorting!).

2.7.4 Wrap Up

So we've seen:

- ...

Now, let's go on to the practice quiz!

2.7.5 Post-Lecture Ed Notes

Some folks were trying to work a bit more on understanding how to translate comparison sorting algorithms into decision trees. Unfortunately, it's a bit hard to manually draw out decision trees, especially for the slower algorithms when you have a large input size.

So in case it helps, here's some Python code that shows how to do this in a general way, i.e., you give it any comparison-based sorting algorithm written in Python and it turns it into a decision tree. You could use this either to visualize the before/after of the transformation, or if you want to dig into the code itself you could try to see how it works. I've demonstrated how to use it with a few different sorting algorithms.

Interestingly, it also lets you see exactly what the built-in Python sorting function is doing!

There was a question about how to implement something like the `search_or_insert` routine we saw pseudocode for in lecture. I indicated that you could use double pointers to translate it naturally. In the notes for today I have included an example C program that compiles and runs and uses that approach. Working through why it works may be instructive (let me know of any bugs if you find them!).

On the topic of the comparison sorting lower bound, there's another way to think about the proof we saw using entropy. Roughly, it goes like this: there are $n!$ many possible inputs, and by the time you return from your sorting algorithm you must know exactly which of those inputs you got. But to describe which of the $n!$ different inputs you are given requires $\log n!$ bits of information. Each comparison gives you at most 1 bit of information, so it lets you peek at at most 1 bit of the $\log n!$ bits specifying what the input. Hence, you need $\log n!$ comparisons at least. You can view the proof we saw in lecture as making this rigorous, or you can also use the language of information theory to make this notion of "bits of information" more rigorous directly. Unfortunately I don't yet have a great reference that does so in this context yet.

If you're interested in other lower bounds, here's a classic but still almost-approachable paper (at least the first few pages) from Andrew Yao while he was at Stanford: `https://dl.acm.org/doi/pdf/10.1145/322261.322274`

If you're interested in reading about non-comparison-based sorting algorithms, maybe start here: `https://en.wikipedia.org/wiki/Counting_sort` and then go on to read `https://yourbasic.org/algorithms/fastest-sorting-algorithm/`.

You might also be interested in sorting networks: `https://en.wikipedia.org/wiki/Sorting_network`

Name: _____ **Stanford ID Number:** _____

2.7.6 Lecture 7/5 Practice Quiz Question

Here is the BST insertion code from lecture:

```
def search_or_insert(root, v):
    if root is empty: # NOT FOUND, insert it
        node = new node
        node.value = v
        replace root with node
        return node
    if v == root.value:
        return root
    if v < root.value:
        return search_or_insert(root.left, v)
    if v > root.value:
        return search_or_insert(root.right, v)
```

Show the tree that results if you start with an empty tree and make the following insertions in this order: 5, 1, 2, 6, 7, 8.

2.8 Regarding the Sorting Code

Now that we're wrapping up sorting, wanted to post some rough + ready implementations of the different sorting algorithms for you all to play with. I definitely suggest implementing them yourselves without looking at anyone else's first, but in case anyone has gotten stuck on implementing any one of them up here you go:

Some things to note:

- I added new base cases to merge and quicksort that turn the problem over to insertion sort once the problem gets small enough. You can try tuning this cutoff for your machine. This is a pretty common/popular implementation choice.
- Even without using insertion sort as a base case, mergesort runs about 4x faster on a sorted list than an unsorted list on my machine! It's interesting to think of explanations for this; the main two that I have involve the branch predictor and the fact that a sorted list gets you more quickly out of the first, "slow" loop in merge and into the three "fast" loops at the end of merge. (In fact, the compiler turning those three fast loops in the end into a 4-way vectorized memcpy would seem to perfectly account for this speedup.)
- Heapsort is consistently the slowest out of all the "good" sorts.
- My implementation of quicksort is often a bit slower than my implementation of merge sort, especially on mostly sorted arrays.
- On arrays with only $\Theta(n)$ inversions, insertion sort beats my system's standard sort function by 5–10x.
- If you increase the size to something like 1024*1024 then you start to get a slightly clearer pattern where Quicksort clearly outperforms merge on random arrays. But at that size, it takes way to long to run selection/insertion sort!
- The way we're getting random numbers is actually pretty bad/not really random, see, e.g., `https://stackoverflow.com/questions/49878942/why-is-rand6-biased` for some discussion.
- Changing the optimization level and compiler produces some interesting effects too!

There's a ton of things you could do to play around with this code, including: optimize it, try to find bugs, try different input distributions, try sorting things bigger than integers, try implementing them for different input types (linked lists), etc. Go wild !

2.9 Week 2 Problem Sheet

2.9.1 Lecture 7/1

Quicksort and k-Selection

Problem 10: Give pseudocode showing how to modify the partition function to do a *three-way partitioning*, i.e., partition the list into values less-than, equal-to, and greater-than the pivot. Describe how this can be used to speed up quicksort when the array has many duplicate elements.

Hint: Hint(s) are available for this problem on Canvas.

Problem 11: (CLRS) Consider an arbitrary execution of the quickselect k-selection algorithm. Say a partitioning during that execution is *helpful* if either (i) the randomly chosen pivot happens to be in the middle 50% of the array (i.e., at least a quarter of elements are smaller than it and a quarter are larger), or (ii) it is the last partitioning performed before the execution ends. Answer the following questions:

1. What is an upper bound on the expected number of partitionings before a helpful partitioning occurs?
2. Let n be the original number of elements in the list and suppose there are m helpful partitionings. Define X_0 to be the number of partitionings up to (and including) the first helpful partitioning, X_1 the number of partitionings after that up to (and including) the second helpful partitioning, etc. Prove that the total time taken by this execution of quickselect is at most $Cn\left(\sum_{i=0}^{m} X_i 0.75^i\right)$ for some constant C.
3. Prove that the expected time for quickselect is $\Theta(n)$.

Hint: Hint(s) are available for this problem on Canvas.

2.9.2 Lecture 7/3: Heaps, heapsort, Sorting Stability

Heaps

Problem 12: In lecture, we only saw relatively high-level pseudocode for the heap operations and for heapsort. This is because we tried to be agnostic to how the heap was implemented, only at the very end discussing how you could implement it as a single contiguous array.

Rewrite the pseudocode from lecture to explicitly operate on a heap stored in a contiguous array. *(It's up to you how detailed you want to get with this, but I suggest going all the way and implementing an executable version of heapsort in Python.)*

Hint: Hint(s) are available for this problem on Canvas.

Problem 13: Recall in lecture we showed how to create a heap from an arbitrary binary tree by repeatedly `sink`ing nodes on the last layer, then the second last, etc.

Answer the following questions, under the additional assumption that the initial binary tree is *perfect*, meaning every layer is fully filled in with nodes.

Hint: Hint(s) are available for this problem on Canvas.

Problem 14: (SW) Design an algorithm to merge k distinct sorted lists, for any integer k. If across all the k lists there are a total of n elements, your algorithm should take time $O(n \log k)$.

Now, use this primitive to explore the concept of a k-way mergesort algorithm, where you split the array into k chunks rather than in half. What is the running time of this k-way merge sort, in terms of both k and n? What happens in the extreme case when you pick $k = n$?

Hint: Hint(s) are available for this problem on Canvas.

Problem 15: Compare heapsort to the $\Theta(n \log n)$ variant of selection sort from the first homework.

Problem 16: Revisit all the sorting methods we've learned so far (selection, insertion, merge, quick, heap). Which of them are stable sorts? For the ones that *are* stable, describe a typo some careless programmers might make in their implementation that would make it *no longer* a stable sorting algorithm, even though it still sorts. For the ones that are *not* stable sorting algorithms, determine whether there is a simple "fix" to make any of them stable and explain any tradeoffs that come with your fix.

2.9.3 Lecture 7/5: Comparison Lower Bounds, Intro to Searching + BSTs

Problem 17: Consider the following highly unreasonable, but pedagogically useful scenario: you wish to write a program to find the user's ranking of ice cream flavors, but the user isn't able to keep track of all n items at once. Instead, the most you can ask of them is to pick their favorite ice cream flavor out of $\sqrt{n}$ possibilities (where the possibilities are chosen by you). Answer the following questions:

1. Prove a lower bound on the number of such questions you have to ask the user.
2. Prove that lower bound is tight by describing an algorithm with that number of queries (it's fine for this to be asymptotically tight, i.e., to just have the same big-Theta).
3. Repeat your lower bound analysis, but for the case where you get not just their favorite, but their ranking of those $\sqrt{n}$ flavors.
4. Does your lower bound in part (3) necessarily imply there exists an algorithm with that worst-case number of rankings?

Hint: Hint(s) are available for this problem on Canvas.

Problem 18: *(This is an intentionally open-ended question, and different students will go different directions with it. Hence, we probably won't be able to give as much conccrete guidance on this problem in OHs compared to other problems.)*

We're about to embark on a two-plus week exploration of the *search* problem, which involves storing a dynamically updated set of items supporting insertion, deletion, and search operations. Research, then compare and contrast the following simpler ways of storing such a set:

1. Contiguous array (ordered or unordered),
2. Singly linked list (ordered or unordered),
3. Doubly linked list (ordered or unordered).

For each one, you might think about the average, best, and worst-case time complexity for each of the three operations (insert, delete, search). For practice, I suggest computing *exact* average case times. You can also think about questions like, how much extra memory is used? What assumptions do

you need to make about the elements being stored? Other topics you might want to research include: interpolation search, intrusive linked lists, sentinel nodes, and the xor trick for doubly linked lists.

Problem 19: Given *any* BST T containing the numbers $1, 2, \ldots, n$, prove that there exists some reordering of $1, 2, \ldots, n$ where, if you INSERT the numbers in that order using the BST insertion routine seen in lecture, it creates the BST T.

Hint: Hint(s) are available for this problem on Canvas.

Problem 20: *(This problem is significantly more involved than others on this problem set; we suggest leaving it for last and referring to the hints on Canvas if we don't cover rotations on Friday.)*

In some (vague) sense, binary search trees mostly make use of the left-to-right direction (your left descendants must be less than you, who must be smaller than your right descendants). By contrast, max heaps only really make use of the up-or-down direction (your parent must be bigger than you). In this problem, we'll show how to combine the two and use it to create a very simple balanced tree.

Suppose instead of storing individual items in each node, you want each node to store a *pair* of items. We'll represent a pair as an object, e.g., x, with two fields, $x.a$ and $x.b$.

Show how, if you give up on balancing the tree, you can store a dynamically updating set of pair-objects $x_1, x_2, \ldots, x_n$ with the following properties:

1. The binary tree should form a *binary search tree* with respect to the a field, namely: If x appears in the left (right) subtree of y, then $x.a \leq y.a$ ($x.a \geq y.a$).
2. The binary tree should form a *heap* with respect to the b field, namely, if x is an ancestor of y, then $x.b \geq y.b$.

Your answer should describe algorithms for inserting and removing items from this tree, and you should describe the worst-case running time in terms of the depth of the tree. (The data structure you will likely reinvent is called a *Cartesian tree*.)

Then, answer the following questions:

1. In class, we saw that inserting $1, 2, \ldots, n$ in a random order led to a good expected balance. Suppose you inserted the objects $x_1, x_2, \ldots, x_n$, where $x_i.a = i$ and $x_i.b = i$, into your Cartesian tree in a random order; what would the expected depth of the tree be? Compute the *exact* probability that your tree has that depth.
2. Suppose you implement a binary search tree in the following way:
   (a) Whenever the user requests to insert a value k into the BST, generate a random number r and insert the object x with $x.a = k$ and $x.b = r$.
   (b) BST deletion is the same as your Cartesian tree deletion.

   What can you say about the balance of the resulting tree? What can you say about the expected running time of the BST operations? (This data structure is known as a *treap*.)

Hint: Hint(s) are available for this problem on Canvas.

2.10 Study Guide for First Quiz (Sorting)

While we don't expect any changes, everything in this document is tentative and subject to change by announcement from the instructor.

Topics

The following topics are in-scope:

1. Working rigorously with the definition of $O(\ldots)$, and working informally (but correctly) with $\Theta(\ldots)$ and $\Omega(\ldots)$.
2. Meaning of worst-case, best-case, average-case, and expected-case algorithm analyses.
3. Selection sort, insertion sort, merge sort, quicksort, quickselect, and heapsort.
4. Divide-and-conquer design technique and analyses.
5. Decision trees and the comparison sorting lower bound.

In particular, the first quiz will *not* ask about loop invariants or BSTs, but you should be comfortable doing correctness proofs for recursive functions without loops (like the mergesort correctness proof from lecture, assuming without proof the lemma that merge is correct).

Having done the homework questions might be useful in answering quiz questions, but we will write the quiz assuming you haven't done the homework, so you don't have to memorize, e.g., the definition of inversions, if you don't want to.

Question Templates

While there may be a few exceptions, we'll try to stick to the following question templates for the quiz:

1. Some kind of big-O practice problem that involves expanding definitions and applying 103 proof techniques, e.g., prove "if $f(n) = O(g(n))$ and $g(n) = O(k(n))$, then $f(n) = O(k(n))$." (See week 1 homework problem 1.1.)
2. Question(s) proposing a slight modification to one of the algorithms we've seen in lecture, then asking things like: Is it still correct? What is the effect on (best or worst or average or expected) running time? (See the 6/24 lecture extra credit.)
3. Question(s) where some type of input is described and we ask things about the behavior (e.g., running time) of a particular algorithm on that type of input. (See the 6/26 lecture extra credit).
4. Some kind of algorithm design problem using divide-and-conquer. (See the 6/28 lecture extra credit.)
5. At most one question where you will have to reproduce essentially the comparison sort lower bound, except in a different setting and/or with a different branching factor. (See the week 2 homework problem for that lecture.)
6. Question(s) proposing a scenario and then asking you to compare + contrast different sorting algorithms for that scenario. E.g., if you are running on an embedded device with limited memory, would heapsort or mergesort be a better choice? (See week 2 homework for the heapsort lecture.)

Questions will be easier than the average homework question, and probably a little bit harder than the average post-lecture EC credit. The motivation for having so many questions is to ensure that no one question has an outsized impact on your grade.

Chapter 3

Searching

3.1 Lecture 7/10 Notes: AVL Trees

Nice job everyone on the exam. We'll try to get it graded by this weekend, and I'll try to get a solution manual together by Friday. I sampled some of the exams and it seems like overall people did great!

Let's continue our exploration of binary search trees.

3.1.1 Warning about binary tree pseudocode

Note that with binary trees there are lots of different ways to implement them and even more ways to implement operations using them. If you're curious, the BST lecture notes have a C implementation of search/insert that I quite like using double pointers. But you can do the same thing without double pointers. Some BST operations become easier if you have a parent pointer, but that takes extra space. And for many of the operations we'll see today, it's a lot easier to understand by seeing diagrams rather than a list of complicated pointer manipulations.

So, when we talk about BSTs, we're usually going to use pretty high-level pseudocode. Basically, the rules we'll try to follow are:

1. Operations that require iterating through more than a finite number of pointer operations will be shown explicitly via recursion or a `for` loop, but
2. Operations that involve reading from a modifying a finite/bounded number of nodes we'll explain in words and/or pictures.

One example was the BST search/insertion routine from Friday: the searching part was shown explicitly via recursion, but the actualy hooking things together was just written down, not told explicitly how to manipulate the pointers.

3.1.2 Tracking Height With BST Insertion

Let's do a little warmup exercise here: tracking the height of each node in a tree while you insert into the tree. You basically do the normal BST insertion, except now you update the heights on the way up.

First, let's look at a more general BST insertion routine that lets us do things "on the way up:"

```
1 def search_or_insert(root, v):
2     if root is empty: # NOT FOUND, insert it
3         node = new node
4         node.value = v
5         replace root with node
6         postinsert(node)
7         return node
```

```
8       if v == root.value: return root
9       child = root.left if v < root.value else root.right
10      node = search_or_insert(child, v)
11      postinsert(root)
12      return node
```

Now, what should we do on the way up? Every node visited might have had its height change, so we'll update the height:

```
1   def postinsert(node):
2       # CONVENTION: height of empty node is zero
3       node.height = 1 + max(node.left.height, node.right.height)
```

3.1.3 AVL Balance

On Friday we saw that pretty much all the operations we wanted to do on trees required traversing the tree from the root down to some node, hence their time was bounded by the number of nodes in the tree. So ideally we'd like to keep our BSTs *complete*, like with heaps! Unfortunately, it's pretty hard to do that; in fact, on the homework you'll prove that, if you want to do insertions while maintaining the completeness of a BST, you need to do $\Theta(n)$ work in the worst case.

In this lecture we're going to study *AVL trees*, which require something a lot weaker than completeness, but which still turns out to be good enough to get us our $O(\log n)$ bounds. The definition of balanced we'll use is as follows:

Definition 13. *A binary tree is* AVL balanced *if at every node the height of the left subtree is at most* ± 1 *the height of the right subtree.*

We call an AVL balanced BST an AVL tree*.*

(Fun fact: according to Wikipedia, AVL is actually only two people, Georgy Adelson-Velsky and Evgenii Landis!)

When talking about complete heaps, we noted that a complete tree has the nice property that its depth is $O(\log n)$. In fact, AVL trees have the same property!

Theorem 18. *An AVL tree with n nodes has depth $O(\log n)$.*

Proof. (Adapted from TAOCP) We're going to start by flipping the problem on its head and ask the following question: what's the fewest number of nodes we can put in an AVL-balanced tree of height h?

Well, call that number C_h and pick, for each h, a tree T_h "witnessing" it. Now, the tree T_h must have either its left or right subtree be depth $h-1$. WLOG assume it's the right subtree, and notice if it's anything other than T_{h-1} we can replace the right subtree with T_{h-1} without increasing the number of nodes or affecting the AVL balance property of the whole tree. Similarly, if the left subtree is anything other than T_{h-2} we cane replace it with T_{h-2}. So, in other words, we have the recurrence:

$$C_h = 1 + C_{h-2} + C_{h-1}.$$

with base cases $C_1 = 1$ and $C_2 = 2$. This is very similar to the Fibonacci sequence, and in fact, you can pretty much complete the analysis here if you know the closed-form solution for the Fibonacci sequence. But that's a bit outside the scope of this class, so let's see an alternative way to get a bound on C_h.

What we're going to do is describe two more sequences, C'_h and C''_h, where it will be easy enough to see the following chains of inequalities:

$$C_h \geq C'_h \geq C''_h \geq 2^{\lfloor h/2 \rfloor}.$$

Namely, first take C'_h to be defined by the same base cases and the recurrence

$$C'_h = C'_{h-2} + C'_{h-1}.$$

We can see that $C_h \geq C'_h$ because all we've done is removed some additions from the ultimate sum (you can prove this more rigorously by induction). Then, take C''_h to be defined by the same base cases and the recurrence

$$C''_h = C''_{h-2} + C''_{h-2} = 2C''_{h-2}.$$

You can prove by induction that $C'_h \geq C''_h$. Handwaving, the reason is that C''_h is *increasing*, so replacing C''_{h-1} with C''_{h-2} only makes the sum smaller.

Finally, the substitution method shows us that, at least when $h = 2k$,

$$\begin{aligned} C''_h &= 2C''_{h-2} \\ &= 2^2 C''_{h-2(2)} \\ &= 2^3 C''_{h-2(3)} \\ &\vdots \\ &= 2^{k-1} C''_2 \\ &= 2^k. \end{aligned}$$

So now following the chain of inequalities, we've seen

$$C_h \geq 2^{\lfloor h/2 \rfloor}.$$

Cool! So what we've shown is that an AVL tree of height h has at *least* $\sqrt{2}^h$ nodes (modulo rounding/assuming h is even).

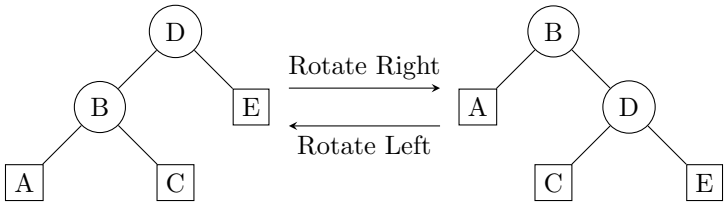
Suppose for sake of contradiction some AVL-balanced tree has n nodes but *more than* $\log_{\sqrt{2}} n$ height. Well, because it has *more than* $\log_{\sqrt{2}} n$ height, it must have *more than* $C_{\log_{\sqrt{2}} n} = \sqrt{2}^{\log_{\sqrt{2}} n} = n$ nodes, which is a contradiction. Hence, by contradiction we see that AVL trees have height $O(\log n)$.

(For ease of exposition we've glossed over corner cases here and above when n is not a power of $\sqrt{2}$; this situation is similar to how we assumed the input was a power of two in the analysis of merge sort. We've also not explicitly shown that C_h is strictly increasing, nor have we shown rigorously that $C''_h \leq C'_h \leq C_h$. All of those steps can be proven more rigorously by induction.) □

Now, there's a lot of other definitions of balance we could have looked at, and you'll see some on the homework. But it turns out AVL balance is a very natural one in part because of the connection to the Fibonacci numbers and also because insertion can be done very efficiently while maintaining AVL balance (as we're about to see!).

3.1.4 Manipulating BSTs

In heapsort, the key operation we used to fix up a heap was to swap a child node with its parent. In BSTs, the key operation we'll use over and over again is the *rotation*. There are two fundamental rotations, the left rotation and the right rotation. They can be seen below:



Notice that the nodes have changed positions, yet they stay in the same relative order! So the left and right rotations *preserve the BST property while changing its shape.*

In fact, you'll see on your homework that these two operations are actually very special; we didn't just pick them arbitrarily. They're special because using just these two operations, you can make *any* other BST-property-preserving modification to the tree you want!

3.1.5 AVL Insertion

Now, suppose you have an AVL-balanced tree and you go to insert a node. *The resulting tree is not necessarily AVL-balanced any more!* So, on the way back up from the insertion, you need to check whether it's balanced and fix it up if it's not:

```
def postinsert(node):
    # CONVENTION: height of empty node is zero
    node.height = 1 + max(node.left.height, node.right.height)
    if abs(node.left.height - node.right.height) > 1:
        determine which case the tree falls into
        perform either 1 or 2 balancing operations
        update the heights
```

To fix it up, there are a lot of different cases. We're not going to explicitly think through all of them, although we will think through the two "unique" ones (the rest are essentially the same, just symmetric over left/right). Knuth gives diagrams for both. Basically, in one of the cases you need a single rotation. In the other case, you need a double rotation.

The rotations can be summarized as follows:

1. If the insertion happened "to the right of the right of the unbalanced node" (a "zig-zig"), just rotate at that unbalanced node to bring the inserted item towards the root by one.

2. If the insertion happened "to the left of the right of the unbalanced node" (a "zig-zag"), rotate the inserted item to the root twice, once at the child of the unbalanced node and once at the unbalanced node itself.

There are a number of symmetric cases to consider, but those two cases give the core idea.

Something interesting to notice: in either case, the tree after the rotation now has the same height it did before! So once you do a rotation, you actually don't need to update heights or do any more rotations on the way up (but it doesn't really hurt asymptotically to do so). This raises a bit of a mystery which is, how does the height of the tree ever grow then?? (Answer: it grows in the other cases, when we don't need to do a rotation at that node because the balance factor hasn't gotten too far off.)

3.1.6 Time Complexity for AVL Trees

AVL trees require us to visit each node on the way back up. On each of those we do at most a constant number of rotations (in fact we do at most two total!). So in total we do at most a constant number of operations per insertion more than the normal BST insertion, so it's still $O(depth)$.

But thanks to the very first theorem we saw, we know that the depth of an AVL tree is always $O(\log n)$, so our operations take time $O(\log n)$! *(Recall for AVL tree's we're only going to see search/insert, but you can also do delete in $O(\log n)$ time — that's out of scope for this class.)*

3.1.7 Memory-Efficient AVL Trees

It's a bit unfortunate that the AVL tree data structure requires us to use up a whole extra field on all our nodes for the height. Turns out, you don't actually need to know the height! You can just store the *balance factor* of the node, i.e., the difference between its right and left subtree heights. Then, on the way back up, you keep track of the height of the node you're currently at and add or subtract the balance factor to figure out the height of the other node.

There are only 3 possible balance factors, so storing it requires only 2 bits. It turns out on almost all modern computers pointers actually only use up, say, 48-bits of the 64-bit word size. So you can use up 2 of the unused 16 bits in one of the child pointers to store the balance factor, which essentially makes AVL trees 'space free.'

3.1.8 Deletion in AVL Trees

We're not going to talk about deletion in AVL trees, but apparently it's possible (just a lot more casework). One interesting thing about AVL tree deletion is that you may have to do $\Omega(\log n)$ rotations in the worst case, so it doesn't have the surprising property of insertion that $O(1)$ rotations are required.

3.1.9 Post-Lecture Ed Notes

Also, just want to reiterate that by far the best resource I know of for this is Knuth 6.2.3. We basically covered the first 3–4 pages of that chapter in lecture. Ah, and it's worth noting that Knuth calls them just "balanced" trees, whereas we refer to them as "AVL trees" or "AVL-balanced BSTs." I think the latter terminology is much more common, since there are many different notions of 'balance' for binary trees.

There was a question about doing the AVL insertion without using $\Theta(\log n)$ extra space for the call stack. See below for a better way to do this, but I described briefly one hack to accomplish this by "turning the pointers around" as you go down the tree. If you're interested in learning more about it, I learned the hack from Knuth 2.3.5, where he uses essentially the same idea for garbage collection when going from Algorithms B/C to Algorithms D/E (this sort of space saving is especially important in the context of garbage collection because you can't assume the heap graph is shallow, and also you generally only garbage collect when you run out of space so your garbage collector should try to avoid requesting even more space!!).

There was a question about what should happen if there are multiple sources of unbalance. This cannot happen if you're only inserting a single item into an AVL tree. But if you have a totally unbalanced BST that you then want to turn into an AVL tree, you may run into such a case. One thing you could do, if your starting tree is a BST, is read all the items off in sorted order then use the result of Problem 5 on this homework to insert them into an AVL tree in time $O(n)$. It seems likely to me that you could also do a heapify-like approach, where you fix the tree in a bottom-up fashion, or first find the median and bring it to the root, etc. But I haven't thought through those approaches. If you come up with something that works, feel free to post about!

We said briefly that if you know the closed form solution to the Fibonacci sequence, then you can more directly get the bound we needed for C_h. Wikipedia has a nice description of how to derive that closed-form solution here: `https://en.wikipedia.org/wiki/Fibonacci_sequence#Relation_to_the_golden_ratio` Meanwhile, Knuth 1.2.8 derives the same relation using generating functions. So feel free to check those out if you're interested! I think that approach gives you a slightly tighter bound than the $\sqrt{2}^n$ that we saw; the bound you get from the Fibonacci sequence should be ϕ^n where $\phi \approx 1.6$ vs. our result with $\sqrt{2} \approx 1.4$.

Unfortunately, one issue with the reduce-to-Fibonacci-sequence approach is I'm not sure how to use it to understand the weakened variants of AVL balance, e.g., as you'll do in HW Problem 3 (whereas I think the approach we saw in lecture does work for those).

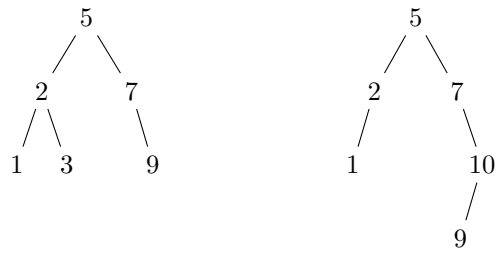
Oh BTW: I think the hack I described for avoiding the $O(\log n)$ space overhead is really overkill; you can observe that, during the first pass down the tree, you know exactly whether or not that node will become unbalanced after doing the insertion. So you just need to keep track of the last such node encountered on the way down, and jump back up there after the insertion to do the rotations (then you can do another pass down from there to fix up the heights). I believe something like this is what Knuth's algorithm does; if you're interested in this question I highly recommend studying his algorithm A in 6.2.3.

Oh, and since this recursion-requires-stack-usage-only-sometimes thing keeps coming up, wanted to link this here: `https://en.wikipedia.org/wiki/Tail_call` which explains why the compiler can turn recursion into loops in some specific scenarios but not others.

Name: _____ **Stanford ID Number:** _____

3.1.10 Lecture 7/10 Practice Quiz Question

Part 1: Circle the BST below that is an AVL tree, and explain why the other isn't.



Part 2: For the one that *is* an AVL tree, show the result of inserting 8 using the AVL insertion procedure we saw in lecture. If multiple rotations are needed, clearly draw what the tree looks like after each rotation.

3.2 Lecture 7/12 Notes: 2–3 Trees, B-Trees, and Red-Black Trees

3.2.1 HW2 Reflection Thoughts

Lecture:

- Seems like lectures are getting a bit harder for people, but still an OK pace. Definitely suggest coming to OHs.
- try to get a study guide for quiz 2 out earlier
- continuing issues with recurrence relations, and the comparison sorting lower bound
- stability could have used more time dedicated to it, especially with examples.
- pseudocode starting to get harder

HW:

- people liked the suggested problems
- seems like this HW was a bit harder, but still within the range of reasonableness. will try to keep it here, at least for the 'suggested' problems.
- people liked the more concrete pseudocode/code problems

3.2.2 Lecture 7/12 Notes: 2–3 Trees, B-Trees, and Red-Black Trees

There are three closely connected types of trees that people often talk about:

1. Red–black trees,
2. 2–3 trees, and
3. B-trees.

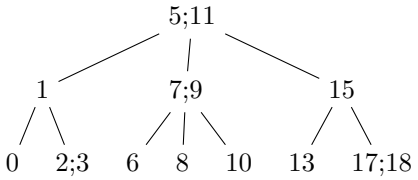
Roughly speaking, 2–3 trees are a special type of B-tree and red-black trees are just a special way of representing 2–3 trees. So today we're going to focus on 2–3 trees, which will give us the most bang for our buck, i.e., after learning about 2–3 trees you should be ready to understand both red–black trees and B-trees with little help.

Remember from the HW we have this issue that we can't do a BST with perfect, or even complete, balance. AVL trees got around this by weakening the balance requirement to just AVL balance. 2–3 trees are going to get around this by requiring perfect balance, but weakening the *binary* requirement of BSTs!

3.2.3 2–3 Trees

Definition 14. *A 2–3 tree is a perfectly balanced tree where: every node has 1 or 2 values, every non-leaf node has one more child than values, and it satisfies the generalized search tree property (draw it).*

In the lecture notes we'll draw 2–3 trees like so:



We call every node with two keys a *3-node*, because they 'should' (excluding the leaves) have three children. Meanwhile, we call the other nodes *2-nodes* because they 'should' have two children. *(During certain operations, we'll also end up temporarily breaking the 2–3 tree properties by introducing* 4-nodes *which have three keys and four children, but we'll split those into 3- or 2- nodes before finishing the operation.)*

3.2.4 2–3 Tree Search

Search is basically unmodified from normal BST search, except now we need to pick one from at most 3 possible children to recurse in rather than 2.

```
def search(root, v):
    if v is a value in root:
        return root
    if root is empty or leaf:
        return NotFound
    child = (find child for value v)
    return search(child, v)
```

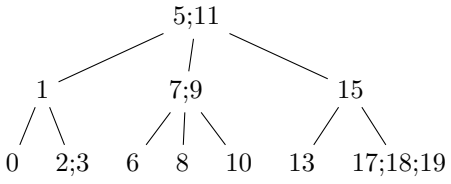
3.2.5 2–3 Tree Insertion

Insertion is where things get more complicated:

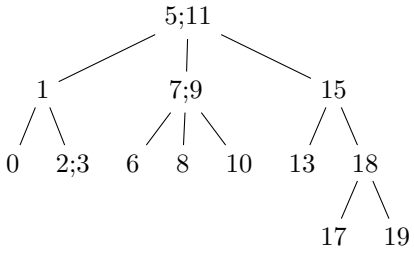
```
def insert(root, v):
    if v in root.values: return
    if root is empty:
        replace root with new leaf node
    if root is leaf:
        add v to root.values
    else:
        child = (find child that v must be in)
        insert(child, v)
        if child split:
            absorb it into root
    if root is now a 4-node:
        split it
```

We do the normal thing of finding where the value 'should' go, except now we stop at a leaf and just add the value to that leaf. If the leaf was a 2-node, we can stop: all we've done is turn it into a 3-node, but the tree stays a perfectly balanced 2–3 tree.

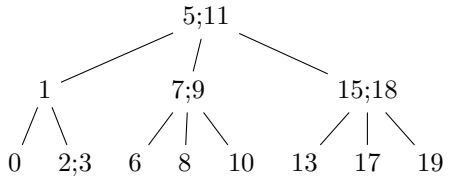
On the other hand, if the leaf was a 3-node and so now became a *4-node*, we somehow need to fix this. We can fix this by *splitting* the node into three separate nodes. For example, if we wanted to insert 19 in the tree before, we would first do an insertion at the leaf, making a 4-node:



Then we would *split* that 4-node into three 2-nodes:



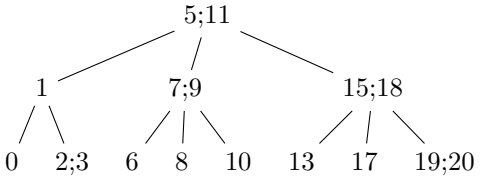
But now the tree is unbalanced! To fix that up, we can *absorb* the 2-node we just created (18) into its parent on the way back up the tree:



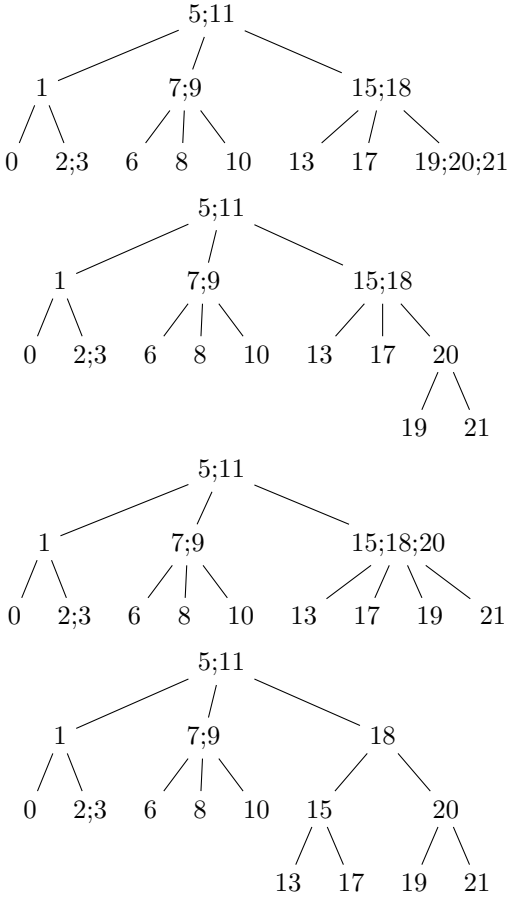
Which produces a perfectly balanced 2–3 tree, as desired!

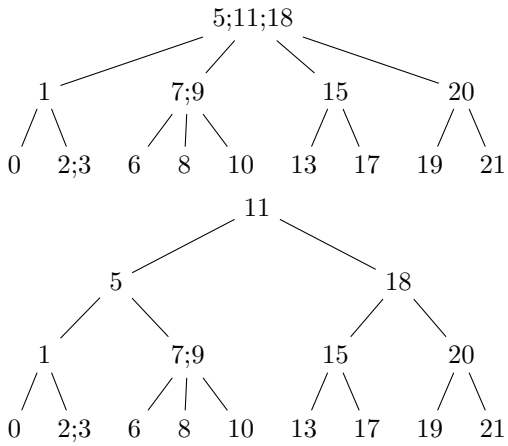
Note that, if the parent node was *already* a 3-node, this absorption would cause an issue because it would create a 4-node at the parent. But this can be handled by doing the exact same thing to the parent: split it into three 2-nodes, then absorb the middle one of those into its parent (the root).

To see this, consider first inserting 20, which requires no split/absorbs:



Then inserting 21, which will trigger multiple operations as shown below:





One interesting thing to note here is that 2–3 trees always grow "at the root", whereas BSTs like AVL trees first grow at the "leaves" and then intermittently rotate those leaves up towards the root.

3.2.6 2–3 Tree Deletion

Deletion in 2–3 trees is a bit harder than insertion. I think Knuth doesn't really cover deletion for 2–3 trees, so our decision is to **not** require knowing 2–3 tree deletion for exams. But, the ability to store a set or map with guaranteed $O(\log n)$ insertion, search, and delete is *so* important to so many different applications, I think it's worth walking through how to do it so you can say you've seen it at least once/you know people aren't making this up!

On one of the HW problems you'll see a really clever way to do deletion via a special *splitting* operation. But for now we're going to take a more direct approach. Also, as in SW, instead of showing full deletion we're going to show *delete-min*. This makes the problem a little bit simpler because we know the min is all the way along the leftmost branch of the tree, and that it's in a leaf. Deleting an arbitrary value is *almost* the same procedure, except that you might have to switch right/left if you take a different branch, and you'll have to do the swap-with-successor trick to make sure you're always deleting from a leaf.

Without further ado, here's some pseudocode:

```
def delete_min(root):
    if root is a 2-node:
        squash or rotate to make it a 3- or 4-node
    if root is leaf:
        remove minimum value from root
    else:
        delete_min(root.left)
    if root is a 4-node:
        split it

```

The basic observation is this: if the leaf we're deleting from is a 3-node, then we can just remove the value and everything is fine (all we've done is change it to a 2-node). But if it's a 2-node that we're deleting from, suddenly that node would now become empty, messing with the perfect balance of the tree. So, *on the way down to that node*, you should manipulate the tree so that every node is always at least a 3-node before recursing on it (you may need to make the root a 4-node, but all the others should be fine). Then, on the way back up, you may find that the root is still a 4-node, in which case you can split it back into 2-nodes.

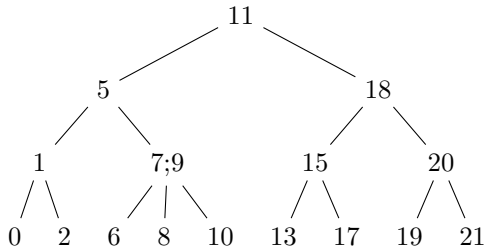
Sedgewick and Wayne give a few cases that are needed for this. I think they all boil down into the following two:

- *(The Squash)* If your first two children are 2-nodes, squash them together with your smallest value to create a 4-node. (Note if you are the overall root, this may shrink the height by 1.)

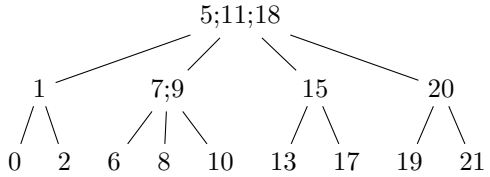
- *(The Rotate)* If your left child is a 2-node but its immediate sibling is a 3-node, you can "borrow" the smallest value from the right child, use that to replace your smallest value, and then give your smallest value to your left child. Note that the child pointers need to change a bit too. This process is very reminiscent of rotating a BST.

Note that none of these operations ruin the balance of the tree. The only thing that might happen is your root might become a 4-node, but we can split that at the very end if it's still there.

This is easiest to see with an example. Consider this tree, which is very similar to the one we saw above:

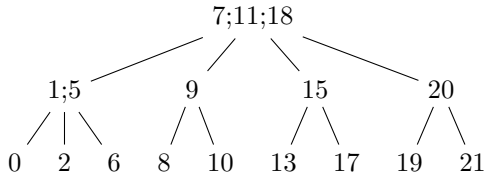


Suppose we wanted to delete the min (zero) from this tree. The root is a 2-node, so we first need to use one of the operations to make it into either a 3- or 4-node before continuing. In this case, the operation that works is to squash the first two layers together into a big 4-node:

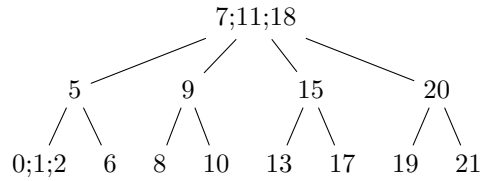


making a 4-node root.

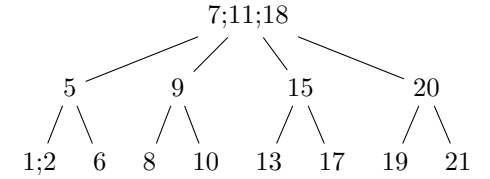
Now, before we recurse onto the node 1, we need to make it a 3-node. To do this, we can do essentially a rotation which allows it to "borrow" the 7 from its sibling; really, the parent borrows the 7 and then the node in question borrows the 5 from the parent.



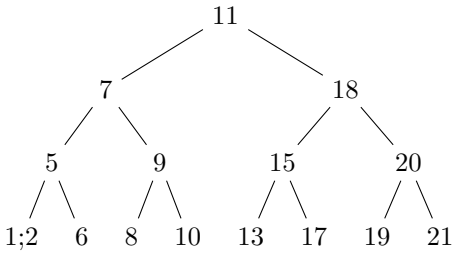
Now we can recurse into the 3-node "1;5." Before recursing to the leaf 0, we need to make it into a 3-node. Here, we can't borrow from the sibling (2) because it's a 2-node. What we can do is squish 0 and 2 together after taking the value 1 from their parent, making a 4-node:



Finally, we can now recurse into the 0;1;2 leaf and delete the min:



On the way back up, we may run into some 4-nodes that we created. To handle them, you essentially unsquash, i.e., reverse the operations you did that created them in the first place.



3.2.7 Generalizing to B-trees and Red-Black Trees

At the start, we said once you understand 2–3 trees you can easily understand both B-trees and red-black trees. We're going to briefly explain these here. For an exam, we don't expect you to know anything about B-trees, but we do expect you to know how to convert between 2–3 trees and red-black trees.

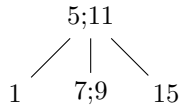
B-trees

A B-tree is just like a 2–3 tree, except instead of 2–3, it's a "$\lceil t/2 \rceil$–$\lceil t/2 \rceil + 1$–$\cdots$–t" tree. For any choice of the parameter $t \geq 3$ you can write down similar insertion and deletion algorithms to the above. Note there is some disagreement among authors about the *exact* definition of a B-tree, e.g., some people only accept 2–3–4 trees as B-trees, not 2–3 trees. But those differences are not particularly important.

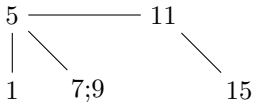
Red-Black Trees

You can think of red-black trees as exactly the same as a 2–3 tree, except with a clever way of storing the 2–3 tree as a colored binary tree. Specifically, we *split* every 3-node into two 2-nodes, and then color the first of those 2-nodes red to indicate that it should be interpreted as "squished together with" its parent.

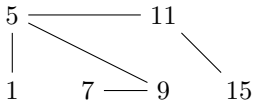
It is easy to visualize like so. Suppose we start the following 2–3 tree:



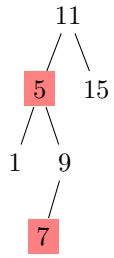
We can split the 3-nodes using *horizontal* edges. If we do this to the root we get:



(Note there is some ambiguity here about whether to give the middle child to the left or to the right; we're going to use *left-leaning* red-black trees, which will always give the node to the *left*.) If we do this again to the 7;9 node, we get:



(Note here again there was ambiguity about whether to point 5 to 9 or 7; in a left-leaning red-black tree we're always going to point the parent to the rightmost node in such a horizontal group.) Finally, if we are careful to color the left-side nodes in each horizontal edge a special color (red), then we can "let all the edges fall" while being able to recover the original tree:



This process then gives us a colored BST that contains exactly the same info as the original 2–3 tree. To recover the original 2–3 tree, you just collapse the red nodes into their parents to create 3-nodes. In fact, because every RB tree is equivalent to a 2–3 tree, you can rewrite the 2–3 tree operations (insertion, deletion) to work on RB trees! The situation here is pretty similar to how you can do heaps on contiguous arrays for heapsort. But for the sake of this class, all that we want you to be able to do is (1) convert between 2–3 trees and RB trees, and (2) do insertions on 2–3 trees.

3.2.8 Post-Lecture Ed Notes

Fun fact I forgot to mention: Sedgewick (of SW fame!) co-invented the red-black tree!

For 2-3 trees and red-black trees, the reference we're going to use is Sedgewick and Wayne (though Knuth also has pretty good coverage of this). In particular, I wouldn't suggest using CLRS for this.

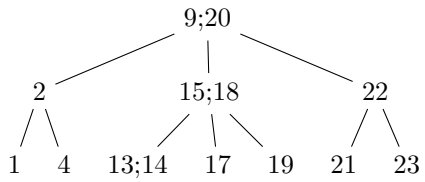
There was a question about why one would use red-black trees over AVL trees. After doing some Googling, it seems like the answer is very complicated. You might start reading here: `https://cs.stackexchange.com/questions/41969/why-are-red-black-trees-so-popular` Overall, it seems like B-trees (with much larger max branching factor than 3) are by far the best performing, but they're a bit problematic because you have to move values around a lot so they're difficult to predict how iterators will invalidate and also you can't do intrusive B-trees. Between AVL and RB trees, it seems like there are a lot of tradeoffs between search, insert, delete time. At that point, it's probably easiest to just try both in your application and see which is best. Worth noting that GCC's std:map seems to be implemented as an RB tree.

If you finish problem 6 and are looking for a challenge, try repeating problem 6 but instead of requiring the tree be complete, require it be just "full", i.e., every layer except the last is completely filled in (but it's OK for the last layer to not have all its nodes on the left). The approach in the hint doesn't work any more, but it would be nice to see either another such bound showing it's not possible to maintain fullness, or an algorithm that does insertions in $O(\log n)$ time while maintaining fullness. (I don't know the answer !)

Name: _____ **Stanford ID Number:** _____

3.2.9 Lecture 7/12 Practice Quiz Question

Draw the result of inserting 10 into the following 2–3 tree.



3.3 Week 3 Problem Sheet

Some of these problems require you to draw trees. You're welcome to do this in LaTeX (check out the forest package) or you can hand-draw them and add photos of your drawings to the LaTeX document using INCLUDEGRAPHICS.

3.3.1 Lecture 7/10

Tree Rotations and AVL Trees

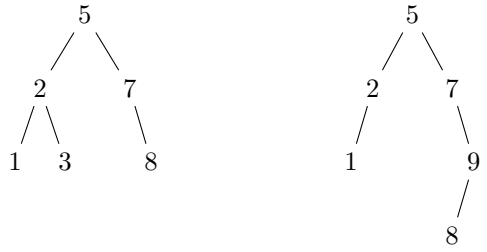
Problem 21: Prove that the two rotation operations we saw in class (left and right) are *complete*, meaning, any valid BST T_1 storing the numbers $1, 2, \ldots, n$ can be turned into any *other* valid BST T_2 storing the same numbers using only a sequence of those two rotation operations.

Hint: Hint(s) are available for this problem on Canvas.

Problem 22: (Adapted from SW) Write a recursive procedure to check whether a given binary tree is an AVL tree (both a BST and having the AVL property). It should have total time complexity proportional to the number of nodes in the tree and use extra space proportional to the depth of the tree. You can assume the tree has no duplicates, but you cannot assume that nodes in the tree are annotated with their height (you need to compute the height).

Problem 23: Consider weakening the definition of an AVL tree to the following: for any node, the height of the left subtree is within $\pm k$ of the height of the right subtree, for some constant $k \geq 1$. Prove that any such tree with at most n nodes never has depth larger than $O(\log n)$.

Problem 24: Which of the following is an AVL tree? Explain your answer. For any of the below that is an AVL tree, show the result of inserting 40, 20, 30, and then 10, in that order.



Problem 25: *(Knuth 6.2.3.21) (This problem is more difficult than it looks; I suggest finishing other problems before starting it.)* Suppose your user is giving you an increasing sequence of numbers $x_1 < x_2 < x_3 < \cdots$ one-by-one. Describe an algorithm for accepting each number and maintaining an AVL tree on the entries seen so far. Every time you are given a new number your algorithm should do $O(1)$ work. At any point in time, the user may request to see the AVL tree for all the numbers they've given you so far — you shouldn't have to do any extra work to show them this tree, rather you should keep the tree always up-to-date.

Note: just running the AVL insertion routine for each number seen is not good enough, because it takes $O(\log n)$ time on each insertion. You must make use of the fact that the numbers are increasing. Be careful with the data structures you use!

Hint: Hint(s) are available for this problem on Canvas.

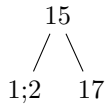
Problem 26: *(This problem is difficult. I suggest finishing other problems before starting it.)* Heaps had the interesting property that it was possible to implement them efficiently while making sure the heap remains a *complete* binary trees (all levels except the last are completely filled in, and the last level has all elements moved as far to the left as possible). On the other hand, AVL trees allow significant imbalance in the tree; in fact, most AVL trees are not complete. Prove that it is impossible to keep a normal binary tree complete under arbitrary insertions without frequently needing to do $\Omega(n)$ work per insertion.

Hint: Hint(s) are available for this problem on Canvas.

3.3.2 Lecture 7/12

B-Trees and Red-Black Trees

Problem 27: Draw how you would represent the following 2–3 tree as a red-black tree (note we're using "1;2" to notate a node with two keys/values inside of it, i.e., a 3-node). Then, on the 2–3 tree, show the result of inserting the numbers 0, 5, 10, and 20, in that order.



Problem 28: Your friend tells you they came up with an amazing new data structure where insertion, deletion, and find-max all take at most $O(\log \log n)$ time. Prove that your friend is either lying, or hiding some restriction of their algorithm that makes it less general than normal BSTs.

Hint: Hint(s) are available for this problem on Canvas.

Problem 29: *(This problem is difficult; I suggest finishing other problems before starting it.)* (Modified from CLRS 18-2) This problem asks you to make more powerful 2–3-tree operations than just insert and remove. In all of the below, you can assume you've modified your 2–3 trees to keep track of the height of every node.

1. Design an algorithm for *joining* two 2–3-trees T_1 and T_2 at the element x, i.e., if all the elements satisfy $T_1 < x < T_2$, make a new 2–3 tree containing all the elements in T_1 and T_2 along with the element x. If the two trees have the same height, your algorithm should run in time $O(1)$. Otherwise, your algorithm should take time proportional to the difference in height of the two trees.

2. Design an algorithm for *splitting* a 2–3-tree T at a node x, i.e., return a 2–3-tree T_1 having all elements less than x and a 2–3-tree T_2 having all elements greater than x. Your algorithm should take time $O(\log n)$.

3. Explain how to implement insert and delete using the join and split operations above.

4. Describe how to use 2–3-trees to implement a list data structure with the following operations, all taking $O(\log n)$ time: (i) find-ith-element, (ii) split one list into two at index i, and (iii) concatenate two lists.

Hint: Hint(s) are available for this problem on Canvas.

3.4 Lecture 7/15 Notes: Amortized Running Time

(This lecture follow CLRS chapter 16 pretty closely; that's the best reference for this out of the textbooks used this quarter.)

We've been talking about searching and binary search trees for the past few lectures. On Wednesday we're going to introduce you to the *splay tree*, which is one of the absolute best (and one of the simplest!) balanced binary search tree algorithms out there. Interestingly, splay trees have pretty bad worst-case time. On the other hand, they have good *amortized* time. Before we get there, however, we're going to take a little detour, and spend today's lecture trying to figure out what that means on some much-simpler examples.

3.4.1 Motivating Example for Amortized Analysis

Before rigorously defining amortized analysis or proving things about it, we're going to spend some time talking through a motivating example to explain why it could be useful, i.e., why an algorithm with good worst-case amortized time might still be useful even if it has bad worst-case nonamortized time.

Imagine you're writing a compiler, and you need some data structure to keep track of all the functions in the program being compiled. Something like a binary search tree would work: you want to be able to insert a function into the data structure when it's declared, and then search in the data structure for that function whenever it's used.

Let's suppose the compiler designer is considering two possible data structures, with the following properties:

1. Data structure A guarantees the *worst-case* time per operation is $O(\sqrt{n})$.
2. Data structure B has *worst-case* time per operation only $O(n)$, but it guarantees that a *sequence* of n operations never takes more than $O(n \log n)$ time.

Which should they use?

Well, the compiler is essentially performing a sequence of operations on the data structure; let's say it does n operations. In data structure A, we know that *each one of those operations* takes $O(\sqrt{n})$ time, for a total of $O(n\sqrt{n})$ time. Meanwhile, in data structure B, we know there might be a few of those operations that take $O(n)$ time, but *overall*, adding them all up, the total time taken by the data structure during compilation is only $O(n \log n)$!

The key here is that, *from the user's perspective*, all the care about is the total time it takes to finish the compilation. If you tell the user, "oh, but aren't you upset this one data structure operation in the middle took so long for version B?" they'll think you're crazy and point out that, even though that one operation took a long time, so many other operations were so much faster that the total time was much shorter!

So in this scenario it seems quite reasonable to use data structure B, which provides a better amortized time, even though its worst-case time is worse.

3.4.2 Definition of Amortized Runtime

Amortized analysis is a pretty broad term that essentially means "analyzing the cost per operation averaged over a sequence of operations." For today, we'll work with the following pretty reasonable definition:

Definition 15. *An algorithm has* worst-case running time $O(A(n))$ amortized over n operations *if every sequence of n operations starting from an initial state takes time $O(nA(n))$.*

We may sometimes drop the "amortized over" statement and just say something like, it has amortized time $O(A(n))$.

This definition (based on the one in CLRS) is pretty good, except perhaps that it ignores any concept of input size. For the example(s) we'll see today, this is reasonable because the data structure will always start off empty and operations will be used to fill it up one at a time, so the input size is naturally bounded in the number of operations.

In more complicated situations, though, you might run into statements like: *if the input to each operation is size m, then the operation has time $O(m^2)$ amortized over m^3 operations,* which means: for any m, a

sequence of m^3 operations each one having input size m takes time $O(m^5)$. Similarly, a statement like: *starting with a tree of size n, search takes $O(1)$ time amortized over $\log n$ operations*, would mean: performing a sequence of $\log n$ search operations on a tree of size n takes total time $O(\log n)$. *(Note these are made up examples to explain what amortization means; not claiming any algorithms have those time guarantees!)*

3.4.3 Our Running Problem for Today

Before we talk about the running example we'll use for today, it's important to understand something about how computers work. Most computers have a *flat address space*: you can basically think of memory as one huge array, with all your data in different spots. When you create a *contiguous array* of size n, you're basically asking the operating system (saying operating system but often it's really your programming language runtime's allocator handling your immediate request) to find a block of n unused slots in that big huge array to reserve as belonging to you. The problem is, what happens if you want to increase the size of that contiguous array? The operating system might have put other things next to your array, assuming that you'd only use n slots! So you have to ask the operating system for a new region of memory, of bigger size, and then copy everything over there, but this takes $O(n)$ work!

This leads to one of the most classic amortized algorithms problems: the *growing dynamic array*.

Definition 16. *The* growing dynamic array *problem is to maintain a contiguous array of elements with the operation* Push *for growing the array by one.*

Crucially, the dynamic array data structure *does not* know ahead of time how many things are going to be in the array! So it doesn't know how big of an array to allocate ahead-of-time.

The algorithm we'll be looking at is basically the one that we described above: when too many items are inserted, ask the OS for more space and copy everything over:

```
def push(array, value):
    n = array.n_elements
    s = array.size_of_allocation
    if n == s:
        new_data = (request region of size 2*(n+1))
        for i = 0, 1, ..., n - 1:
            new_data[i] = array.data[i]
        (release array.data back to operating system)
        array.data = new_data
        array.size_of_allocation = 2*(n+1)

    array.data[n] = value
    array.n_elements = n + 1
```

The crucial thing to note is that, when we ask the OS for more space, we ask for twice as much space as we really need.

3.4.4 "Brute-Force Approach" to Amortized Analysis

The most immediate way to go about doing an amortized analysis is just to think about a sequence of n operations and how long it could possibly take to complete all of them in a row. CLRS calls this the "aggregate analysis" method.

Before we get started, let's look at what happens after the first few insertions. In the below table, n is the number of elements in the array after that insertion, s is the size of the allocated region, and t is the time to perform that operation. C is some large constant that depends on exactly how the code is implemented and how you measure running time.

Push Value	x_1	x_2	x_3	x_4	x_5	x_6	x_7	$\cdots$
n	1	2	3	4	5	6	7	$\cdots$
s	2	2	6	6	6	6	14	$\cdots$
t	$\leq C \cdot 1$	$\leq C$	$\leq C \cdot 3$	$\leq C$	$\leq C$	$\leq C$	$\leq C \cdot 7$	$\cdots$

More generally, consider a subsequence $x_i, x_{i+1}, \ldots, x_{i+k}, x_{i+k+1}$ where x_i and x_{i+k+1} trigger resizes, but $x_{i+1}, \ldots, x_{i+k}$ all have enough space. Notice in particular that, after x_i, the array will have size $2i$, hence the second resize will only be triggered on the $i + k + 1 = 2i + 1$ insertion, i.e., $k = i$.

We will now compute the average time per operation *just in the subsequence* $x_i, \ldots, x_{i+k}$. The x_i push requires $\leq Ci$ time, while all the others require $\leq C$ time. So the total time for that subsequence is

$$T_{i,\ldots,i+k} \leq Ci + \sum_{j=1}^{k} C = Ci + Ck = Ci + Ci = 2Ci.$$

And there are $k + 1 = i + 1$ operations in that subsequence, hence

$$\frac{T_{i,\ldots,i+k}}{k+1} = \frac{2Ci}{i+1} \leq 2C = O(1).$$

Now we're almost done. Notice we can partition the entire sequence into such subsequences, and within each subsequence the average time per operation is bounded above by $2C$. But the following lemma tells us the average time per operation across the *whole* sequence should be bounded above by this per-subsequence bound, completing the proof:

Lemma 2. *Let P be any finite (multi)set of numbers, and $P^{(1)}, \ldots, P^{(k)}$ be some partitioning of that set. Then,*

$$\frac{1}{|P|} \sum_{x \in P} x \leq \max_i \frac{1}{|P^{(i)}|} \sum_{x \in P^{(i)}} x.$$

Proof. Define $M \coloneqq \max_i \frac{1}{|P^{(i)}|} \sum_{x \in P^{(i)}} x$. Because the $P^{(i)}$s partition P, we have

$$\frac{1}{|P|} \sum_{x \in P} x = \frac{1}{|P|} \sum_i \sum_{x \in P^{(i)}} x = \sum_i \frac{|P^{(i)}|}{|P|} \left(\frac{1}{|P^{(i)}|} \sum_{x \in P^{(i)}} x \right) \leq \sum_i \frac{|P^{(i)}|}{|P|} M = M \frac{\sum_i |P^{(i)}|}{|P|} = M \frac{|P|}{|P|} = M.$$

□

In total, then, we've partitioned the steps into subsequences and shown that the average within each partition is at most $2C$, hence by the lemma above, the average across the entire sequence is at most $2C$, i.e., $O(1)$. This proves the theorem:

Theorem 19. *The dynamic array push operation takes amortized time $O(1)$.*

3.4.5 The Potential Method

The direct method helps make it clear what we're doing, but when we get to more complicated algorithms it becomes very difficult to track everything that's going on across operations in a sequence. The method that's most used in practice is called *the potential method*, and you can see it as a simplified template for doing aggregate analysis. (Demo in lecture with cube.)

In the potential method, you define a *potential function* Φ which takes as input the state of your data structure after some operation in the sequence and produces as output a number. Using the potential method involves proving things about how the potential changes during the operation. The following is the key theorem of the potential method:

Theorem 20. *Let s_0 be the initial state of a data structure and $s_1, \ldots, s_n$ be the states after any sequence of n operations.*

Suppose you can prove the following:

1. $\Phi(s_i) \geq \Phi(s_0)$ *for all* i.
2. *For any operation in the sequence, if* t_i *is the time actually taken by the algorithm then* $t_i + (\Phi(s_i) - \Phi(s_{i-1})) \leq f(n)$.

Then, the amortized time per step is $O(f(n))$.

Proof. The total time of the whole sequence is

$$T \coloneqq \sum_{i=1}^{n} t_i,$$

so it suffices to show that $T \leq nf(n)$.

But in fact:

$$nf(n) = \sum_{i=1}^{n} f(n) \geq \sum_{i=1}^{n} (t_i + (\Phi(s_i) - \Phi(s_{i-1}))) = T + (\Phi(s_n) - \Phi(s_0)) \geq T.$$

□

Using the potential method, we can make a much simpler proof for the dynamic arrays amortized bound:

Theorem 21. *The dynamic array push operation takes amortized time* $O(1)$.

Proof. Note there is a constant C such that the time taken by the algorithm when i items need to be copied is $\leq C(i+1)$.

Then, we will take the potential function to be $2Ck$, where k is the number of items in the second half of the array.

We can see that $\Phi(s_0) = 0$ and it remains nonnegative, so the first condition is met.

For the second condition, there are two cases:

1. On a normal push, we do $\leq C$ work and increase the potential by at most C, hence for those steps $t_i + (\Phi(s_i) - \Phi'(s_{i-1})) = O(1)$.
2. If x_i is a resize push, we need to copy i items, which takes $\leq Ci$ time. But our starting potential in this case is exactly $2C\frac{i-1}{2} = C(i-1)$, so in total $t_i + (\Phi(s_i) - \Phi(s_{i-1})) \leq Ci + (0 - C(i-1)) = C = O(1)$.

In either case the desired inequality is shown, so we can conclude $O(1)$ amortized time per operation. □

3.4.6 Post-Lecture Ed Notes

There was a question about the etymology of "amortize." Turns out, it does indeed come from the same "mort" ("death") as "mortgage", with the idea that you're "killing off a debt" slowly over time. In this context, the "debt" is the infrequent worst-case operations, and "paying off the debt" means making other operations faster than the average time to compensate for those longer-running worst-case operations.

CLRS Chapter 16 is the best reference I know for amortized analysis, and it has a lot of great examples beyond just the dynamic table one. In addition to the aggregate (direct) method and the potential method, they describe the accounting method which I think of as a special case of the potential method (instead of having a single black-box potential function, you associate parts of the potential with different objects in your data structure). It has a nice analogy with money; I suggest reading that section if you have time.

Another important thing to note: in the CLRS they actually change the definition of amortized time throughout chapters 16.1, 16.2, and 16.3. E.g., in 16.3 they define the amortized time to be what we called $c_i + (\Phi(s_i) - \Phi(s_{i-1}))$, and then on the side from that they argue if Φ never drops below its initial value then this definition of amortized time overapproximates the true time when summed over n timesteps. I find this changing definition confusing, which is why we didn't change the definition in lecture. Instead, for this class, we're going to use one consistent definition of amortized time (average time per operation averaged over a sequence of n operations) and think of the potential method as a way to prove things about the amortized time indirectly, rather than redefining amortized time to be dependent on a potential function. But it's important to know that this is something some authors do (i.e., define amortized time with reference to a potential function) when you're reading papers, etc.

Name: _____ **Stanford ID Number:** _____

3.4.7 Lecture 7/15 EC Question

Suppose instead of doubling the size of the array every time we resize, we just increased it by one. What would be (i) the worst-case and (ii) the amortized time per insertion with that modified algorithm?

3.5 Lecture 7/17 Notes: Splay Trees and Union-Find

Today we're going to do something unusual for this class: we'll describe two algorithms (splay trees and union-find) *without* proving anything about their running time. The reason is that these algorithms are very useful, so you should know about them, but the proofs for their running time are, unfortunately, far too complicated for lecture. Thankfully, you'll at least see on the homework: (1) the proof of the splay tree amortized time, and (2) the proof of a worst-case time for a modified version of the union-find algorithm.

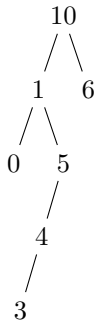
3.5.1 Splay Trees

A *splay tree* is really just a normal BST: when we say splay tree, we don't mean anything special about what the tree *looks like*, what we mean is just that whenever we access the tree we use the special splay tree operations.

Splay trees are simultaneously one of the best and simplest balanced binary search tree algorithms we'll see in this class.

The key intuition behind splay trees is this: *whenever you access a node, move it up to the root.*

Example tree:



When we access 3, we'll use rotations in a special way to move it up to the root.

There are at least two reasons you might want to do this:

1. Good *average time*: if your input distribution involves accessing 3 very frequently, then it will keep 3 near the top of the tree to speed up the average access time.

2. Good *amortized time*: remember in amortized time we're OK with a few worst-case operations, but we want to avoid *sequences* of worst-case operations. If we left 3 where it was, you would have a sequence of worst case operations by just repeatedly looking up 3. By moving it up towards the root, you rule out this possibility, making it a lot harder for someone to force you into a sequence of worst-case operations.

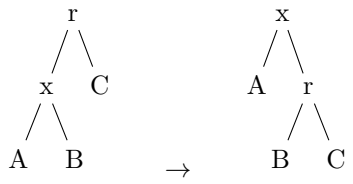
Splaying

The key operation is the *splay*, which is how we bring a node up to the root:

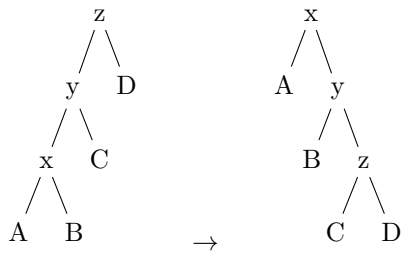
```
def splay(node):
    while node is not root:
        if node is zig from root:
            do splay-zig rotation to move node to the root level
        if node is zig-zig from grandparent:
            do splay-zig-zig rotation to move node up two levels
        if node is zig-zag from grandparent:
            do splay-zig-zag rotation to move node up two levels
```

Note that the rotation for zig-zag is the same as the AVL zig-zag rotation, but for zig-zig it's a bit different. The three rotation types are shown below.

1. If r is the root *of the entire tree* and we're trying to splay its immediate child x, this looks like

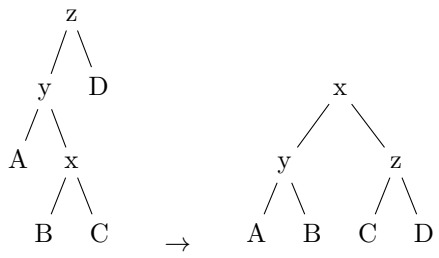


2. If the path to x involves a zig-zig step, we rotate like so:



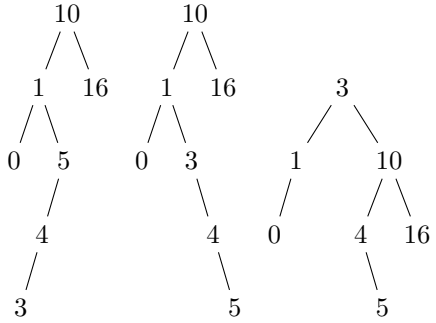
   Which you can implement as first rotating right around z, then rotating right around y. *(Note: surprisingly, this is different than doing the rotations in the other direction!)*

3. If the path to x involves a zig-zag step, we rotate like so:

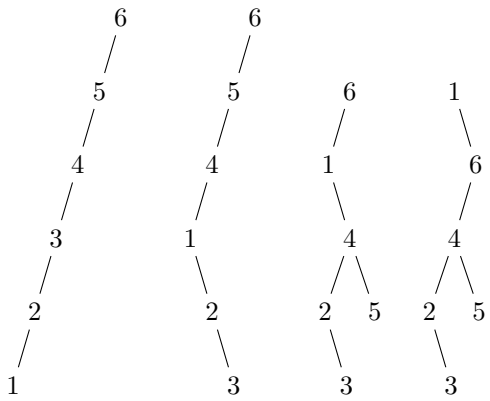


   Which you can implement as first rotating left around y, then rotating right around z. *(Note: surprisingly, this is different than doing the rotations in the other direction!)*

Let's see what happens when we splay 3.



It's also instructive to see what happens when we splay the leaf 1 in this tree:



Splay Tree Operations

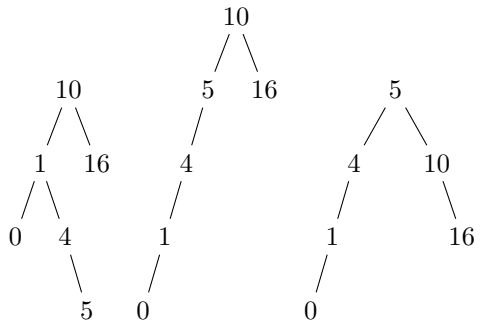
You can use splaying to make splay tree versions of most BST operations.

Search for Node The search routine looks for that value, then makes sure to splay whatever that last node it sees is:

```
def splay_search(root, v):
    if root is empty: return
    if v == root.value:
        splay(root)
        return root
    child = root.left if v < root.value else root.right
    if child is empty:
        splay(root)
        return NotFound
    return splay_search(child, v)
```

Note: there's something subtle here, which is that we splay a node *even if we don't find what we're looking for!* It turns out this is pretty important to the amortized time bound.

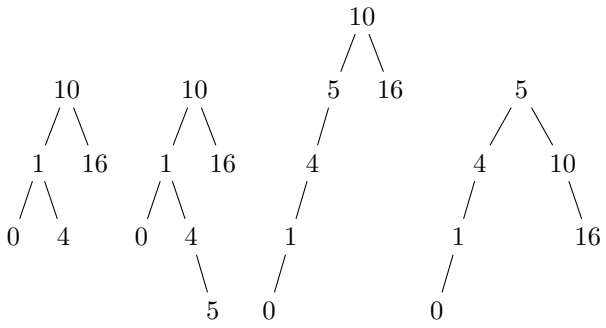
Example: searching for 5



Insert a Value To insert a value in a splay tree, do the normal BST insertion and then splay the resulting node.

```
def splay_insert(root, v):
    new_node = normal_bst_insert(root, v)
    splay(new_node)
    return new_node
```

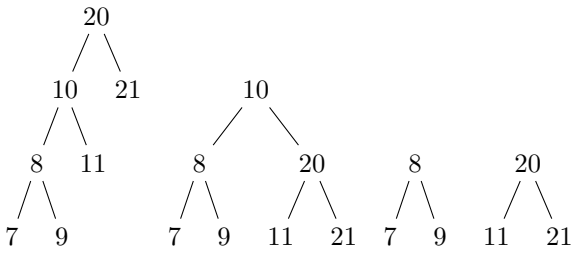
Example: inserting 5



Split at a Node The *split* method takes a value v in the splay tree and then breaks the tree into two splay trees, one containing all nodes with values (strictly) less than v and one containing all nodes with values (strictly) greater than v. To do so, we find the node with value v, splay it up to the root, and then remove it: the two parentless children are now roots of the two trees we want!

```
def splay_split(root, v):
    root = splay_search(root, v)
    left, right = root.left, root.right
    root.left, root.right = empty, empty
    return left, right
```

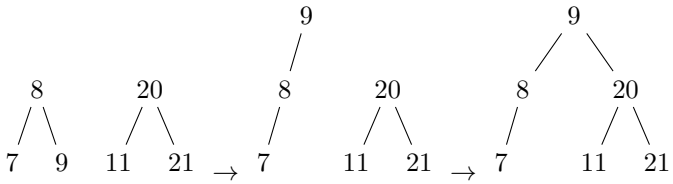
Example: splitting 10



Join Two Trees On the other hand, if the values in T_1 are all smaller than the smallest value in T_2, we can *join* the two trees into one by splaying the rightmost node T_1, then connecting the second tree as a right child of that node.

```
def splay_join(root1, root2):
    root1 = splay(find_max(root1))
    root1.right = root2
    return root1
```

Example: join the trees 7/8/9 and 11/20/21 from above:



Delete a Value To delete a value from a splay tree, first split the tree at that value and then join the two resulting trees back together.

```
def splay_delete(root, v):
    smaller, greater = splay_split(root, v)
    return splay_join(smaller, greater)
```

Example: see the previous two examples; they deleted 10!

Splay tree time bounds

Theorem 22. *The amortized time per operation for a sequence of n of the above splay tree operations is $O(\log n)$.*

Proof. The proof is too complicated for lecture/exams, but you'll see most of it on the homework! □

(There are more interesting things you can say about the *average* time for splay tree accesses. We won't get into that this quarter, except to point out that if you access the same element multiple times in a row, it will stay at the root so searches will be faster.)

Notably, if the tree starts in any possible configuration, it's possible for the *worst-case* time per operation to be $\Theta(n)$, but the above amortized bound still holds.

Splay Tree Implementation Considerations

The pseudocode we've given above is a *bottom-up* splay implementation. It turns out in practice it's often better to do the splaying *top-down*, while you search. That also makes it easier to do many of the above operations without parent pointers; in fact, with this optimization, splay trees are the most space-efficient balanced BST we've seen! We won't worry about such optimizations in this class.

3.5.2 Union-Find

We're going to talk about one more algorithm today, the *union-find* algorithm (sometimes called *disjoint sets*).

Here's a fun, but not super practically useful motivating example. Consider a group of people, maybe: Zach, Jason, Andrew, Bob, Mary, Kate, and Steve. Suppose we know, for each person, who their friends are. We want to know if there are any *friend islands*, i.e., groups of people who are all friends with each other (or friends of friends of . . . of friends) but not friends with anyone else outside of their island.

Our approach will be the following: start by assuming everyone lives in their own island. Then, iterate through all the people. For each person, *combine* their island with the islands of each one of of their friends.

(TODO: insert example)

We need a data structure for keeping track of the islands with the ability to join two islands. We might also, e.g., want the ability to check if two people belong to the same island or not, or to list everyone in a particular island. The idea behind union-find is to store them in a *forest of trees*, with each person being a node and having a pointer to its parent in the tree.

Given a node in the tree, we can find the root of the tree it's in:

```
def find_root(A):
    if A.parent is empty:
        return A
    A.parent = find_root(A.parent)
    return A.parent
```

And given two nodes in the tree, we can join their clusters into one:

```
def union(A, B):
    A = find_root(A)
    B = find_root(B)
    if A == B:
        return
    if B.weight < A.weight:
        A, B = B, A
    # now A is smaller
    A.parent = B
    B.weight = B.weight + A.weight
    return B
```

We can, e.g., check if two nodes are in the same cluster like so:

```
def same_set(A, B):
    return find_root(A) == find_root(B)
```

Then our islands pseudocode is:

```
def find_islands(people):
    for person in people:
        for friend in person.friends():
            union(friend, person)
```

Two things about these code snippets are important:

1. *Path compression:* When we find the root, we update all of the nodes along the way to point directly at the root, so if we ever run find-root on any of them again, we'll find the root much more quickly.
2. *Join-by-size:* when we connect together two islands, we make the larger one the root.

With these two optimizations, the following amortized time bound holds:

Theorem 23. *A sequence of n union-find operations takes $O(\alpha(n))$ amortized time, where $\alpha(n)$ is the* inverse Ackermann function.

Proof. Out of scope for this class. You'll see a proof of $O(\log n)$ worst-case per-operation time on the homework. □

The inverse Ackermann function is a really fascinating function, but, for the most part, outside the scope of this class. For this class, all you need to know is that $\alpha(n)$ grows *extremely* slowly: if you live in the real world, you can assume $\alpha(n) \leq 5$.

3.5.3 Post-Lecture Ed Notes

One thing I realized was a bit ambiguous in the lecture pseudocode while doing this: "`normal_bst_insert`" in the pseudocode should return the newly inserted node (in the C code above it turned out to be easier to have it return the overall root of the tree, and then do a separate `splay_search` step to find it — the two have identical behavior, except the C code does an extra search down the tree so it's about twice as slow).

If you're interested in Splay trees, the original paper is quite a reasonable read, especially with all the work we've been doing to understand the potential method: `https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf`

There was a proposal relating to move-to-root trees, which came before splay trees. You can read about those here: `https://dl.acm.org/doi/pdf/10.1145/322092.322094`

There was another word-choice question, this time about the word "splay." Merriam Webster says: "splay: to cause to spread outward," which I guess is vaguely what you can imagine is happening to the tree as you drag the node-to-be-splayed up through it? AFAIK the original paper doesn't motivate the term.

Union-find is often used in type inference systems for programming languages. `https://www.cs.cmu.edu/~janh/courses/ra20/assets/pdf/lect03.pdf` seems to discuss this. At a high level, you might imagine if you see a statement like "a = b" and you know "a" has type "pointer to X" and b has type "pointer to Y", where X and Y are unknown type variables, then you should infer X and Y are the same. Union-find can help keep track of sets of types that you have inferred must be the same.

You can read more about the Ackermann function here: `https://en.wikipedia.org/wiki/Ackermann_function` just remember that in union-find we're dealing with its inverse, which is extremely slow-growing.

The "friend islands" problem can be solved slightly faster using DFS/BFS, which we'll explore in the graphs section. The caveat is that the DFS/BFS approach can't as easily handle cases where new friendships are made over time.

(A student asked for textbook recommendations regarding amortized analysis, splay trees, and union find. Below is my response:)

Amortized analysis: I think CLRS is the only one that covers it directly, and they do a pretty good job (their 'accounting method' is worth a read/may help clarify things left vague by lecture). But they define things subtly differently so just be careful to use the lecture notes from class as the 'gold standard' where needed.

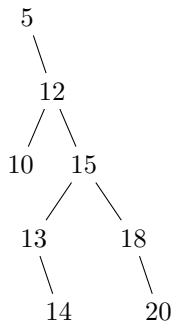
Union-find: both CLRS and SW contain discussions of union-find; I think both are fine, tho like usual I prefer SW. Note both of those describe a bunch of different variants of union-find, e.g., using join-by-height instead of join-by-size, but in this class we'll refer to the variant I wrote on the board (path compression + join-by-size) as just normal "union find".

Splay trees: I don't think any of the textbooks cover them, unfortunately, so you'll have to refer to my lecture notes + anything you can find online. Wikipedia has a pretty extended article: `https://en.wikipedia.org/wiki/Splay_tree` and here's a writeup from a Cornell variant of 161: `https://www.cs.cornell.edu/courses/cs3110/2013sp/recitations/rec08-splay/rec08.html` In fact, if you're willing to put up with some annoying scanning, I found the original paper to be surprisingly readable: `https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf` at least up to Theorem 1, there shouldn't really be anything new/beyond the class, especially if you've done the relevant homework problems.

Name: _____ **Stanford ID Number:** _____

3.5.4 Lecture 7/17 EC Question

Draw the result of splaying the following tree at node 14:



3.6 Lecture 7/19 Notes: Hashing and Risky Hash Sets

These past two weeks we've been thinking about the *search* problem, where you want to store a set of items with the operations search, insert, and delete.

The algorithms we've seen so far, based on BSTs, have been very general; they work for any types of values as long as you can compare them. However, similar to the comparison sorting lower bound, it's possible to do better if we know more about our data!

3.6.1 Direct Addressing Sets

In a BST, when you want to check whether a value is in the BST or not, you have to follow a path through the tree to find out where that value "is supposed to go." This process of searching for "the node where the item should be" is what takes so much time in the BST operations. What if instead we knew ahead-of-time what all the possible values were, and we preallocated spots for all of the possible items? Then when we want to check if an item is in the set, we just directly check its spot. This idea leads to the following *direct addressing* data structure.

Suppose we know that all of the n distinct items we want to store are numbers in some small range $0, 1, \ldots, M-1$. Then the *direct addressing* data structure keeps an array of M *buckets*. We decide ahead of time that value i "belongs in" bucket i — no searching through a tree is needed, we can just go directly to that bucket and see whether value i is there or not.

We can then implement search, insert, and delete like so, where `buckets` is an array of values of size at least M. Initially, the buckets should all have a special 'empty' value in them.

```
def search(buckets, v):
    return buckets[v] == v
```

```
def insert(buckets, v):
    buckets[v] = v
```

```
def delete(buckets, v):
    buckets[v] = empty
```

Example with $M = 10$:

1. Originally we have the buckets $[\perp, \perp, \perp, \perp, \perp, \perp, \perp, \perp, \perp, \perp]$, where $\perp$ is a special value that means "empty".
2. If we insert 3, we have the buckets $[\perp, \perp, \perp, 3, \perp, \perp, \perp, \perp, \perp, \perp]$
3. If we insert 4, we have the buckets $[\perp, \perp, \perp, 3, 4, \perp, \perp, \perp, \perp, \perp]$
4. If we insert 0, we have the buckets $[0, \perp, \perp, 3, 4, \perp, \perp, \perp, \perp, \perp]$
5. If we delete 3, we have the buckets $[0, \perp, \perp, \perp, 4, \perp, \perp, \perp, \perp, 9]$
6. If we insert 9, we have the buckets $[0, \perp, \perp, \perp, 4, \perp, \perp, \perp, \perp, 9]$
7. To check whether 1 is in the set, we look at bucket 1 and see it's $\perp$ so 1 is not in the set.
8. To check whether 4 is in the set, we look at bucket 4, see it does indeed have 4 in it, so it's good to go!

The space usage of this approach is $O(M)$ while the time per operation is $O(1)$.

Problems With Direct Addressing

If we have very few items in a very large set of possible values, e.g., 100 items out of a possible 2^{64}, then this approach won't work very well because the space usage will be unreasonably huge. Also, it essentially only works for items that are numbers, otherwise we can't use them to index into the array of buckets.

3.6.2 Hash Functions and Risky Hash Sets

Today we're going to experiment with an approach that we'll call *risky hash sets* to avoid the space usage associated with counters. They're essentially a worse version of a data structure called a *Bloom filter*, but since it's so much worse, we're going to give it a different name (Bloom filters are out-of-scope for the class).

The basic observation behind risky hash sets is that, if you have *way* more buckets than items, almost all of those buckets are going to be empty. So we want to try and *collapse* a bunch of those buckets together. This might cause false positives, but (as we'll see) the results will still be "approximately dependable."

Every risky hash set needs a *hash function*, which says how to map "logical buckets" in the big, direct-address set that we imagine happening behind-the-scenes, to the smaller set of buckets that we actually store. For example, if we used the hash function $h(x) := \lfloor x/4 \rfloor$ in the earlier example, we'd only need to store the "hash buckets" $[0, 4, 9]$. Once we've decided on a hash function, we can modify our direct addressing pseudocode to work directly on the hash buckets. The resulting pseudocode we'll call a *risky hash set*.

```
def risky_search(buckets, v):
    return buckets[hash(v)] == v
```

```
def risky_insert(buckets, v):
    buckets[hash(v)] = v
```

```
def risky_delete(buckets, v):
    buckets[hash(v)] = empty
```

Two important things to note before we see an example:

1. Of course, if you try to store more than 3 things in this risky hash set you're going to have a problem! Imagine inserting 8: it will bump 9 out of the bucket, as if it never existed! That's part of the reason we call it *risky*.

2. In fact, you might run into problems even earlier than that if your hash function happens to map multiple nonempty logical buckets into the same hash bucket.

To resolve the first issue, we generally want to make sure we have more hash buckets than items. To resolve the second issue, we want our hash function to "spread the values around evenly," making sure not to, e.g., map all the logical buckets into a single hash bucket (this desired property is where the word 'hash' comes from). In practice, we commonly perform analyses *assuming the hash function is chosen at random from the set of all possible hash functions*. This makes many analyses much easier, but is a potentially inaccurate assumption in practice. We'll talk more about that later today!

Collisions

Of course, this risky hash set is quite risky! If two different values have the same hash, they'll collide in the same bucket, and the first one will get booted out as if it was never inserted in the first place.

We're going to analyze the frequency of such collisions in two ways: First, what is the chance you get no collisions? Second, what is the chance your lookups are correct?

The Birthday Paradox *(This approach is adapted from CLRS Chapter 5.4.)* What we're about to prove is this theorem:

Theorem 24. *If you place n items at random into k buckets, the expected number of (unordered) pairs of items that land in the same bucket is $\binom{n}{2}\frac{1}{k}$.*

First, we're going to take a little detour and imagine a fun scenario. Imagine you're at a party with n total people, and someone makes the following offer: you pay them \$100 to play the game, but once the game starts, they'll give you \$100 for every pair of people at the party that you find where both people in the pair have the same birthday. Assuming the birthdays of the attendees are uniformly and independently distributed, should you take the bet? What will your expected profit be?

Let's start by analyzing the following question: how many people do you need for the expected number of birthday collisions to be at least 1? Let's say the number of people is n and the number of possible birthdays is k (we know $k = 365$ or 366, but we'll leave it as a variable for now for generality). The probability that any two share a birthday is $\frac{k}{k^2} = \frac{1}{k}$ because there are k^2 possible pairs of birthdays they could have and only k pairs of identical birthdays. Now, let's ask: what is the expected number of *unique pairs* of people with the same birthday?

$$\text{E[number of pairs of people with same birthday]} = \sum_{\{p_1,p_2\}\subseteq\{1,\ldots,n\}} \frac{1}{k} = \binom{n}{2}\frac{1}{k},$$

which now finishes the theorem!

Let's revisit the bet. We only want to take the bet if this expectation is at least 1. So we solve

$$\binom{n}{2}\frac{1}{k} \geq 1,$$

which tells us we need

$$n^2 - n \geq 2k,$$

i.e., taking a square root and ignoring constant factors,

$$\Theta(n) \geq \Theta(\sqrt{k}),$$

which tells us that, as long as the number of people is well over the square root of the number of 'days,' we should take the bet!

In particular, going back to the concrete inequality, we can find for $k = 365.25$ we need:

$$n^2 - n \geq 2(365.25),$$

i.e.,

$$n^2 - n - 730.5 \geq 0$$

which we can solve using the quadratic formula to find:

$$n = \frac{1 \pm \sqrt{1 + 4 \cdot 730.5}}{2} \approx \frac{1}{2} \pm 27.$$

So, as long as the party has about 28 people, the expected number of collisions you find is going to be at least one, hence by linearity of expectation your expected profit is $-100 + 100\mathbf{E}\,[\text{number of collisions}] \gneq -100 + 100 \cdot 1 = 0$ dollars.

Warning: In lecture today I described a slightly different bet, for which (as [anonymized student] helpfully + correctly pointed out) the above analysis does not properly apply to tell you when you should/should not take the bet. If you're interested in that nuance, as a mea culpa for that mistake I've added an appendix to the end of these lecture notes that explains this in more detail and showing how you could analyze the original bet. But that's out of scope for this class; the main thing you need to know is the theorem above, that the expected number of collisions is $\binom{n}{2}\frac{1}{k}$.

Birthday Paradox and Risky Hash Sets The birthday problem tells us that, if $n^2 - n \geq 2k$, we'll likely have hash collisions. So if we want to *avoid* having too many collisions, we'll need to ensure $n^2 - n < 2k$ (in fact, it suffices to ensure $n < \sqrt{k}$, which is a simpler inequality we'll use from here on out).

Rehashing and the Less-Risky Hash Set This analysis, along with the idea of a growing array, leads to a natural idea. For ease of exposition, we'll assume here we only need to handle insertions.

1. Keep a list of everything inserted into the RHS.
2. Let n be the length of this list and k be the number of hash buckets. When an insertion causes the length of this list to exceed $\sqrt{k}$:
   (a) Create a new blank RHS with a new hash function and $k' = 4n^2$ buckets.
   (b) Insert every item in the list into the new RHS.
   (c) Switch to using that RHS from now on.

Based on our earlier analysis, this will ensure we never expect more than a constant number of collisions, which is pretty good. But what about the time of doing all these rehash operations?

Turns out, the amortized time is pretty good! Suppose we do a rehash with n_0 items, making a new RHS with $k' = 4n_0^2$ buckets. We won't rehash again until i more insertions, at the point:

$$n + i \geq \sqrt{4n_0^2} = 2n_0.$$

But this takes n_0 more time, hence it's amortized away (same argument as for dynamically growing arrays).

Summary of What We've Done and Where We're Going If you choose an insertion sequence $x_1, \ldots, x_n$ of values, then insert them into the growing RHS with hash functions chosen at random, you are ensured:

1. $O(1)$ amortized worst-case time per operation.
2. $O(n^2)$ space usage.
3. On expectation, at most $O(1)$ collisions at any point in that sequence, i.e., lookups that would report "not in the set" even though the value was inserted.

The $O(1)$ time per operation sounds great, but the other two are quite sad: n^2 space is a *lot* of overhead, and while "fewer than one expected collisions" sounds *good*, what would sound even better is to have *guaranteed* zero collisions. On Friday we'll talk about how to avoid those issues by handling collisions in a smarter way than just booting out whoever was there before.

Picking Hash Functions

Before that, let's wrap up today by talking a little bit more about those magical "hash functions" that are supposed to randomly spread our values around a smaller range.

Let's think about what kinds of values we might have:

1. Strings
2. Integers
3. Machine words (essentially, integers in the range $0, \ldots, 2^{64} - 1$)
4. Etc.

At least the first two of these are *infinite sets*, so there are infinitely many possible hash functions. So it seems pretty hard to truly pick one of those functions out at random, and even if we did so, how would represent it?

The answer is that we don't. This is concerning, because if our hash function *isn't* really chosen at random, the user could be a little bit evil, try to reverse-engineer our hash function, and insert a long sequence of colliding inputs. This would ruin our "$O(1)$ expected collisions" property, and make the RHS data structure arbitrarily bad!

To avoid this scenario, three techniques are frequently used:

1. If you know ahead of time exactly what inputs you're going to insert into the RHS, you can think really hard (or have a computer think really hard for you) to devise a hash function that guarantees no collisions between those inputs.

2. Make the hash function *really, really complicated*, so that no matter how hard they try, it's difficult for an evil user to find two inputs that would map to the same bucket. (This approach would need to be balanced against the speed of the hash function!) Another way to interpret this approach is that "randomness" sort of means "unpredictability," so you can try to make your hash function unpredictable by making it complicated.

3. 'Sprinkle in' just enough randomness to make it difficult for the attacker to reliably cause collisions (frequently called salting the hash function). There are at least two ways to think about this:

   (a) Use one of the "complicated hash functions" above, but append some random string to the end of whatever you're hashing (this is what Python and some other languages do).

   (b) Pick the hash function at random from a *subset* of all the (maybe infinitely many) possible hash functions. It turns out it is possible to do this in a way that actually *guarantees* the same $O(1)$ expected collisions property of a truly randomly chosen hash function; you'll see this on the homework!

Appendix: The Birthday Bet

This section is out-of-scope for the class; it's mainly here to correct a mistake in the way I presented the motivating birthday scenario in lecture.

In lecture, when discussing the birthday problem, I framed the bet this way: if you find *any number of collisions* you get \$100, otherwise you lose \$100 (call this "Bet A" and the one I describe in the notes above "Bet B"). We then saw that once you have more than ≈ 27.5 people at your party, the expected number of collisions (pairs of people with the same birthday) is at least 1. **However,** this alone does not necessarily mean you should take bet A, because the outcome of the bet is not a linear function of the number of collisions (so we can't apply linearity of expectation). For example, if all we know is the expected number of collisions is at least 1, that could hypothetically be the case because there is, say, a 75% chance of zero collisions and a 25% chance of 10 collisions! In that scenario, the expected number of collisions would still be greater than 1, but we **shouldn't** take Bet A (at least, in the long run/if we're OK with risk) because our expected profit is $0.75 \cdot -100 + 0.25 \cdot 100 = -50$ dollars. On the other hand, even in the same scenario, we probably **should** take Bet B, because the expected profit works out to $0.75 \cdot -100 + 0.25 \cdot (1000 - 100) = 150$ dollars. Even though we still expect to lose Bet B most of the time, the much higher payoff when we do win makes it a reasonable choice on expectation.

Analyzing Bet A

In the rest of this appendix, I'll describe an analysis of Bet A. In lecture, I indicated that you should be able to use Markov's inequality to transform the expected value into a useful analysis for Bet A, but I no longer believe that's true (or at least, all the ways of applying of Markov's inequality I've tried don't seem to give anything nontrivial). Instead, I see no way of getting around just directly computing the probabilities of the two outcomes, i.e., of having no collisions vs. having more than zero collisions. So that's what we'll do below.

(Adapted from CLRS Chapter 5.4) Let's directly compute the probability that there are no collisions. To do so, we can imagine iterating through the n people at the party one-by-one, and making sure we have seen no repeated birthdays yet. This leads to the following product, where the term $\frac{k-i}{k}$ captures the probability that the $i+1$th person's birthday doesn't collide with any of the previous i people's birthdays.

$$\Pr[\text{No collisions}] = \frac{k}{k}\frac{k-1}{k}\frac{k-2}{k}\cdots\frac{k-n+1}{k}.$$

Unfortunately, bounding this product is not very elementary.

One approach you can take is to write it as $\frac{k!}{k^n(k-n)!}$ and then apply Stirling's approximation, but Stirling's approximation is generally only stated as an asymptotic result that holds for large n, so it's not immediately obvious how applicable it is in this concrete setting.

Instead, CLRS suggests to use the following lemma

Lemma 3. *For all real x, $1 + x \le e^x$.*

Proof. The fact is immediate for $x \le -1$ because $e^x > 0$. If you know the series definition of e^x, this fact is also immediate for the remaining cases because all higher-order terms in the sum are either nonegative or have decreasing magnitude.

But I prefer thinking of e^x as the inverse of ln, which is itself defined in terms of the area under the curve $\frac{1}{x}$, so let's see how to approach it that way. Because the logarithm is increasing and injective, it suffices to show that $\ln(1+x) \le x$, i.e., $x - \ln(1+x) \ge 0$. Considering the derivative of this quantity, i.e., $1 - \frac{1}{1+x}$ we see that $x - \ln(1+x)$ is decreasing in the range $[-1, 0)$ and increasing in the range $(0, 1]$, so $-\ln(1+0) = 0$ is the minimum of $x - \ln(1+x)$ in the range $[-1, \infty)$, as desired. □

Using this lemma, we can rewrite

$$\begin{aligned}
\Pr[\text{No collisions}] &= \frac{k}{k}\frac{k-1}{k}\frac{k-2}{k}\cdots\frac{k-n+1}{k} \\
&= \left(1 + \frac{-1}{k}\right)\left(1 + \frac{-2}{k}\right)\cdots\left(1 + \frac{-n+1}{k}\right) \\
&\le e^{-1/k}e^{-2/k}\cdots e^{(-n+1)/k} \\
&= e^{\sum_{i=1}^{n-1} i/k} \\
&= e^{-n(n-1)/(2k)}.
\end{aligned}$$

We want this to be *at most* $\frac{1}{2}$, so we solve

$$\begin{aligned}
e^{-n(n-1)/(2k)} &\le \frac{1}{2} \\
-n(n-1)/(2k) &\le \ln\frac{1}{2} \\
-n(n-1)/(2k) &\le -\ln 2 \\
n^2 - n &\le 2k\ln 2.
\end{aligned}$$

Filling in $k = 365.25$, this bound tells us we need just above 23 people at the party to have a 50% chance of winning the bet, so if your party has 24 or more people you should take Bet A in addition to Bet B.

3.6.3 Post-Lecture Ed Notes

A student asked: "As a developer creating risky hash, isnt that randomness a double edge sword for me if I myself cant find the same bucket twice?"

Sorry for any confusion — that randomness is chosen once, at the beginning of the program, then stored (you may also need to rechoose it every time you rehash, if you're doing the growing version). It's not the case that the randomness you sprinkle in changes every time you call the hash function, rather, it changes every time you run a new copy of the program.

If you found the birthday problem interesting, here's a paper from our very own Persi Diaconis on a class of related problems: `https://www.stat.berkeley.edu/~aldous/157/Papers/diaconis_mosteller.pdf`

The "risky hash sets" we saw are most similar to Bloom filters: `https://en.wikipedia.org/wiki/Bloom_filter` the main changes to get to Bloom filters are: (1) use multiple hash functions and (2) just store 0/1 in each bucket, not the value itself.

Of particular interest is the counting bloom filter: `https://en.wikipedia.org/wiki/Bloom_filter#Counting_Bloom_filters` Basically, if in each bucket you store a count of the number of items, you can keep track of two sets A and B then compute a bloom filter for the difference A - B. The special thing is that the collision rate for the computed bloom filter is independent of the number of collisions in A and B individually, instead it's only dependent on the number of collisions in the set A-B. So regardless of how large A and B are, the counting bloom filters you use for those two sets can be very small so long as all you care about in the end is computing A-B, and A-B has very few collisions. This is useful in the context of set reconciliation.

Poor choice of hash function can cause some real-world consequences. See `https://lwn.net/Articles/474912/` for example!

Name: _____ **Stanford ID Number:** _____

3.6.4 Lecture 7/19 EC Question

In lecture we've been focusing on data structures that maintain a set of items under the operations insert, delete, and search, where search only indicates whether or not the item is in the set.

Suppose instead of maintaining a *set*, you want to maintain a *key–value dictionary*. We want the ability to insert *pairs* of items, (k, v), at a time. We call the first item in the pair the *key* k. We call the second item in the pair the *associated value* v. When we search for a key k, it should return the *pair* (k, v) that was inserted with that key (if any exists). For example, if we first insert the pairs $(1, A)$, $(3, C)$, $(6, F)$, then search for 3, it should return $(3, C)$. You can assume no two pairs have the same key.

Part 1: Explain (in two-to-three English sentences or short pseudocode) how to modify the **risky hash set** we saw today to implement a key-value dictionary, not just a set. *Hint: store the entire pair in the bucket, but only hash one part of the pair to the hash function.*

Part 2: Explain (in two-to-three English sentences or short pseudocode) how to modify the **binary search trees** we saw earlier in this class to implement a key-value dictionary, not just a set. *Hint: compare only part of the pair.*

3.7 Week 4 Problem Sheet

3.7.1 Lecture 7/15

Amortized Analysis

Problem 30: For each of the following applications, explain whether you would be OK using an algorithm that guarantees only (i) an average-case $O(1)$ bound, (ii) an amortized $O(1)$ bound, (iii) a worst-case $O(1)$ bound. In some cases there may be some nuance; feel free to explain your answers.

1. An algorithm that will run after every pacemaker pulse to keep track of the number of pulses.
2. An algorithm that converts batches of user-uploaded files into a different filetype, then emails the user a link to download the converted files.
3. You're writing a program to generate prime numbers that works by generating a random number and then checking if it's prime. The primality checker is the algorithm in question, i.e., the one with average/amortized/worst-case bounds.
4. An algorithm that runs on every frame of an interactive game.

Problem 31: *(Adapted from MIT 6.046J Spring 2015 Course Notes)* Suppose you start with an empty 2–3 tree and perform a sequence of n insertions. First, prove that the *worst-case* number of split operations that occurs per insertion is $\Theta(\log n)$. Then, prove that the *amortized* number of split operations that occurs per insertion is $O(1)$. Your answer should use the potential method and explicitly define a potential function.

Hint: Hint(s) are available for this problem on Canvas.

Problem 32: *(Adapted from CLRS)* In class we analyzed the dynamic array data structure with a push operation. In many applications, however, our arrays both grow and shrink. When possible, we would like to tell the operating system that we're not using as much space as we originally needed, and let it *defragment* by moving our data to a region in memory that is a better fit. So, define a corresponding *pop* operation that removes an element from the back of the array. You can also modify the *push* function from class if needed. The push and pop methods should together ensure:

1. If n is the number of elements and s is the size of the allocation region as requested from the operating system, your algorithms should ensure that $n \geq \lfloor \frac{1}{4} s \rfloor$.
2. In a sequence of push and pop operations, the amortized cost per operation should always remain $O(1)$.
3. The only things you can do with the operating system are release a whole allocation block and request a new one of a particular size, both of which you can assume happen in $O(1)$ time.

You should give two proofs: one using the direct method, and one using the potential method (precisely state the potential function you're using!).

Hint: Hint(s) are available for this problem on Canvas.

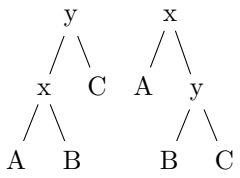
Problem 33: *(Adapted from Sleator and Tarjan. In the homework for Wednesday you will do an amortized analysis of a data structure called the* splay tree*. The standard amortized analysis of splay trees uses the potential method we saw in class. While the argument is not extremely deep, it does rely on a lot of detailed and easy-to-mess-up algebraic manipulations. In this problem you'll do pretty much*

all the algebra needed, which will make finishing up the amortized analysis after Wednesday's lecture super easy!)

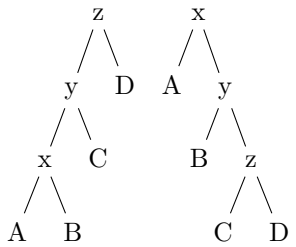
Given a binary tree T of size n, a node in the tree x, define the *size* of a node $s(x)$ to be $\frac{1}{n}$ times the number of nodes in the subtree rooted at x. Then, define the *rank* of a node $r(x)$ to be $\log s(x)$, and the potential of a tree $\Phi = \sum_{x \in T} r(x)$.

Now, consider the following three tree operations, where capital letters refer to subtrees:

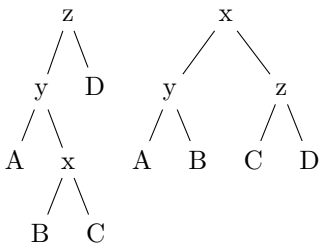
1. The single rotate:



2. The zig-zig rotate:



3. The zig-zag rotate:



Let Φ (s, r) be the potential (sizes, ranks) of the tree before one of the above operations and Φ' (s', r') the potential (sizes, ranks) afterwards. Prove the following facts, in this order:

1. If a single rotation is done:
   (a) The rank of y has decreased, i.e., $0 \geq r'(y) - r(y)$.
   (b) The rank of x has increased, i.e., $r'(x) - r(x) > 0$.
   (c) The change in potential satisfies $\Phi' - \Phi \leq 3(r'(x) - r(x))$.
   (d) The relation $1 + (\Phi' - \Phi) \leq 1 + 3(r'(x) - r(x))$.
2. If a zig-zig is done:
   (a) The new rank of x is the same as the old rank of z, i.e., $r'(x) - r(z) = 0$.
   (b) The new rank of x is greater than the new rank of y, i.e., $r'(x) \geq r'(y)$.
   (c) The change in potential satisfies $\Phi' - \Phi \leq r'(x) + r'(z) - r(x) - r(y)$.

(d) The old rank of y was greater than the old rank of x, hence $r(x) + r(y) \geq 2r(x)$.

(e) The change in potential satisfies $\Phi' - \Phi \leq r'(x) + r'(z) - 2r(x)$.

(f) For positive a, b with $a + b \leq 1$, we have $\log a + \log b \leq -2$. (Use the base-2 logarithm. My approach to this step requires some calculus, so if you haven't taken a calculus class yet feel free to just assume this.)

(g) $r(x) - r'(x) = \log(s(x)/s'(x))$ and $r'(z) - r'(x) = \log(s'(z)/s'(x))$. (You can assume the reader is familiar with standard properties of the logarithm.)

(h) $s(x) + s'(z) \leq s'(x)$

(i) $(s(x)/s'(x)) + (s'(z)/s'(x)) \leq 1$.

(j) $2 \leq (r'(x) - r(x)) + (r'(x) - r'(z))$.

(k) The potential function satisfies $2 + (\Phi' - \Phi) \leq 3(r'(x) - r(x))$.

3. If a zig-zag is done:

(a) The new rank of x is the same as the old rank of z, i.e., $r'(x) - r(z) = 0$.

(b) The change in potential satisfies $\Phi' - \Phi \leq r'(y) + r'(z) - r(x) - r(y)$.

(c) The old rank of y was greater than the old rank of x, hence $r(x) + r(y) \geq 2r(x)$.

(d) The change in potential satisfies $\Phi' - \Phi \leq r'(y) + r'(z) - 2r(x)$.

(e) $r'(y) - r'(x) = \log(s'(y)/s'(x))$ and $r'(z) - r'(x) = \log(s'(z)/s'(x))$.

(f) $(s'(y)/s'(x)) + (s'(z)/s'(x)) \leq 1$.

(g) $(r'(y) - r'(x)) + (r'(z) - r'(x)) \leq -2$.

(h) The potential function satisfies $2 + (\Phi' - \Phi) \leq 2(r'(x) - r(x)) \leq 3(r'(x) - r(x))$.

3.7.2 Lecture 7/17

Splay Trees and Union-Find

Problem 34: Show the result of the following sequence of splay tree operations: insert 7, insert 8, insert 6, insert 4, search for 5, search for 7, insert 1, insert 2, insert 0, insert 3, split at 6, join the two trees.

Problem 35: Show the union-find forest that results after each operation in the following sequence of union-find operations on integer sets: union(1, 2), union(3, 4), union(1, 4), union(4, 5), same-set?(3, 8), union(6, 7), union(8, 6), same-set?(6, 1).

Problem 36: *(Adapted from Sleator and Tarjan)* Recall the terminology and results of Problem 4. We are now going to consider the amortized time of a sequence of n splay-search operations in a tree with n nodes. Our cost model will assume that a single rotate costs 1 unit of time to perform, a splay-zig-zig costs 2, and a splay-zig-zag costs 2; everything else is free (this is reasonable because you need to rotate proportional to how far down the tree you are, which hence also captures the search time).

1. Let t be the total cost of splaying a node x and $\Delta\Phi$ be the change in potential before and after splaying node x all the way up to the root. Prove that $t + \Delta\Phi \leq 1 + 3(r'(x) - r(x))$, where $r'(x)$ is the rank of x after the entire splaying operation is done and $r(x)$ is the rank of x before the splaying operation starts.

2. Let x be any node in any BST of size n. Prove that $-\log n \leq r(x) \leq 0$.

3. Prove that $t + \Delta\Phi \leq 1 + 3\log n$.

4. Consider any sequence of n splay-search operations starting from a BST with n nodes in it. Conclude that the amortized time per splay-search operation in that sequence is $O(\log n)$.

For the final part, you might need the following fact: you can weaken the first condition of the potential method theorem from lecture to just $|\Phi(s_i) - \Phi(s_0)| = O(nf(n))$.

Hint: Hint(s) are available for this problem on Canvas.

Problem 37: *In class we told you that the union-find algorithm shown has $O(\alpha(n))$ amortized per-operation time for a sequence of n operations. Unfortunately, proving that is extremely tedious. What we're going to do in this problem is analyze a simplified version of that algorithm, and show that even removing many of its optimizations, it still performs quite well.*

The union-find algorithm we saw in class had two optimizations: (1) we always made the tree with more items in it the root during a join, and (2) we did path compression where we update the parent pointers every time we did a find-root. Consider modifying that union-find algorithm to remove optimization (2), i.e., no longer do path compression. Furthermore, instead of always making the tree with more items in it the root, make the *tallest* of the two trees the root. Prove that this modified union-find algorithm has worst-case $O(\log n)$ time per operation in a sequence of n operations.

Then, prove that the *original* union-find algorithm has worst-case $O(\log n)$ time per operation in a sequence of n operations. *I don't think you can prove this as a corollary of the above; rather, you will hopefully find that your proof from above should either work directly on the original union find algorithm, or can be made to do so with few modifications. If not, make sure to take a peek at the hint.*

Hint: Hint(s) are available for this problem on Canvas.

Problem 38: Suppose instead of the special zig-zig and zig-zag splay tree operations, we did something a little simpler: until the node in question is the root, either rotate left or rotate right around its immediate parent to move it further up the tree by one. This is called a *move-to-root* tree, and was proposed before splay trees were discovered (see Allen and Munro, 1978).

1. Draw a tree and pick a node x where splaying x and move-to-root'ing x result in different trees.
2. Prove that move-to-root trees do *not* have the same amortized access property that we proved for splay trees in Problem 7 by constructing a tree with n nodes and sequence of n lookups that takes $\Theta(n^2)$ total time.
3. Try the same sequence of lookups on a splay tree to get an intuitive understanding of why the same worst-case behavior doesn't happen.
4. Explain why the amortized time bound we saw above for splay trees doesn't apply to move-to-root trees. *(You don't need too much for this part — just point to a step in the proof from problem 4 or 7 that doesn't apply to move-to-root trees.)*

Hint: Hint(s) are available for this problem on Canvas.

3.7.3 Lecture 7/19

Hashing and Risky Hash Sets

Problem 39: *Here's another practically unlikely, morally dubious, yet pedagogically interesting scenario:* Imagine some careless developer decides to use the "risky hash set" we saw in lecture to keep track of the set of people (represented by numerical user IDs) who have paid for their streaming service. Even worse, they're using the following naïve hash function: take the remainder after dividing the user

ID by 2^{16}. Now, imagine you have a noisy neighbor who likes to watch that streaming service way too late into the night at way too high a volume. You know the streaming service is using this risky hash set with this naïve hash function, you know your neighbor's user ID is 123, and the streaming service allows you to choose your own ID when you sign up (as long as it isn't already taken). What could you do to bump your neighbor off of their ability to stream (at least until they contact customer service)?

Problem 40: *(Carter and Wegman)*

1. Prove that, for any prime p and numbers $0 \leq x, y, r, s < p$, with $x \not\equiv y$ and $m \not\equiv 0$ the equations

$$xm + n \equiv r, \quad ym + n \equiv s$$

   have a unique solution for the variables n, m mod p.

2. Fix a number $b \leq p$ and another number $0 \leq s < p$. Explain why there are at most $p\lceil \frac{p}{b} \rceil$ choices of $0 \leq r, s < p$ satisfying $r \equiv s \bmod b$.

3. Define the hash function $h_{m,n}(x) \coloneqq ((xm + n) \bmod p) \bmod b$. This hash function maps values from the range $0, 1, \ldots, p-1$ into the buckets $0, 1, \ldots, b-1$. If $x \neq y$, then for how many choices of $0 < m < p$, $0 \leq n < p$ is $h_{m,n}(x) = h_{m,n}(y)$? *(An overapproximation is fine; no need for an exact count.)*

4. Explain why, if the streaming service used this hash function with m and n chosen randomly and kept secret from everyone, it would make it a lot harder to carry out your plan in problem 10. (More generally, you should explain why using the hash function $h_{m,n}$ in a risky hash set makes the analysis we did in lecture about hash collisions 'correct' even though the function $h_{m,n}$ isn't really chosen from the set of *all* possible hash functions.)

For part 1, you can use the fact that $\mathbb{Z}_p$ is a field. In this context, the important consequence of that fact is that, if $a \not\equiv 0$, then there exists a unique value $a^{-1} \in \{1, 2, \ldots, p-1\}$ such that $aa^{-1} \equiv 1 \bmod p$.

Hint: Hint(s) are available for this problem on Canvas.

Problem 41: *(I originally saw this problem somewhere in Knuth, but can't find it now. Will update if I find a reference.)* Given a set V of possible values, we use the term *shuffler function* to refer to any bijection on the values in V, i.e., $s : V \to V$[a]. Suppose instead of a hash function your program begins by picking a random shuffler function s. Show how you can use this shuffler function to construct a BST-like data structure with good *expected* depth. In other words, for *any* sequence of insertions, if s is chosen uniformly at random from the set of all shuffler functions, the expected amortized time to perform any sequence of n insertions should be $O(\log n)$ per operation.

(Alternatively, you can prove the expected depth of the tree should be $O(\log n)$. But that uses a result we haven't seen explicitly in lecture.)

Your data structure should be very similar to a BST, meaning, values should be associated with nodes in some tree. Your nodes should only need fields for the value, left-child, and right-child pointers, no additional space, e.g., for storing a red/black color. The one caveat is that your data structure only needs to support insertion and searching quickly — it's OK if operations like find-min, find-max, and deletion can make things arbitrarily slow. Also, you're welcome to reference theorems from lecture.

[a] If you find this definition interesting, you may want to read about blockciphers!

3.8 Lecture 7/22 Notes: Hash Tables With Collision Resolution

Last week we saw *risky hash sets*, which allowed $O(1)$ operation times but required $\Theta(n^2)$ space and had some chance of dropping items (but the expected number of such items was $O(1)$).

Today we'll see how to fix the latter two problems. In particular, we'll use only $\Theta(n)$ buckets rather than $\Theta(n^2)$. Unfortunately, that means we will expect

$$\binom{n}{2}\frac{1}{n} = \frac{n-1}{2}$$

collisions! Crucially, however, those collisions are spread out among $\Theta(n)$ buckets: in fact, then, we only expect about

$$\frac{n-1}{2}\frac{1}{n} = \frac{n-1}{2n} \approx \frac{1}{2}$$

collisions *per bucket*. So we need to try and take advantage of this property.

3.8.1 Revisiting the Problem

Recall that the main problem was as follows: suppose, for example, you have a RHS with 4 buckets and the hash function x mod 4. Then, if I insert 4, 5, 8, the RHS will have to bump 4 out of its place to insert 8!

3.8.2 Big Buckets and Separate Chaining

The most natural way to solve this problem is to just make our buckets bigger, so they can store multiple items in the case of collisions. But we don't really know ahead-of-time a hard limit on how big these buckets will have to be (e.g., if we get very unlucky, we could have all the items land in the same bucket!). So if we want to be really resilient, we need the buckets themselves to be represented as something like a linked list, so we can add arbitrarily many items in that bucket as needed.

This leads to the idea of *separate chaining*.

```
def insert(buckets, v):
    if search(buckets, v): return
    (maybe rehash)
    buckets[hash(v)].prepend(v)
```

```
def search(buckets, v):
    for node in buckets[hash(v)]:
        if node.value == v:
            return True
    return False
```

```
def delete(buckets, v):
    for node in buckets[hash(v)]:
        if node.value == v:
            delete node from list
    (maybe rehash)
```

In our earlier example, instead of bumping 4 out, we'd just make a linked list with 8 and 4 both.

Knobs and Measures for Separate Chaining Hash Tables

1. What's the *load factor*? (Number of items divided by number of buckets.)
2. How frequently do you rehash? For separate chaining, we'll assume you rehash to keep the load factor between 0.25 and 1.0, i.e., $\Theta(n)$ buckets for n items.
3. What's the expected chain size? Max chain size? These correspond to search times.
4. What hash function(s) are used?

Black-Box Expected Lookup Time

Suppose some sequence of insertions is determined then a separate chaining hash table is used to store those insertions with load factor 1 (n buckets, n items) and a randomly chosen hash function. Suppose you are allowed to pick one value x and search for it, but you have to pick this value *without* knowing what the hash function is or what the hash table itself looks like. In this black-box setting, what is the expected time to lookup x?

First, note that it doesn't matter what x is except for whether or not it was inserted.

First, assume x was not inserted. Then it hashes to a random bucket, so our question is, what is the expected length of the chain in that bucket? The chance any given value landed in the first bucket is $\frac{1}{n}$ independent of the others, so the expected number that land in that bucket is

$$\frac{n}{n} = 1,$$

hence the expected lookup time is $O(1)$.

On the other hand, if x *was* inserted, then we know for sure at least it exists in the bucket, but we can repeat the analysis on the remaining $n-1$ items to see that the expected number that land in that bucket is

$$1 + \frac{n-1}{n} \leq 2 = O(1).$$

So in either case, the expected lookup time is $O(1)$.

White-Box Worst-Case Lookup Time

Suppose instead you are allowed to pick x *knowing the hash function*, i.e., after looking at all the buckets and seeing which is most full. What is the worst-case expected lookup time in this white-box setting? Well, to maximize the time taken you should pick a value that lands in the bucket with the longest chain. So let's ask: is the expected value of the longest chain also $O(1)$? Surprisingly, the answer is no!

You'll investigate this a bit further on the homework, but let's work through a little bit here. The probability of any given bucket having a chain of length at least k is

$$\begin{aligned}
\sum_{i=k}^{n} \binom{n}{i} \frac{1}{n}^{i} \left(\frac{n-1}{n}\right)^{n-i} &\geq \binom{n}{k} \frac{1}{n}^{k} \left(\frac{n-1}{n}\right)^{n-k} \\
&\geq \binom{n}{k} \frac{1}{n}^{k} \left(\frac{n-1}{n}\right)^{n} \\
&\to \frac{n!}{k!(n-k)!} \frac{1}{n^k} e^{-1} \\
&\approx \frac{n^k}{k!} \frac{1}{n^k} e^{-1} \qquad \text{(Not quite correct! See HW.)} \\
&= \frac{1}{k!e}.
\end{aligned}$$

On the homework you'll see that the approximation we made is not quite valid, because k is not a constant. But you'll also see that the rest of this argument works out. Anyways, if you take $k = \beta(n)$ where β is an

continuous, monotonic inverse of $n!$, then this slightly wrong analysis tells us we should expect about

$$n\frac{1}{ne} = \frac{1}{e}$$

of the bins to have at least $\beta(n)$ balls. But $\beta(n) \neq O(1)$, so in fact, the worst-case expected lookup time in this setting is not $O(1)$.

Conclusion

Separate chaining gives us pretty much everything we want:

1. $O(n)$ space overhead, and
2. Expected $O(1)$ lookup times.

3.8.3 Linear Probing

One major issue with separate chaining is that linked lists are *terrible* for cache performance. Linear probing is an approach that has much better cache locality. The idea is simple: if you try to insert into a bucket that's already full, try the next bucket!

```
def lp_insert(buckets, v):
    if search(buckets, v): return
    (maybe rehash)
    b = hash(v)
    while buckets[b] not empty: b = next bucket
    buckets[b] = v
```

```
def lp_search(buckets, v):
    b = hash(v)
    while buckets[b] != v and buckets[b] not empty and not looped around:
        b = next bucket
    return buckets[b] == v
```

```
def lp_delete(buckets, v):
    if not search(buckets, v): return
    b = hash(v)
    while buckets[b] != v:
        b = next bucket
    buckets[b] = (tombstone)
    (maybe rehash)
```

On the homework you'll consider how frequently rehashing needs to be done.

White-Box and Black-Box Worst-Case for Linear Probing With Load Factor 1

Let's revisit our original setting; n buckets for n items, no tombstones. What's the worst case now if we try to look something up? Surprisingly, if the thing we're looking up is *not* in the table, the worst case is now extremely bad! That's because we'll have to iterate through the whole table to determine that the value is not in it!

So the worst-case behavior for the load-factor-one setting with linear probing is actually $\Theta(n)$.

Primary Clustering

Unfortunately, this is a problem even at lower load factors. Define a *run* to be a contiguous sequence of filled-in buckets (possibly wrapping around the table). The worst-case lookup time corresponds to the length of the longest run. We won't analyze that quantity in this class, but one thing to note is the following: on each insertion, a run of length k is *twice as likely to be extended* than a run of length $k/2$. This "rich get richer" phenomenon means that we some runs may get absurdly long, causing quite poor worst-case lookup times even with smaller load factors (according to Bender et al., according to Knuth there's a bigger issue at play).

Less Extreme

One thing that's important to know is that I've probably made this scenario sound worse than it is; my understanding is that, as long as you keep the load factor below $\frac{1}{2}$ (or any constant < 1) expected lookup times remain $O(1)$. So really it just means you need to overprovision space by a small constant factor; not a bad deal to have such great cache locality. In fact, compared to separate chaining, this overprovisioning is 'free' because separate chaining also needed an extra $\Theta(n)$ 'next' pointers to implement the linked lists! In fact, recent research has shown linear probing is even better (in theory) than we used to think. Unfortunately, it's just not super easy to analyze.

3.8.4 Post-Lecture Ed Notes

One of your classmates mentioned this interesting hash function in a private Ed post: `https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternative-to-int`

Personally, I've found djb2 works well for strings: `https://doc.riot-os.org/group__sys__hashes__djb2.html`

Knuth has an interesting discussion of the following, extremely surprising theorem and its application to hash function design: `https://en.wikipedia.org/wiki/Three-gap_theorem`

The following paper makes an interesting argument that primary clustering isn't actually as bad as people (including Knuth!) claim it is: `https://arxiv.org/abs/2107.01250`

Name: _____ **Stanford ID Number:** _____

3.8.5 Lecture 7/22 EC Question

Consider the hash table $[5, 2, \perp, \perp, 3]$ using linear probing with the hash function $h(x) = 3x \bmod 5$. Give a number $x \notin \{5, 2, 3\}$ such that inserting x takes the worst-case possible time.

3.9 Lecture 7/24 Sketch: Quiz 2 Review

1. Meaning of bounds when some interpretation is necessary (problem vs. algorithm)
2. Red-black trees: they're not just trees colored red/black!
3. Potential method: work through the proof one more time, this time give the slightly more general theorem that was needed for splay trees (but they don't need it for teh quiz).
4. Potential method: coming up with potential functions (out-of-scope, but lots of people were interested). Key idea for some of the more trivial examples: make the potential larger the closer you get to an 'expensive' operation.
5. Potential method: examples HW4 problem 2, HW 4 problem 3
6. Union find: friendship islands example has a better solution that we'll see in the graphs section!

3.10 Week 5 Problem Sheet

Since we have the quiz on Friday, you are only required to turn in two problems for this week.

3.10.1 Lecture 7/22: Hash Sets and Hash Maps

Problem 42: Assume a hash table for storing integers uses the naïve hash function $h(x) = x \bmod b$ where b is the number of buckets. For each collision resolution method (separate chaining and linear probing), describe a sequence of n insertions having worst-case per-operation asymptotic time and compute that worst-case per-operation time.

For simplicity, you can assume the hash table starts with $b = 2n$ empty buckets and it never rehashes. Remember that our hash table pseudocode includes a search before every insertion.

Problem 43: Assume you have a hash table using either separate chaining or linear probing. Explain how to implement the following ability: at any point, the user may change into "iteration mode" where they can repeatedly call a NextItem method to iterate through all the items in the hash table. You can assume they don't perform insertions or deletions while in iteration mode. If there are n items in the hash table and $\Theta(n)$ buckets, the amortized time to call NextItem n times (i.e., iterate through all the items) should be $O(1)$ per call. Your modification should require only $O(1)$ additional space. Also explain why your modification would *not* work on the risky hash set we saw in class with quadratic overprovisioning, i.e., n^2 buckets, even if there were no collisions.

Problem 44: In practice, time is often dominated by *cache misses*. In a separate chaining hash table, the number of cache misses is roughly one (for reading the pointer stored in the bucket) plus the number of unique linked list nodes you have to read from after that. For each of the following two ideas, assume the hash table contains n items then compute: (i) the expected number of cache misses after searching for a missing value and (ii) the total expected space usage (assume each bucket is a one-word pointer to a linked list node, and each linked list node contains one word per value stored and one word for the pointer to the next node).

1. Make each linked list node store two values, and assume there are n buckets (load factor one).
2. The linked list nodes only store one value, but you have $2n$ buckets (load factor one half).

Note: getting exact counts for the first approach will be difficult; instead, try to get good enough bounds that you can state with confidence which one is better. More generally, this problem is a bit of an "algebra playground;" see the hint if you got lost in the sandbox! You can use the approximation $\left(\frac{n-1}{n}\right)^{n-O(1)} \approx e^{-1}$ *without proof if needed.*

Hint: Hint(s) are available for this problem on Canvas.

Problem 45: Explain how to determine when to rehash in the linear probing scheme. You should guarantee that the load factor stays between $\frac{1}{4}$ and 1. The expected time should be $O(1)$ amortized over a sequence of n operations; for simplicity, you can assume that rehashing is the only operation that takes any time.

If you have extra time, try to modify your approach so the load factor stays between $\frac{1}{4}$ *and* $\frac{1}{2}$*. Why is this extremely desirable in linear probing, rather than letting the load factor get all the way up to 1?*

Hint: Hint(s) are available for this problem on Canvas.

Problem 46: *(Adapted from the CMU Spring 2011 Randomized Algorithms class blog.)* In lecture we saw, for separate chaining, that the probability of any given bucket having a chain of length at least k was greater than $\binom{n}{k}\frac{1}{n^k}\left(\frac{n-1}{n}\right)^n$. We then performed an analysis that relied on the dubious assumption $\binom{n}{k} \approx n^k/k!$ even when k is a function of n. Instead, prove with more care the following:

1. The limit $\left(\frac{n-1}{n}\right)^n \to e^{-1}$ as $n \to \infty$. *(My solution to this part requires some differential calculus; if you haven't taken calculus before, feel free to assume $\left(\frac{n-1}{n}\right)^n \approx e^{-1}$ for the rest of this problem.)*
2. That, when $a \geq b > 1$, $\frac{a}{b} \leq \frac{a-1}{b-1}$.
3. That $\binom{n}{k} \geq \left(\frac{n}{k}\right)^k$ and so $\binom{n}{k}\frac{1}{n^k e} \geq \frac{1}{k^k e}$.
4. That, when $\beta(n)$ is chosen such that $\beta(n)^{\beta(n)} = n$, we expect at least a constant number of buckets in a separate chaining hash table with n buckets and n items to have chains of length $\beta(n)$.

Hint: Hint(s) are available for this problem on Canvas.

Problem 47: Suppose you have access to an API running over the internet that implements a hash table (i.e., has search/insert/delete endpoints) using separate chaining, and assume you have a stable enough connection that you can reliably time how long it takes the server to finish each of those operations. Suppose the hash table starts off empty and you are able to perform $O(n)$ operations. Explain how you can, with at least constant probability, within $O(n)$ operations, find some operation that consistently takes $\Omega(\beta(n))$ time for $\beta(n) \neq O(1)$.

(It's fine if you only prove that operation takes $\Omega(\beta(n))$ time on expectation, rather than with at least constant probability. If you want to try to prove it has at least constant probability, see the hint.)

Hint: Hint(s) are available for this problem on Canvas.

3.10.2 Lecture 7/24: Review

Problem 48: Explain how to modify splay trees to keep track of the sum of all the values in the splay tree. Your modification should also work for the split and join operations, e.g., after split it should be possible to compute the sum of the values in each of the two resulting trees.

Problem 49: Explain how to implement the same NextItem operation as described in Problem 2, except for BSTs. You can assume the BST has parent pointers, i.e., given a node in the BST you can find its parent in $O(1)$ time (in addition to its value, its left child, and its right child, all also in $O(1)$ time). Your NextItem method should iterate through the items in sorted order, and you should prove that it takes $O(1)$ amortized time per call to iterate through the entire tree, regardless of how balanced the original tree is. Finally, determine whether the worst-case time per call is also $O(1)$.

Hint: Hint(s) are available for this problem on Canvas.

Problem 50: Explain in different words (e.g., using phrases like "For all algorithms ..." or "There exists an input ...") what the following statements most likely mean:

1. Problem X requires $\Omega(n)$ space.
2. Problem X requires $O(n^2)$ space.
3. Algorithm Y has worst-case time $O(n^3)$.

4. Algorithm Y has worst-case time $\Omega(n)$.
5. Data structure Z has worst-case per-operation time $\Theta(n^2)$.
6. Data structure Z has amortized per-operation time $O(n^2)$.

3.11 Study Guide for Second Quiz (Searching)

While we don't expect any changes, everything in this document is tentative and subject to change by announcement from the instructor.

Topics

The following topics are in-scope:

1. BSTs and BST operations
2. AVL tree and 2-3 tree definitions and insertion operations; deletion not needed for either, but you should know the running time of deletion.
3. Converting between 2-3 trees and red-black trees.
4. Meaning of amortized analysis, the direct method, and the potential method.
5. Amortized analysis of dynamic array.
6. Splay tree operations (splay, search, split, join, insert, delete) and amortized time bound (not proof, just meaning).
7. Union-find algorithm and union-find amortized bound (not proof, just meaning).
8. Direct addressing, expected number of collisions in a risky hash set, rehashing idea.
9. Usefulness of random hash functions vs. pre-chosen hash functions.
10. Separate chaining, linear probing, and the analyses we do in lecture for them.

Having done the homework questions might be useful in answering quiz questions, but we will write the quiz assuming you haven't done the homework.

Question Templates

While there may be a few exceptions, we'll try to stick to the following question templates for the quiz:

1. Given an algorithm using one of the data structures we talked about, how long does it take to run?
2. Question(s) proposing a slight modification to one of the algorithms we've seen in lecture, then asking things like: Is it still correct? What is the effect on running time?
3. Question(s) where some type of input is described and we ask things about the behavior (e.g., running time) of a particular algorithm on that type of input.
4. Question(s) proposing a scenario and then asking you to compare + contrast different search data structures for that scenario, or design an algorithm using the search data structures we've seen for that scenario.
5. Question(s) proposing an algorithm and then asking you to analyze the amortized time using either the direct or potential method. In the potential method case, we will provide you the potential function to use.
6. Question(s) asking you to simulate some of the algorithms from lecture (or modified versions of those algorithms) on an example input.

Questions will be easier than the average homework question, and probably a little bit harder than the average post-lecture EC credit. The motivation for having so many questions is to ensure that no one question has an outsized impact on your grade.

Chapter 4

Graphs and Algorithm Design

4.1 Lecture 7/29: Welcome to Graphs!

The following scenarios all have something in common:

1. A map showing how towns are connected by roads.
2. The Web, where webpages can connect to other webpages by hyperlinks.
3. The Facebook social network where people can be friends with other people.
4. A university course offering, where courses can have other courses as prerequisites.

What is it? They are all made up of *objects* (towns, webpages, people, courses) and pairs of objects can be *related* in some way (roads, links, friendship, prerequisites).

For the next two weeks, we're going to study algorithms that can be applied to *all* of these problem domains. To do so, we need some general way of representing what is common between all of them. This is the notion of a *graph*.

Definition 17. *A* graph *is a set V of nodes (also called vertices) and a function $E : V \times V \to \mathbb{R} \cup \{\perp\}$ of* edge weights*.*

The 'nodes' are the objects (towns, webpages, people, courses), and the edge weights are the relations (roads, links, friendship, prerequisites). Generally, if there is no relation between nodes a and b then we set $E(a, b) = \perp$. Otherwise, we set $E(a, b)$ to be the "amount of relation" between them; in the map setting, for example, $E(a, b)$ might be the length of the road taking you from a to b. We usually say there "is an edge" from a to b only when $E(a, b) \neq \perp$.

Note: there are many equivalent definitions of a graph. We'll stick to this one for lecture, but it doesn't matter too much what you use.

4.1.1 Interacting With Graphs

A graph is a mathematical object; it only really exists in our imagination. But computers are real, physical things, so we need to ask: how should we represent a graph in our computer programs? There are a number of options, including:

1. The *adjacency matrix* approach: store a big $n \times n$-sized matrix where entry i, j has the value of $E(i, j)$.
2. The *adjacency list* approach: store for each node a the list of all other nodes b that it has an edge to, i.e., $\text{Out}(a) \coloneqq \{b \mid E(a, b) \neq \perp\}$. We often also assume that similar lists are available for all incoming edges, i.e., $\text{In}(a) \coloneqq \{b \mid E(b, a) \neq \perp\}$.

In this class we'll mostly assume the adjacency list approach if nothing else is stated. But you should be aware that different representations may be better in different contexts. In other words, the main assumption we'll make is that you can iterate through the outgoing and incoming edges of a node in $O(1)$ time per edge. This is generally a pretty reasonable assumption; sometimes you have to play pointer tricks to get it to work but we won't worry about those implementation details in this class. For cases where the values of edges are important, you can imagine storing the out list in an extended form that also includes the weight of that edge, e.g., $\text{Out}(a) \coloneqq \{(b, E(a,b)) \mid E(a,b) \neq \perp\}$.

4.1.2 Graph Parameters

Given a graph, there are a number of questions we can ask about it.

In many settings there won't be any need to distinguish between "amounts of relation," e.g., course b either depends on a or it doesn't! In those cases we'll just have $E(a,b) = \perp$ or 1, and we'll call those graphs *unweighted*.

Definition 18. *A graph is* unweighted *if for every $a, b \in V$, $E(a,b) \in \{\perp, 1\}$. (In this scenario, we usually draw the edges without numbers on top.)*

In many scenarios, the relation is *symmetric*: if I'm Facebook friends with you, then you must be Facebook friends with me. We'll call such graphs *undirected*.

Definition 19. *A graph is* undirected *if for every $a, b \in V$, $E(a,b) = E(b,a)$. (In this scenario, we usually draw the edges without arrows.)*

Some graphs are *dense*: they have a lot of edges. Other graphs are *sparse*: they have relatively few edges.

Definition 20. *Let n be the number of vertices in a graph and m be the number of edges a graph is* sparse *if $m = O(n)$ and it is dense if $m = \Theta(n^2)$.*

(Note: this definition only really makes sense if you apply it to a *sequence* of graphs, not a single graph, because otherwise the asymptotic notation is meaningless. But we still often talk about the sparsity of a single graph in informal terms.)

4.1.3 BFS and DFS: What Can I Reach?

One of the most fundamental questions you can ask about a graph is the following: if I start at node x and walk along edges what other nodes can I reach? In the map setting, for example, this corresponds to asking what other cities I can reach from town x.

We can solve this problem like so:

```
def explore(graph, start):
    reachable = set()
    frontier = set({start})
    while frontier:
        node = frontier.pop()
        reachable.insert(node)
        frontier.insert_all(graph.Out(node) - reachable)
    return reachable
```

The idea is to start at x and explore paths through the graph one-by-one. We do so by keeping track of two sets:

1. A set of nodes that we can definitely reach from x (the "reachable set"),
2. And a subset of those nodes (the "frontier") that we have not yet explored to see what is reachable from them.

Theorem 25. *The reachable-from algorithm correctly returns all the nodes that can be reached from* `start`.

Proof. (Sketch) Let R be the reachable set and F be the frontier set. The algorithm maintains this invariant:

1. `start` $\in (F \cup R)$, and
2. For any node $x \in R$, we have $\text{Out}(x) \subseteq (F \cup R)$.
3. For any node $x \in R$ there exists a path from `start` to x.

Thus, when the loop exits because $F = \emptyset$, we know:

1. `start` $\in R$;
2. For any $x \in R$, we have $\text{Out}(x) \subseteq R$;
3. For any $x \in R$, there is a path from `start` to x

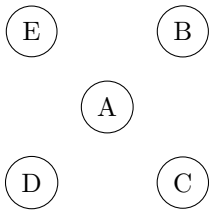
Hence, every node in R is indeed reachable from `start` and we just need to prove the other direction, i.e., that any reachable node is in R. Let x be such a reachable node and consider any path $p_1 = \texttt{start}, p_2, \ldots, p_n = x$. By property (1) we know $p_1 \in R$, then by property (2) we know $p_2 \in R$. Applying property (2) repeatedly we see $p_3 \in R$, $p_4 \in R$, $\ldots$, $p_n = x \in R$, as desired. (Induction can be used to prove the $\ldots$ part slightly more formally/rigorously.) □

Running Time

Each node is visited once by the loop, and each loop iteration does work proportional to the number of outgoing edges from that node. Hence, the total time is bounded by $O(n + m)$ where m is the number of edges in the graph.

In fact, if you're careful, you can argue that it's $O(1 + m)$ time. Why is this? Other than $O(1)$ time setting up the reachable/frontier sets at the beginning, the time is dominated by inserting edges on line 7. So consider any edge $a \to b$. That edge $a \to b$ is processed *at most one* time, on line 7, when the outer loop processes a. So the total time is $O(1) + O(m) = O(1 + m)$.

Note this only makes a difference asymptotically when $m < n$, i.e., when the graph is *very* sparse and so you have more nodes than edges. In that case, your 'explore' routine won't actually visit every node, which is how you can avoid counting the $O(n)$. For example, consider the graph and start node A:



The running time of explore is ≈ 1, because it *only* visits node A!

BFS and DFS

We've left one big thing ambiguous in the above pseudocode: when you pop from the frontier, which node from the set do you pop? There are two common choices:

1. If you pop the *most-recently inserted item*, then the set essentially becomes a *LIFO stack*, and the resulting algorithm is called *depth-first search* (DFS).
2. If you pop the *least-recently inserted item*, then the set essentially becomes a *FIFO queue*, and the resulting algorithm is called *breadth-first search* (BFS).

I think in 106B you saw implementations of both queues and stacks; if not, you can imagine using a doubly linked list and inserting to either the front or the back depending on what behavior you want.

Both variants are correct. BFS prioritizes visiting all of the nodes close to the start before it checks out further-away nodes, while DFS kind of just "goes for it." Which you use will depend on the application, and a lot of times it doesn't really matter.

Note: BFS/DFS rely on the following behavior of stacks/queues:

1. If you reinsert an item that is *already* in the stack, then that item moves to the top of the stack (i.e., it will be popped next).
2. If you reinsert an item that is *already* in the queue, then that item keeps its place in the queue.

You can remember these rules by assuming items are 'eager' to be popped, hence they don't want to give up their spot in a queue (line), but would like to jump up to the top of the stack if offered.

Algorithm vs. Template

This is a good place to talk about the fact that BFS/DFS are not so much 'algorithms' as they are 'algorithm templates.' For example, frequently you're not worried about getting the set of *every* node that reachable, rather you just want to know if some specific node is reachable ("can I get to Athens?"). So you might modify the loop we saw above to stop if it ever encounters the Athens node. These are the sorts of modifications you'll need to get practiced with making, because these algorithms rarely show up in exactly the form I've described above, rather, they are sort of "morally the same thing" with some small changes.

4.1.4 Topological Sort

We'll end by talking about one more graph problem: topological sort. Consider the course prerequisites example. One question you might ask is: what's an order I could take all classes in so that I meet all prerequisites before taking a class?

The following algorithm solves this problem:

```
def toposort(graph):
    (make a temporary copy of graph)

    ordering = []

    all_prereqs_met = set()
    for node in graph:
        if graph.In(node) is empty:
            all_prereqs_met.insert(node)

    while all_prereqs_met is not empty:
        node = all_prereqs_met.pop()
        ordering.append(node)
        for other in graph.Out(node):
            remove edge node->other
            if graph.In(other) is now empty:
                all_prereqs_met.append(other)

    return ordering
```

The fundamental idea is to keep track of the list of classes where you meet all the prerequisites. At each step, you pick one of those nodes to be the next class you take. Delete all the outgoing edges for that node, since now all of those prerequisites are met for you; if any nodes now have all their prerequisites met, add them to the set. Then repeat!

In general, such an order is called a *topological sort*:

Definition 21. *A* topological sort *(often shortened to* toposort*) of a graph is an ordering of its nodes such that if there is an edge $a \to b$, then node a must appear before node b in the ordering.*

Theorem 26. *Given an acyclic graph, the toposort algorithm returns a toposort of the nodes.*

Proof. (Sketch) You can use the following invariant for the main loop: the nodes (classes) in `all_prereqs_met` are exactly those classes for which all their prerequisites would have been met after taking the classes already in `ordering`. □

Running Time of Toposort

What about the running time? Let n be the number of nodes and m the number of edges. Each edge $a \to b$ is visited exactly once on line 15, namely, when its source node a is deleted. So line 15 runs $O(m)$ times total. Lines, e.g., 12 and 13 run exactly once per node, hence they run $O(n)$ time total. A similar analysis accounts for the other lines, hence the overall running time is $O(n + m)$.

Using Toposorts

Toposorts are quite nice: they essentially turn the graph into a list! Hence, **A Good Rule of Thumb:** If you know you're working with an acyclic graph, first get a toposort and see if that helps you solve your problem.

4.1.5 Algorithm Design

These next two weeks are going to focus pretty heavily on algorithm design. For example, your suggested homeworks will pretty much just involve doing leetcode on your own. In general, there is no 'algorithm' or 'procedure' for solving these sorts of problems: you just have to try lots of things and try to rule out approaches that won't work as quicksy as possible. It does help to practice lots and lots and lots of these sorts of problems so that you can start to see patterns that might hint at what approach is most useful. Also, I suggest always starting a problem by asking yourself the following questions: Is there a graph here? If so, what precisely are the nodes and edges? Then, if you have a well-defined graph, you can start applying techniques like BFS/DFS/toposort that we've started to see today and ask whether they get you any closer to your goal.

4.1.6 Post-Lecture Ed Notes

Welcome to graphs! Here's some code implementing BFS, DFS, and toposorting, and some skeleton code if you want to practice implementing BFS/toposort . . . Also, note:

- For graphs there are many different ways you could represent the graph; in the above code, I've tried to stick pretty close to the adjacency list representation we described in lecture. But in general you might imagine lots of other implementation ideas, e.g., using a hashmap to store a mapping from (a, b) -¿ E(a, b). Each choice of implementation/graph representation will lead to different algorithms and time complexity. For each algorithm, you could ask yourself: if I implement the graph using representation X, how efficient would the algorithm be?
- In the toposort code, I've slightly modified the algorithm so that it doesn't need to make an explicit copy of the whole graph. You may find this modification interesting.
- I used (essentially) a direct addressing table to store sets; in many applications, you can store a set by adding an extra bit to each node in the graph that indicates whether or not it's in the set.
- As usual, the rest of this note is out-of-scope.

(An anonymized student) asked an interesting question: if you're only interested in taking, say, CS 161, can you find an sequence of classes that gets to 161 as fast as possible? The algorithm for this that seems most natural to me is the following: (1) First, do a "backwards BFS/DFS" from CS 161 to figure out all of its prerequisites (including transitive prerequisites, like 106A). (2) Then, do a toposort but do it only within the subgraph consisting of CS 161 and its transitive prerequisites.

The result will be a shortest-length possible sequence of classes you need to take that respects all prerequisites and includes CS 161. It is shortest possible because it includes only (transitive) prerequisites of CS 161, and you must indeed take all of those prerequisites before taking 161.

(Equivalently, you can find a toposort on the whole graph, then remove any classes that are not (transitive) prereqs for 161.)

Also note that the course-planning problem gets a lot more complicated when there are more general prereq constraints (e.g., "you must have taken either X or B, in addition to C"). You could probably find a minimal extension of this problem that is NP-complete; reducing from 3SAT might be the way to go.

Name: _____ **Stanford ID Number:** _____

4.1.7 Lecture 7/29 EC Question

What would happen if you try running the toposort pseudocode on a graph that has a cycle?

4.2 Lecture 7/31: Shortest Paths and Dynamic Programming

4.2.1 Clarify Ambiguity from DFS

Note on Monday's lecture: BFS/DFS rely on the following behavior of stacks/queues:

1. If you reinsert an item that is *already* in the stack, then that item moves to the top of the stack (i.e., it will be popped next).
2. If you reinsert an item that is *already* in the queue, then that item keeps its place in the queue.

You can remember these rules by assuming items are 'eager' to be popped, hence they don't want to give up their spot in a queue (line), but would like to jump up to the top of the stack if offered.

4.2.2 Terminology for Today

- *Path*: sequence of nodes $p_1, \ldots, p_k$ connected by edges. (Often paths are assumed to be acyclic; we won't rely on this assumption explicitly today but you can if you'd like.)
- *Length of path*: sum of the weights of the edges $p_1 \to p_2$, $p_2 \to p_3$, etc.
- *Shortest path* between x and y: path starting at x and ending at y with minimal length. (We'll assume unique, but all our analysis will work if nonunique too.)
- *Distance* between x and y: length of the shortest path between those two.

4.2.3 Today's Lecture

On Monday we saw the `explore` routine, which lets us find all the nodes that can be reached from a specific start node. However, explore only tells us which nodes are reachable from the start node; it doesn't tell us: **how to get there** and **how far away** those nodes are. For example, in the map scenario, we may want to know not just what cities we could possibly drive to, but which of those we can get to the fastest and how we should do so.

Today we'll talk about algorithms that solve this more general problem.

Problem 1. *The* single-source shortest paths *(SSSP) problem is to determine, given a start node s, what the shortest path to every other node in the graph is and how long those paths are.*

Here the *length* of a path through the graph is the sum of the weights on the edges in the path. (Insert example in lecture.)

We'll talk about two SSSP algorithms today, Dijkstra's and Bellman-Ford. Both algorithms have the same structure:

- They keep track of a *distance table* `Dist`, where `Dist`$[x]$ keeps track of the length of the shortest path found so far from s to x.
- They keep track of a *predecessor table* `Pred`, where `Pred`$[x]$ keeps track of the last edge you should take in the shortest path from s to x.
- At each step they choose an edge to *relax* on (see below).

The key operation behind both is *relaxation* of edges. Relaxation says: if I know how to get to a in distance `Dist`$[a]$, and there is an edge $a \to^w b$, then I should know how to get to b in distance `Dist`$[a] + w$. Hence, if `Dist`$[b] >$ `Dist`$[a] + w$, we should update `Dist`$[b]$ using this shorter path.

Code for this `Relax` operation is shown below:

```
1 def relax(graph, dist, pred, edge):
2     if dist[edge.dst] > dist[edge.src] + edge.weight:
3         dist[edge.dst] = dist[edge.src] + edge.weight
4         pred[edge.dst] = edge.src
```

Dijkstra's algorithm and Bellman-Ford both call `Relax` repeatedly; they differ only in what order they call `Relax`.

Dijkstra's Algorithm

The first algorithm, Dijkstra's algorithm, solves this problem quite efficiently *when there are no negative edges* — if given a graph with negative edges, it might return the wrong answer.

In Dijkstra's algorithm, at each step you find the node that is *closest to your start node* (excluding nodes you've already visited) and visit it by relaxing on all of its outgoing edges. Pseudocode is below.

```
def dijkstra(graph, start):
    dist = {node: infinity for node in graph}
    dist[start] = 0
    pred = {node: none for node in graph}

    visited, not_visited = set(), set(graph.nodes)
    while not_visited:
        # returns the item in not_visited that has the smallest dist[node]
        # value.
        node = not_visited.pop_smallest_dist()
        visited.insert(node)
        for edge in graph.Out(node):
            relax(graph, dist, pred, edge)
    return dist, pred
```

The `pop_smallest_dist` method can be implemented in $O(\log n)$ time by storing `not_visited` in a heap; we assume `relax` is modified to update the heap as needed, hence it also takes $O(\log n)$ time.

Why does Dijkstra's work? The key is that on every iteration, `Dist` has the true shortest distances to each node *considering paths that use only the nodes in* ***`visited`*** (and perhaps the node itself), and this invariant is maintained throughout. Additionally, once a node is placed in `visited`, that is the shortest path to that node. In particular, this means Dijkstra's algorithm *visits nodes in order of distance from the start.*

In this sense, Dijkstra's algorithm is sort of like a supercharged version of BFS that visits the nearest node first in a way that takes into account the edge weights.

Example of Dijkstra's Algorithm (Do on board)

Correctness of Dijkstra's Algorithm

Theorem 27. *For graphs with nonnegative edge weights, Dijkstra's algorithm solves the single-source shortest path problem.*

Proof. (Sketch; there are also proofs in CLRS and another on Wikipedia you can refer to if this doesn't convince you! We're also assuming here all shortest paths are unique, but the algorithm also works if there are nonunique shortest paths.) The following two-part invariant is maintained at every iteration of the outer loop and for every node x:

- For any node in `visited`, the shortest path to it involves only other nodes in `visited`. In particular, once a node is added to `visited`, its `Dist` and `Pred` will not change.
- `Dist`$[x]$ is the length of the shortest path from `start` to x where all the intermediate nodes between those two are in `visited`; in other words, it's the shortest path of the form $p_1, \ldots, p_k$ where $p_1 =$ `start`, $p_k = x$, and $p_2, \ldots, p_{k-1} \in$ `visited`.

Why is this the case? It can be directly checked for the first two loop iterations.

After that, suppose it is true at the start of some loop iteration; we need to ensure it's true after popping `node`, adding `node` to visited, and then relaxing all the outgoing edges from `node`.

First, let's prove that the shortest path to `node` involves only other nodes in `visited`. Suppose there were some *shorter* path $p_1, \ldots, p_k =$ `node` where some $p_i \notin$ `visited`. In particular, let p_i be the *first* such node in the path. Because the weights are nonnegative, the path $p_1, \ldots, p_i$ is a path with smaller distance than $p_1, \ldots, p_k$, but that means p_i would have been popped as `node`, not p_k! So that case cannot happen, and hence we know that the shortest path to `node` involves only other nodes in `visited`.

Now let's prove the second property: `Dist`$[x]$ is the length of the shortest path from `start` to x going through only nodes in `visited`. Well, let $p_1, \ldots, p_k = x$ be the shortest path to x going through only nodes in `visited`. We consider three cases:

1. If `node` is not involved in the path, then `Dist`$[x]$ will properly remain unchanged after relaxing on the edges leaving `node` due to the inductive hypothesis (it already had the shortest such path).

2. If $p_{k-1} =$ `node`, then the relaxation on the edge `node` $\to x$ will properly set `Dist`$[x]$ to be the length of this path.

3. If $p_i =$ `node` for some $i < k-1$, then $p_1, \ldots, p_i, p_{i+1}$ is a shorter path to p_{i+1}, which was already in `visited`, but that's a contradiction with the inductive hypothesis that shortest paths never change once the node is added to `visited`.

Hence, the second part of the invariant also holds and we complete the proof. □

Running Time of Dijkstra's Algorithm Recall the `pop_smallest_dist` method can be implemented in $O(\log n)$ time by storing `not_visited` in a heap; we assume `relax` is modified to update the heap as needed, hence it also takes $O(\log n)$ time. Each node (out of n nodes) is visited at most once by the outer loop, and each edge (out of m edges) is visited at most once by the inner loop, so the total time is $O((n+m)\log n)$. (You can improve the performance even more than this, but those optimizations are outside the scope of this class.)

Aside: A-star Out-of-scope for this class, but if you take an AI class you'll learn about the famous A^* (pronounced 'A-star') algorithm that improves the performance of Dijkstra's when you're trying to reach a specific goal node.

Bellman-Ford

Bellman-Ford avoids the issues that Dijkstra's has with negative-weight edges; it works for any graph that has no negative-weight cycles. Bellman-Ford is actually a bit simpler than Dijkstra's: relax on *all* the edges, and do this n times! (In fact, you can get away with $n-1$ times.) Here's the pseudocode:

```
def BF(graph, start):
    dist = {node: infinity for node in graph}
    dist[start] = 0
    pred = {node: none for node in graph}

    for i in len(graph.nodes):
        for edge in graph.edges:
            relax(graph, dist, pred, edge)
    return dist, pred
```

(Note in class I think the outer loop goes from $i = 1$ up to $i = n$; this loop goes from $i = 0$ up to $i = n-1$. The difference doesn't matter to this class.)

Example of Bellman-Ford (Do on whiteboard)

Correctness of Bellman-Ford Why is Bellman-Ford correct? The key idea is that after the ith outer-loop iteration, the algorithm has accounted for all paths with $\leq i$ edges. Then the $i+1$th iteration tries to extend those paths by one edge, i.e., looks for paths having $i+1$ edges that are even shorter. It ends once it's considered all paths involving at most n edges; but acyclic paths among n nodes can have at most $n-1$ edges, so at that point, we've considered all possible acyclic paths.

Theorem 28. *For graphs without negative-weight cycles, Bellman-Ford solves the SSSP problem.*

Proof. (Sketch, see CLRS for extended version.) The outer loop maintains the following invariant right before every outer loop iteration i and for every node x:

- `Dist`$[x]$ is *at most* the length of the shortest path from `start` to x out of all such paths having i edges.

It's true at the very first loop iteration, because the only such path is the singleton path `start`.

Suppose it's true at the start of iteration i; then we need to prove it remains true for paths of length $i+1$ after relaxing on all the edges. Well, let $p_1, \ldots, p_{i+2} = x$ be the shortest path with $i+1$ edges between $p_1 =$ `start` and some node x. By the IH, `Dist`$[p_{i+1}]$ is *at most* the length of $p_1, \ldots, p_{i+1}$, hence when we relax on the edge $p_{i+1} \to p_{i+2}$ it will ensure that `Dist`$[p_{i+2}]$ is *at most* `Dist`$[p_{i+1}] + E(p_{i+1}, p_{i+2})$, i.e., the length of $p_1, \ldots, p_{i+2}$ as desired.

Finally, note that every acyclic path in a graph of n nodes involves at most $n-1$ edges, so this guarantees after n iterations of the outer loop that we will have found the shortest path to every node. (Really we could have stopped after $n-1$ iterations, but the difference isn't asymptotically meaningful.) □

Running Time of Bellman-Ford Relaxation in Bellman-Ford takes $O(1)$ time (it just updates the two tables) and relaxation happens once per edge, once per node, hence in total the time taken is $O(nm)$.

4.2.4 Dynamic Programming

Dynamic programming is a weird name for the following algorithm design process:

1. First, write a recursive solution to your problem that solves a big instance by combining the solutions to smaller instances (like divide-and-conquer!)
2. Then, *memoize* the recursion to avoid doing repeated work.
3. Third (usually optional), turn it into a bottom-up, table-filling algorithm.

Fibonacci

Recall the friendly Fibonacci sequence: it has the base cases $F_0 = F_1 = 1$ and the recurrence relation $F_i = F_{i-1} + F_{i-2}$. A natural way to compute the Fibonacci sequence uses recursion, as shown below:

```
1 def fib(n):
2     if n <= 1: return 1
3     return fib(n-1) + fib(n-2)
```

What is the running time of this recursive code? We won't work through a rigorous proof here, but it's not very good at all: if you start drawing a recursion tree, you'll see there are about $\sqrt{2}^n$ many nodes!

One critical thing to notice in the recursion tree, however, is that *many values are recomputed multiple times*! We can avoid this duplicated work by *memoizing* the computation: add a 'memory' dictionary that remembers what values we've already computed and just spits them out if you ever see them again. The below code does this:

```
1 memo = {}
2 def fib_memo(n):
3     if n <= 1: return 1
4
```

```
    if n in memo: return memo[n]
    memo[n] = fib_memo(n-1) + fib_memo(n-2)
    return memo[n]
```

(This sort of transformation can be made automatically to most recursive functions. In the future, we'll indicate this just by adding a `@memoize` annotation before the function.)

Now, what happens when we run this code? The left-most recursion branch will compute fib of n, $n-1$, $\ldots$, 1, and all of those values will be saved in the memo dictionary. Then, all the other recursive calls will be cut short, because their answers have already been computed and saved, and so they'll return immediately. This makes the recursion tree must smaller, and in fact, brings the overall time complexity down to $O(n)$.

Dynamic Programming/Memoized Recursion: Measuring Time

Usually, once you memoize your recursive function, the running time is just dominated by the *non-memoized* calls, i.e., the number of unique recursive inputs that can be produced during the recursive calls and the time taken to compute those the first time they are requested. For example, when computing the Fibonacci of n, the only possible recursive calls it can make are on the smaller numbers $0, 1, \ldots, n$, and each of those calls does only $O(1)$ work other than recursing, hence in total it's $\Theta(n)$ time to compute Fibonacci of n.

Fibonacci, Bottom-Up

It is also common to interpret such memoized, recursive functions in a "bottom-up" fashion, where you fill in the memo with input–output pairs of increasing size until you get to the number you're interested in. This approach for Fibonacci is shown below:

```
def fib_bottomup(n):
    memo = {}
    for i = 0, 1, 2, 3, ..., n:
        if i <= 1:
            memo[i] = 1
        else:
            memo[i] = memo[i-1] + memo[i-2]
    return memo[n]
```

(In fact, in the Fibonacci case, you can improve the space usage to $O(1)$ by only keeping around the last two entries in the memo table!)

Floyd-Warshall

Floyd-Warshall is a dynamic programming algorithm for solving the *all-pairs shortest path* problem: this is just like the single-source shortest path, except we want to know the distances between *all* pairs of nodes, rather than fixing a single start node.

Pseudocode for a top-down version of Floyd-Warshall is below. It assumes your nodes are numbered $0, 1, \ldots, n-1$.

```
@memoize
def FW_distance(graph, start, end, k):
    if k == -1:
        if start == end: return 0
        if E(start, end) is empty: return infinity
        return E(start, end)

    # the best path from start -> end using only nodes 0, 1, ..., k
    without_k = FW_distance(graph, start, end, k-1)
    with_k    = FW_distance(graph, start, k, k-1) + FW_distance(graph, k, end, k-1)
```

```
11      return min(without_k, with_k)
12
13  def FW_allpairs(graph):
14      return [[FW_distance(graph, start, end, len(graph.nodes)-1) for end in graph.nodes]
15              for start in graph.nodes]
```

The key is the `FW_distance` method. It gives the length of the shortest path from `start` to `end` *out of all such paths that only go through nodes* $0, 1, \ldots, k$. The recursive case says: any path from `start` to `end` that only goes through the nodes $0, 1, \ldots, k$ is either:

- A path from `start` to `end` that only goes through the nodes $0, 1, \ldots, k-1$, or
- A path from `start` to k that only goes through the nodes $0, 1, \ldots, k-1$, followed by a path from k to `end` that only goes through the nodes $0, 1, \ldots, k-1$.

This top-down implementation runs in time $O(n^3)$ because there are n possibilities for each of the three arguments, and the function does $O(1)$ work excluding its recursive calls. You can turn this recursive/top-down algorithm into a bottom-up algorithm, and you can also modify it to store predecessors like we saw in Bellman-Ford and Dijkstra's.

Unless we cover it in-class on Friday, we won't require you to know how Floyd-Warshall works, but you should know that it solves the all-pairs shortest paths problem in $O(n^3)$ time.

About the Name

It's customary to include the following quote from Richard Bellman, who is credited with inventing/naming DP, and whic hI've copied from Wikipedia where they attribute it to the autobiography *Eye of the Hurricane: An Autobiography*:

> The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

4.2.5 Summary of Today

We saw:

1. Dijkstra's algorithm for SSSP, which takes time $O((n+m)\log n)$ for the version we saw, but might break if there are negative-weight edges.
2. The Bellman-Ford algorithm for SSSP, which takes time $O(nm)$ and works as long as there aren't negative-weight cycles.

3. The idea of *dynamic programming*, where you write down the results of a recursive function so that you don't have to do repeated work.

4. The Floyd-Warshall algorithm for all-pairs shortest paths, which takes time $O(n^3)$ rather than the naïve $O(n^2m)$ it might take Bellman-Ford.

4.2.6 Post-Lecture Ed Notes

Also, one thing I wanted to reiterate is that for the graphs section, since it's so tightly packed into these two weeks, we won't have time to go through the algorithms and correctness proofs in detail in lecture. The hope is that, since we've spent most of this quarter practicing that skill, you can start to practice doing those steps on your own. I definitely suggest reading the proofs in the lecture notes posted to Canvas as well as the textbook chapters. For graphs, Erickson really shines.

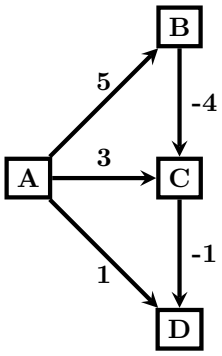
In particular, I think Erickson, SW, and CLRS all explain where the term "relax" comes from, and you may find that reading about that will help you internalize how the shortest-path algorithms we saw today work.

We saw a few different ways of computing fibonacci today, but if your computer can do real number operations (addition, power, multiplication, etc.) in $O(1)$ time then you can actually compute Fib(n) in $O(1)$ time! See `https://en.wikipedia.org/wiki/Fibonacci_sequence#Relation_to_the_golden_ratio` and the C code above, but note that approach is out-of-scope for this class.

Name: _____ **Stanford ID Number:** _____

4.2.7 Lecture 7/31 EC Question

We said Dijkstra's algorithm doesn't necessarily work in the presence of negative-weight edges. Try running Dijkstra's algorithm on the following graph, which has a negative-weight edge. Start the algorithm at node A. What should the shortest path to D be, and what did Dijkstra's algorithm report? What went wrong? (The last part is a somewhat open-ended question.)



4.3 Lecture 8/2: Minimum Spanning Trees

4.3.1 Terminology for Today

Today we'll be assuming **undirected** graphs.

- An *undirected edge* $a \leftrightarrow b$ refers to both the edges $a \to b$ and $b \to a$ as one. Today we'll always be referring to *undirected* edges.
- A *cut* is a subset $S \subseteq V$ of the vertices in a graph.
- An edge $u \leftrightarrow v$ *crosses the cut* S if exactly one of u, v is in S.
- A *light crossing edge* of cut S is an edge $u \leftrightarrow v$ that crosses S and has minimal weight among all such edges.
- A *spanning tree* of an undirected graph is a subset T of undirected edges in the graph such that (1) every node is connected to every other node in the graph, and (2) there are exactly $|V|-1$ (undirected) edges in T.
- The *weight* of a spanning tree T is the sum of all the edge weights in T (only count the weight on each undirected edge once).

(Do some TPS on these definitions — we should have time today.)

4.3.2 The Minimum Spanning Tree Problem

Suppose you're building a new university, and you want to connect all of the buildings with network cables so computers can communicate between buildings. You can model this scenario as an undirected graph, where nodes represent buildings and there is an edge $a \leftrightarrow^w b$ if placing a cable between a and b costs w dollars.

What is the cheapest set of wires you can lay that connects all the computers together?

The graph problem that captures this question is called the *minimum spanning tree problem*:

Definition 22. *Given a connected, undirected graph, the* minimum spanning tree problem *is to find a spanning tree T of the graph that has minimum weight.*

(Why require it to be a tree? Basically boils down to laying as few wires as possible. If the edges are positive, the tree requirement is implied by just saying all nodes are reachable.)

In general, a single graph may have multiple MSTs. For example, if the graph is unweighted, then all spanning trees are minimal.

4.3.3 The Greedy Algorithm Idea

All the algorithms we see today are going to be *greedy algorithms*. In general, we'll think of greedy algorithms like this:

1. Start with a simple initial solution (e.g., an empty set of edges)
2. Consider a set of *candidate extensions* to that solution (e.g., what edge to add to your set of edges), and pick the best one out of those candidates (e.g., the one that adds the least to the total weight).
3. Repeat.

In some cases, as we'll see, we can prove that such a greedy algorithm solves the problem. In general, whether or not the greedy algorithm correctly solves the problem will depend on the set of candidate extended solutions you consider, and how you measure which one is the best.

4.3.4 The Cut Lemma

Both of the algorithms we'll see today work by making a *cut* in the graph, and then finding a light crossing edge of the cut. They extend the partial solution by adding in that light crossing edge. The below lemma tells us that this process is sound: as long as we keep doing this, we will always be able to extend our partial solution into a 'real' MST.

Lemma 4. *Suppose A is a subset of the edges in some MST, S is a cut that no edge in A crosses, and $u \leftrightarrow v$ is a light crossing edge of S. Then, $A \cup \{u \leftrightarrow v\}$ is also a subset of the edges in some MST.*

Proof. By the assumptions on A, there exists an MST T such that $A \subseteq T$. Notably, there is some walk $p_1 \leftrightarrow p_2 \leftrightarrow p_3 \leftrightarrow \cdots \leftrightarrow p_{n-1} \leftrightarrow p_n$ from $p_1 = u$ to $p_n = v$ already in T. Because $u \leftrightarrow v$ crosses the cut S, one of those edges must cross S as well. Let i be chosen such that $p_i \leftrightarrow p_{i+1}$ is the first such edge that crosses S. Notice then, because $u \leftrightarrow v$ is a light crossing edge of S, that the weight of $u \leftrightarrow v$ is at most the weight of $p_i \leftrightarrow p_{i+1}$. Furthermore, because A did not cross S, $(p_i \leftrightarrow p_{i+1}) \notin A$.

Now, consider the set of edges $T' = (T \setminus \{p_i \leftrightarrow p_{i+1}\}) \cup \{u \leftrightarrow v\}$. It still connects all nodes: any walk that would have used the edge $p_i \leftrightarrow p_{i+1}$ should replace that edge with $p_i \leftrightarrow p_{i-1} \leftrightarrow \cdots \leftrightarrow u \leftrightarrow v \leftrightarrow p_{n-1} \leftrightarrow \cdots \leftrightarrow p_{i+1}$. And it has smaller (or equal) weight as T, since $p_i \leftrightarrow p_{i+1}$ had greater than or equal to weight compared to the edge $u \leftrightarrow v$ we replaced it with. So T' is an MST that contains $A \cup \{u \leftrightarrow v\}$, as desired. □

4.3.5 Prim's Algorithm

Prim's algorithm repeatedly grows the set of nodes reachable. On each iteration, it adds the edge of minimum weight that connects any node already reached with a node not yet reached. The cut property guarantees this is a safe choice: take the cut to be the reached nodes.

```
def prim(graph):
    start_node = (pick any node in graph)

    reached = {start_node}
    outgoing_edges = graph.Out(start_node)

    MST_edges = set()

    while len(reached) != graph.n_nodes:
        edge = outgoing_edges.pop_lightest()
        if edge.dst in reached: continue

        MST_edges.insert(edge)
        reached.insert(edge.dst)
        for out_edge in graph.Out(edge.dst):
            outgoing_edges.insert(out_edge)

    return MST_edges
```

Correctness of Prim's

Theorem 29. *Prim's algorithm correctly finds an MST given any unweighted, connected graph.*

Proof. (Sketch) Main loop maintains the invariant that `MST_edges` is always a subset of some MST (follows by the cut lemma with the cut `reached`). Only terminates once `MST_edges` includes all the nodes, hence is itself an MST. □

Example of Prim's

(Do on board)

Running Time of Prim's

We can use a heap to store the set of edges. The running time is dominated by line 16, which runs exactly once for each of the m edges in the graph and does $O(\log m)$ work each time it's executed, hence in total the running time is $O(m \log m) = O(m \log n^2) = O(m \log n)$, because $m = O(n^2)$.

(Out-of-scope: With a Fibonacci heap, you can apparently take the running time of Prim's down to $O(m + n \log n)$. Even with a normal heap, you can slightly improve the constant factors on the $O(m \log n)$ by checking whenever you insert an edge to the heap whether an edge to that node already exists; if so, either replace it or do nothing depending on the weight of that existing edge.)

4.3.6 Kruskal's Algorithm

In Kruskal's algorithm, we repeatedly grow and connect islands together. On each iteration, it adds the edge of minimum weight that connects two disjoint islands. We'll use union-find to store the islands. Correctness of Kruskal's algorithm again follows from the cut property; take the cut S to be the set of all nodes in the same island as the edge's source.

```
1 def kruskal(graph):
2     MST_edges = set()
3     for edge in sort_by_weight(graph.edges):
4         if not same_set(edge.src, edge.dst):
5             MST_edges.insert(edge)
6             union(edge.src, edge.dst)
7     return MST_edges
```

Correctness of Kruskal's

Theorem 30. *Kruskal's algorithm correctly finds an MST given any unweighted, connected graph.*

Proof. (Sketch) Same argument, except take the cut to be the island of nodes in the same set as the source node of the newly added edge. □

Example of Kruskal's

(Do on board)

Running Time of Kruskal's

It takes $O(m \log m) = O(m \log n)$ time to sort the edges, followed by $O(m\alpha(m))$ time to process all of them. This is dominated by the former, i.e., the time to run Kruskal's is $O(m \log n)$.

But, if you already have the edges in sorted order for some reason, then Kruskal's can be run in only $O(m\alpha(m))$ time, which is just about linear!

4.3.7 Greedy Algorithms More Generally

Both Prim's and Kruskal's were examples of the *greedy algorithm design pattern*: make your partial solution a little better at each step.

In general, when designing an optimal greedy algorithm, you need to:

1. Design a *heuristic* that tells you what is the best choice to pick at each step,
2. Prove that, if there is an optimal solution extending partial solution A, and the heuristic says to extend to partial solution A', then there's still an optimal solution extending A'.

In Prim's algorithm, the heuristic involved picking the lightest edge connecting a reached node with an unreached node. In the toposort, we greedily added any class that we had satisfied all the prereqs for (this worked because taking a class never prevents us from taking another class).

Hints for Designing Heuristics

A good thing to ask yourself when designing heuristics is: what is the "least bad" thing I can pick? (E.g., the lightest weight edge).

Another Greedy Algorithm Example: Activity Selection

Let's look at a classic example of greedy algorithms: activity selection. Suppose you have n activities: activity a_i starts at time s_i and finishes at time f_i. You want to sign up for as many activities as possible, but cannot sign up for overlapping activities.

We're going to try using the greedy strategy. What heuristic should we pick? What we want to avoid is picking an activity that will restrict what other activities we can do in the future. So let's try picking the activity that *finishes first*: this will give us as much remaining time in the day to do other activities!

Lemma 5. *Let A be a set of nonconflicting activities that can be extended into a set $B \supsetneq A$ of nonconflicting activities with maximal size. Let a_i be the activity that does not conflict with anything of A and finishes earliest. Then, there is another set $B' \supset A \cup \{i\}$ of nonconflicting activities that extends $A \cup \{a_i\}$ and has identical (maximal) size to B.*

Proof. Let $a_j \in B \setminus A$ be the earliest-finishing activity in B but not A. By assumption in the problem, a_i finishes *at least as early* as a_j, i.e., $f_i \leq f_j$.

Let the new set be $B' \coloneqq (B \setminus \{a_j\}) \cup \{a_i\}$. Consider any activity $a_k \in B' \setminus A$; we need to show it doesn't conflict with a_j. But we assumed a_j was the earliest-finishing activity in $B \setminus A$, so a_k must start *after* a_j finishes, i.e., $f_j \leq s_k$. But we already saw that $f_i \leq f_j$, hence $f_i \leq f_j \leq s_k$ and so a_i and a_k do not conflict. Hence no two activities in B' conflict, and since B' has the same size as B we've completed the argument. □

This leads to the following natural greedy algorithm:

```
1 def select_activities(activities):
2     selected, unselected = set(), sort_by_finish(activities)
3     while unselected:
4         choice = unselected.pop_earliest_finish()
5         if choice conflicts with selected:
6             continue
7         selected.add(choice)
8     return selected
```

Note the finish times of selected grow monotonically, so line 5 can be implemented by checking only whether `choice` conflicts with the most recently selected activity, i.e., in $O(1)$ time. So the overall time is $O(n \log n)$ time to sort plus $O(n)$ time to select activities, for a total time of $O(n \log n)$.

4.3.8 Post-Lecture Ed Notes

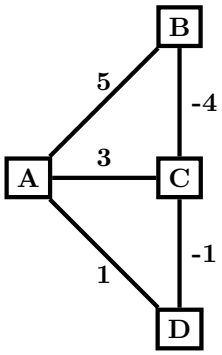
Caveats about code from previous posts still apply. Code for these algorithms is particularly interesting because they combine a lot of algorithms we've seen previously (Prim's uses heaps, Kruskal's uses both sorting and union-find, activity selection uses sorting!).

There was a question about doing MST on directed graphs; per some internet searching, it seems like the analogue of a spanning tree in that setting is a "spanning arborescence" and Edmond's algorithm is suggested for finding a minimum-weight spanning arborescence: `https://en.wikipedia.org/wiki/Edmonds%27_algorithm`

Name: _____ **Stanford ID Number:** _____

4.3.9 Lecture 8/2 EC Question

Run Prim's algorithm on the following graph, using C as the start node.
What is the weight of the resulting MST?



4.4 Week 6 Problem Sheet

Read This First: As I hopefully remembered to mention in class, the next two weeks will be heavily focused on algorithm design rather than algorithm analysis. The main thing you need to do to learn algorithm design is to practice, and practice a lot. This has two benefits. With an extreme amount of practice, you'll be able to try out and reject failing approaches in your head much faster, which lets you do more guess-and-check/trial-and-error than you otherwise would have been able to do. Practice also helps you get better at identifying almost-subconscious patterns that hint at when to use which technique and how to adapt that technique to the application domain.

My strong suggestion is to ignore the problems below and instead go to `https://leetcode.com/problemset/` and spend as much time as you have solving problems there. You should start with the 'easy' problems and then increase difficulty if you find them to be actually easy. You can filter by topic. When you design an algorithm to solve one of those problems, you should also analyze its running time.

In case you already feel comfortable with algorithm design, or have finished all of Leetcode, I've added below some problems that introduce you to algorithms and algorithm techniques that have been useful to my own work/research. They are mostly tilted towards compilers and programming languages work. I've *tried* to make them sort of approachable to folks without background in that area, but they probably still require a fair bit of application domain knowledge to follow. So **do not make the mistake** of spending hours trying to understand what those problems are asking, when you could be spending those same hours grinding leetcode; the problems below are not magically more important or pedagogically useful than the problems on leetcode.

A note on OHs: The CAs this week have also been instructed to ignore this homework assignment and instead focus on Leetcode-style problems in OHs. So if you have an interest in any of these problems, you're probably best reaching out to me individually. But overall **you should consider these problems out-of-scope for this class,** similar to the post-lecture notes I usually give on Ed.

4.4.1 Lecture 7/29: Welcome to Graphs

Problem 51: Recall from 103 the notion of regular languages.

1. Given two regular expressions (or NFAs, or DFAs), determine if they accept the same set of words or not. Your algorithm should take time proportional to the number of reachable states in the product DFA for the two regular expressions plus the size of the NFAs for the two regular expressions.

2. Describe a class of inputs where BFS would perform better on this task and another class of inputs where DFS would perform better.

3. Extend your solution to part (1) to support *infinite state machines*, e.g., Turing machines. If you'd like, you can assume the infinite state machine is encoded by a state transition matrix that you can query one row of at a time (i.e., you can ask for each state and input what the next state would be).

4. Prove (perhaps with reference to a theorem you learned in 103) that it is impossible for any algorithm to 'truly' solve the problem described in part (3).

5. Finally, modify your answer to part (3), if needed, in order to prove that it is a *recognizer*: when given any two nonequivalent programs, your procedure will eventually halt and report them nonequivalent. Does the same theorem hold if you used a different one of BFS/DFS?

(If you like this problem, you might find the concept of 'model checking' interesting.)

Problem 52: Consider a program written in a simple programming language with the following instructions: `set x = 0`, `copy x = y`, `increment x++`, `if x = 0 then skip next line`, `goto line n`. The program halts once it tries to execute past the end of the instruction list. Here's an example

program:

```
[line 0] set y = 0
[line 1] set x = 0
[line 2] if x = 0 then skip next line
[line 3] increment x++
[line 4] goto line 6
[line 5] increment y++
[line 6] increment x++
```

1. Line 5 is *dead code*: there's no way to reach it. Removing dead code helps speed up compilation and save space on small embedded devices. Devise an algorithm for identifying dead code; nodes should be lines in the program, and edges should represent syntactic reachability (i.e., line 2 has an edge to both line 3 and line 4).

2. Line 3 is also dead code, but this requires a slightly more semantic analysis to determine this. Extend your algorithm from part (1) so that it can detect that line 3 is dead. Nodes should now be a pair (l, s) where l is a line number and s is a set of variables that are definitely zero.

(If you find these ideas interesting, you might want to look into static analysis and abstract interpretation.)

Problem 53: Consider a graph G, a *source node* s with no incoming edges, and a *target node* t with no outgoing edges. We say a node x *dominates* t if every path from s to t goes through x. Given an **acyclic** graph G with n nodes and m edges, along with two nodes s, t, describe how to identify the set of all dominators in time $O(n + m)$.
(Variants of the above problem are used to perform conflict analysis in virtually all competitive modern SAT solvers. If you would like to learn more about it, you might consider reading the paper "GRASP — A New Search Algorithm for Satisfiability" or searching for "CDCL UIP." They also appear frequently in compiler optimizations, you may want to search for "dataflow dominators.")

Hint: Hint(s) are available for this problem on Canvas.

Problem 54: In programming languages like C the programmer is expected to manually call the FREE function to release heap memory objects that they know they will never need to refer to again. In higher-level languages, it is more common to use *automatic memory management* techniques that automatically free unused memory for you. The most popular of these is *tracing garbage collection:* interpret the heap as a graph, where nodes correspond to objects in memory and edges correspond to pointers between those objects. Suppose you have paused a program and have access to the heap graph as described above. Explain how to identify objects in memory that can be safely freed.

(There's a lot of work on garbage collection algorithms; if you're interested, you may want to start by skimming the Garbage Collection Handbook or the relevant sections in TAOCP.)

Problem 55: Leetcode topics: Breadth-First Search, Depth-First Search, Topological Sort. `https://leetcode.com/problemset/` For each problem, provide: a description of the graph (nodes and edges), an algorithm solving it, and an asymptotic analysis of the running time. Also, justify any design choices you made (BFS vs. DFS, etc.).

4.4.2 Lecture 7/31: Shortest Paths and Dynamic Programming

Problem 56: Given a context-free grammar and a string s, devise a dynamic programming algorithm to parse s according to the grammar. If the string has length n and the grammar has constant size, your algorithm should parse the string in time $O(n^3)$. *(The algorithm you will likely rediscover is called CYK parsing. If you're interested in learning more about this, you may also wish to search up Earley parsing, PEG parsing, and LR parsing.)*

Hint: Hint(s) are available for this problem on Canvas.

Problem 57: Do the same as Problem 5 except for Leetcode topics: dynamic programming and shortest path. You may also find good problems in CLRS chapter 14.

4.4.3 Lecture 8/2: Prim and Kruskal

Problem 58: *(This problem works through the Euler Tour Technique for storing spanning trees. It's not too related to today's lecture. The paper "Logarithmic Lower Bounds in the Cell-Probe Model" is good follow-on reading to skim; the author tragically passed away about a decade ago but did some of the most exciting modern work in lower bounds I've seen.)* In previous homeworks, we saw how both Splay trees and 2–3 trees can be used to implement a list data structure with $O(\log n)$ insertion, deletion, random access, split, and join. Suppose you have a forest of trees stored in such a list, where the tree is stored in the list in an order described by the following pseudocode:

```
1 def visit(root):
2     if root is empty: return []
3     return [root] + visit(root.left) + visit(root.right) + [root]
```

Notably, every node in the forest appears twice in some list. Explain how to implement the following operations by modifying the lists:

1. (Cut) Given a node in one of the trees, *cut* it and its subtree out of its parent, i.e., delete the incoming edge to that node. Your algorithm should require $O(1)$ many list operations.
2. (Link) Given a root node in one of the trees and a node in some other tree, describe how to *link* them by inserting an edge between the two nodes. Your algorithm should require $O(1)$ many list operations.
3. (Find-Root) Given two nodes in the forest, determine if they are part of the same tree or not. Your algorithm should require $O(1)$ many list operations.

In all cases, you can assume that a 'node' is given to you as a pointer to the first and last instance of that node in the relevant list. Finally, show how to maintain a maximal spanning forest for an unweighted, undirected graph if the graph grows over time and is described by a sequence of edge insertions. Processing each edge insertion should take $O(\log n)$ time.

Problem 59: Do the same as Problems 5/7 except for the topic: minimum spanning tree.

4.5 Lecture 8/5: Max Flow/Min Cut

Most of the content in this lecture/these notes is adapted pretty directly from Erickson, Chapter 10, used under a Creative Commons Attribution License.

There's a classic story (you can read in Erickson) where the USSR and the US were both looking at maps of the USSR rail system. The USSR wanted to figure out how best to utilize the railways, i.e., what the maximum amount of goods they could get from a starting point s to a target point t, without overloading the capacity of any track. Meanwhile, the US wanted (roughly stated) to find a set of low-capacity (presumably poorly guarded) rail links that they could disrupt that would totally prevent the USSR from sending *anything* from s to t. We'll study both of these problems today: the former corresponds to a *maximum flow* problem, and the latter corresponds to a *minimum cut* problem.

4.5.1 Why learn Max Flow/Min Cut?

See many features of LP solving (especially *duality*) for the first time. Problem with application to many other domains.

4.5.2 Special Assumptions for Today's Lecture

We'll assume we're given a graph of the rail network with the following features:

- The graph is directed and has a specified *source* node s and *target* node t.
- $E : V \times V \to \mathbb{R}_{\geq 0}$, i.e., no edges are empty ($\perp$) and no edges are negative.
- We'll represent missing edges (i.e., if there's track between two cities a, b) by $E(a \to b) = 0$, not $E(a \to b) = \perp$ as we were doing earlier.
- No *antiparallel edges*: for any a, b, if $E(a \to b) > 0$, then $E(b \to a) = 0$. (This constraint can be done away with, but the algorithms are easier to understand if we make this assumption.)
- For clarity, we'll write $E(a \to b)$ sometimes instead of $E(a, b)$.
- We will call the edge weights the *capacity* of the edge, since they represent how many units of supplies we could hypothetically push over that railroad.

4.5.3 Graph Flows and the Max Flow Problem

Terminology. In the below, we'll assume a directed graph G, a specified source node s, and a specified target node t.

- A *flow function* $f : V \times V \to \mathbb{R}_{\geq 0}$ assigns each edge $a \to b$ in the graph some *flow* $f(a \to b)$. For any node a, define the *net flow* $\delta(a)$ to be the flow out of the node minus the total flow into the node, i.e., $\delta(a) \coloneqq \sum_v f(a \to v) - \sum_v f(v \to a)$. Finally, we also require that every flow function satisfy:
  1. The flow along each edge must be *at most* the capacity of that edge: $f(a \to b) \leq E(a \to b)$.
  2. For every node a *except for* s and t, the total flow out of the node is identical to the total flow into the node: $\delta(a) = 0$. (Intuitively, this means no station between s and t can hold on to any goods; they must pass along everything they get.)
- The *weight* of a flow is the net flow leaving s, i.e., $\delta(s)$. Note that this is identical to the net flow *reaching* t, i.e., $\delta(s) = -\delta(t)$.
- We say an edge $a \to b$ is *saturated* by the flow if $f(a \to b) = E(a \to b)$.
- We say an edge $a \to b$ is *avoided* by the flow if $f(a \to b) = 0$.

Problem 2. *The* max-flow problem *is to devise a flow having maximum weight.*

4.5.4 Finding Good Flows: The Ford-Fulkerson Method

The Ford-Fulkerson algorithm for finding maximum flows works as follows:

```
def fordfulkerson(graph):
    flow = {edge: 0}
    while True:
        A = find augmentation
        if no augmentation: return flow
        apply augmentation A to graph
```

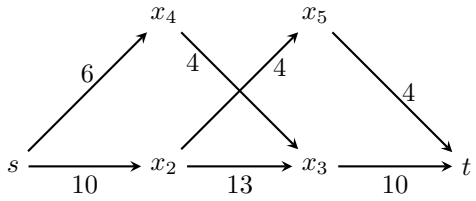
It has a similar flavor to many of the greedy algorithms we've seen: start with a 'simple' flow that doesn't try to send anything. Then, at each step, find some way to improve the flow (we call this an *augmentation*; see below). Keep improving the flow until you can't find any more improvements.

What are augmentations?

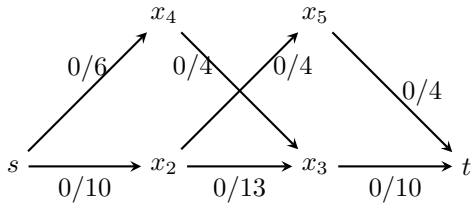
An *augmentation* is a sequence of modifications to a flow, with the following properties:

- For any node x other than s, t: if the augmentation increases the flow entering x, it must then either increase the flow exiting x or decrease some other flow entering x to compensate.
- (Similar symmetric conditions for increasing the flow exiting x, decreasing flow entering x, etc.)

To get a sense for augmentations, consider the following possible rail map, taken from Kevin Wayne's lecture slides (`https://www.cs.princeton.edu/courses/archive/spr04/cos226/lectures/maxflow.4up.pdf`):



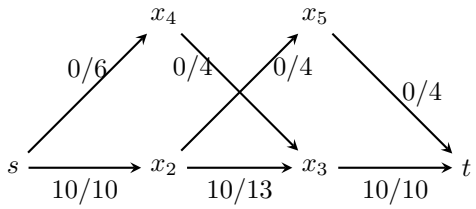
Now, the Ford-Fulkerson method tells us to start with a basic zero flow; we'll draw the flow on each edge before a slash:



Our goal is to *increase the flow out of* s. So we could make the following augmentation:

1. Increase the flow from $s \to x_2$ by 10.
2. Now x_2 has too much incoming flow, so we need to compensate. We can do so by increasing the outgoing flow from $x_2 \to x_3$ by 10.
3. Now x_3 has too much incoming flow, so we need to compensate. We can do so by increasing the outgoing flow from $x_3 \to t$ by 10.

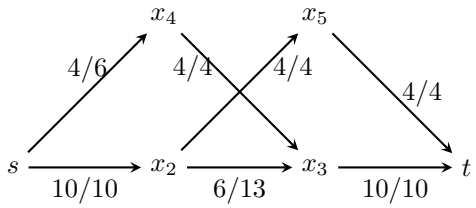
Since t and s don't need to satisfy the $\delta(a) = 0$ condition, we can stop here having successfully increased the flow from s to t:



But increasing the flow isn't always as easy as increasing all the edges along some path. For example, consider the following nonobvious augmentation:

1. Increase the flow from $s \to x_4$ by 4
2. Now x_4 has too much incoming flow, so increase the flow from $x_4 \to x_3$ by 4 to compensate.
3. Now x_3 has too much incoming flow. We can't push any more flow *out* of x_3, but we can decrease the amount of flow coming into x_3 from other sources, i.e., we can decrease the flow from $x_2 \to x_3$ by 4.
4. Now x_2 has too much incoming flow, but we can remedy this by increasing the flow from $x_2 \to x_5$ by 4.
5. Finally, x_5 now has too much incoming flow, but we can fix this by sending it to t by increasing the flow on $x_5 \to t$ by 4.

Although this required the nonobvious move of decreasing the flow from x_2 to x_3, it results in a better flow, of weight 14:



Finding Augmentations: The Residual Graph

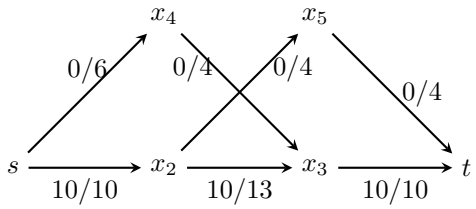
So far, it seems like we're just finding these augmentations by trial-and-error. Is there a more principled way? The answer is yes! The secret is to turn our graph into a related graph called the *residual graph*.

The key thing to remember is the following: an edge $x_i \to x_j$ in the residual graph means "if x_i has too much incoming flow, it can get rid of it by either increasing how much flow it sends to x_j or decreasing how much flow x_j sends to it."

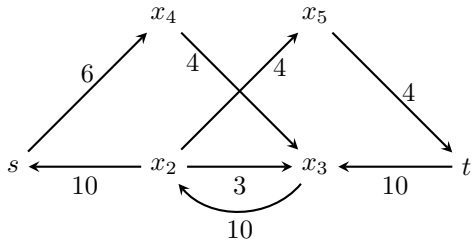
Formally, given a graph (V, E) and a flow f, we define the residual graph (V, E') on the same vertices like so:

1. If $E(a \to b) > 0$, then $E'(a \to b) = E(a \to b) - f(a \to b)$. *(Intuitively, this says: if node a has too many incoming supplies, it can get rid of at most $E(a \to b) - f(a \to b)$ of them by sending them to b along the rail line $a \to b$.)*
2. If $E(a \to b) = 0$, then $E'(a \to b) = f(b \to a)$. *(Intuitively, this says: if a has too many incoming supplies, it can get rid of at most $f(b \to a)$ of them by asking b to send fewer supplies along the rail line $b \to a$.)*

Let's see how we could have used the residual graph to find the last augmentation we just saw. We started with this flow:



And the corresponding residual graph (excluding zero edges) looks like:



It's useful to pause here and restate exactly what some of these edges represent; importantly, edges that go the same direction as those in the original graph have a slightly different meaning from those that go in the opposite direction:

1. The edge $x_2 \to^3 x_3$ in the residual graph means "if x_2 has too much incoming supplies, you can compensate by sending at most 3 units more to x_3."

2. On the other hand, the edge $x_3 \to^{10} x_2$ in the residual graph means "if x_3 has too much incoming supplies, you can compensate by asking x_2 to send it at most 10 fewer units of supplies."

The key idea is that paths from s to t in the residual graph correspond to augmentations in the original graph. For example, the existence of the path $s \to x_4 \to x_3 \to x_2 \to x_5 \to t$ means: if s has some extra supplies to send, it can do so by first sending it to x_4. Then x_4 has too many supplies, but it can get rid of them by sending it to x_3. Now x_3 has too many supplies, but it can solve this by asking x_2 to send fewer supplies. Now x_2 has too many supplies, but it can solve this by sending more to x_5. Finally, x_5 has too many supplies, but it can solve this by sending them to the destination t. This is exactly the augmentation we saw before!

In fact, because of how we defined the residual graph, *any* path from s to t in the residual graph corresponds directly to such an augmentation!

(One small point we haven't discussed yet is exactly how much to augment each edge in the flow by: if you take it to be the minimum edge weight in the path, you'll find everything works!)

(If you are unconvinced by the above exposition of the connection between augmentations and paths in the residual graph, Erickson contains a more algebraic proof.)

Ford-Fulkerson in More Detail

We've now seen that constructing the residual graph gives us an easy way to find augmentations. If we fill this in to the Ford-Fulkerson pseudocode, we get a slightly more detailed procedure:

```
def fordfulkerson(graph):
    flow = {edge: 0}
    while True:
        R = build residual graph(graph, flow)
        path = find path from s to t in residual graph
        if no such path: return flow
        apply augmentation corresponding to path
```

Remaining Question: Is It Optimal??

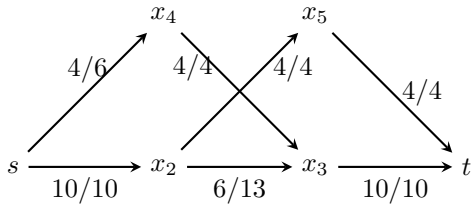
Ford-Fulkerson gives us a way to make better-and-better flows, stopping once there is no path from s to t in the residual graph. But how do we know that the flow it finds is actually the *best*? Maybe there are some more clever ways to modify the flow that aren't represented in the residual graph? In other words, we have yet to show that Ford-Fulkerson actually finds the *maximum* flow. It turns out it *does* find the maximum flow, however, and this is what we'll spend the rest of the day proving! Surprisingly, the proof will take a big detour into the *min-cut problem*.

Before We Move On: Reachable Nodes in the Residual Graph

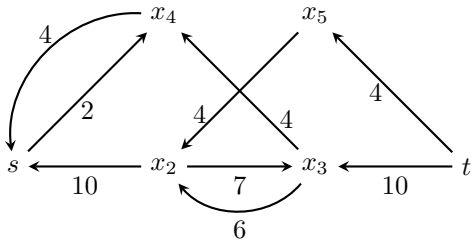
Before we move on, while we're thinking about residual graphs, let's note one interesting fact about the residual graph once Ford-Fulkerson stops:

Lemma 6. *Let C be the set of all nodes reachable from s in the residual graph right before Ford-Fulkerson stops. Then, $s \in C$, $t \notin C$, any edge exiting C is saturated, and any edge entering C is avoided.* (Entering means an edge from a node in C^c to a node C; exiting means the opposite.)

Before proving it, let's see how this lemma works in the running example. After applying the second augmentation, we get this flow:



which corresponds to the following residual graph:



The nodes reachable from s in the residual graph are $C = \{s, x_4\}$, and indeed we see the following in the original (non-residual) graph:

1. Both of the exiting edges ($s \to x_2$, $x_4 \to x_3$) are saturated, and
2. There are no entering edges, hence they're vacuously all avoided.

We can now prove the lemma:

Proof. Suppose for sake of contradiction there were an exiting edge $a \to b$ that wasn't saturated, i.e., $a \in C$ and $b \notin C$ but $f(a \to b) < E(a \to b)$. In that case, by definition of the residual graph, there would be an edge $a \to b$ in the residual graph with weight $E(a \to b) - f(a \to b) > 0$. But then, because $a \in C$ and hence a is reachable from s, b would also be reachable from s in the residual graph. But then $b \in C$, contradicting the assumption $b \notin C$.

Suppose for sake of contradiction there were an entering edge $a \to b$ that wasn't avoided, i.e., $a \notin C$ and $b \in C$ but $f(a \to b) > 0$. In that case, by definition of the residual graph, there would be an edge $b \to a$ in the residual graph with weight $f(a \to b) > 0$. But then, because $b \in C$ in the residual graph, we would also have $a \in C$, a contradiction. □

4.5.5 Graph Cuts

- A *cut* is a subset $C \subseteq V$ of the vertices in a graph.
- An *s–t cut* is a cut C where $s \in C$ and $t \notin C$.
- The *crossing capacity* of an s–t cut C is $\sum_{v \in C} \sum_{u \notin C} E(v, u)$, i.e., the capacity of all the rail edges *leaving* C.

Notice we already saw an s–t cut, specifically the set C of nodes reachable from s in the residual graph upon stopping Ford-Fulkerson. In general, an interesting problem given any graph is the *min-cut* problem:

Problem 3. *The min-cut problem is to find an s–t cut C of minimal crossing capacity.*

4.5.6 Max-Flow, Min-Cut Weak Duality Lemma

The following lemma is called the *weak duality* lemma for max-flow/min-cut. It says that if you have *any* s–t cut and *any* flow, then the absolute best possible thing the flow could do is fully saturate the edges leaving the cut and avoid all the edges going back into the cut. In particular, this means that even the value of the very best flow is at most the capacity of the very best s–t cut.

Lemma 7. *Consider any flow f and any s–t cut C. The weight of f is* at most *the crossing capacity of C. In fact, the weight of f is equal to the crossing capacity of C exactly when all the edges exiting C are saturated, and all the edges entering C are avoided.*

Proof. (Adapted directly from Erickson.) The weight of f is $\delta(s)$, which we know is

$$\begin{aligned}
\delta(s) &= \sum_{v \in C} \delta(v) \\
&= \sum_{v \in C} \left(\sum_{w} f(v \to w) - \sum_{w} f(w \to v) \right) \\
&= \sum_{v \in C} \sum_{w} f(v \to w) - \sum_{v \in C} \sum_{w} f(w \to v) \\
&= \sum_{v \in C} \left(\sum_{w \in C} f(v \to w) + \sum_{w \notin C} f(v \to w) \right) - \sum_{v \in C} \left(\sum_{w \in C} f(w \to v) + \sum_{w \notin C} f(w \to v) \right) \\
&= \sum_{v \in C} \sum_{w \in C} f(v \to w) + \sum_{v \in C} \sum_{w \notin C} f(v \to w) - \sum_{v \in C} \sum_{w \in C} f(w \to v) + \sum_{v \in C} \sum_{w \notin C} f(w \to v) \\
&= \sum_{v \in C} \sum_{w \notin C} f(v \to w) - \sum_{v \in C} \sum_{w \notin C} f(w \to v) \\
&\leq \sum_{v \in C} \sum_{w \notin C} f(v \to w) \\
&\leq \sum_{v \in C} \sum_{w \notin C} E(v \to w),
\end{aligned}$$

which is indeed the weight of the cut.

The first equality follows because, by the definition of a flow, $\delta(v) = 0$ for every node v in the cut C *except for* s, so the sum just adds in a bunch of other zero terms. The second equality expands the definition of $\delta(v)$ and the third regroups the terms in the sum. The fourth equality separates out the nodes according to whether or not they are in C, the fifth again regroups terms, and the sixth cancels out two identical sums (both just count the flow among edges internal to C).

In words, what we've done is add up all the incoming/outgoing edges involving nodes in C. Edges totally internal to C cancel out, and all that's left is the edges flowing in from C^c and out from C to C^c.

Also notice that the first $\leq$ is an equality *only when* the subtracted-off term is zero, i.e., only when there is no flow along the edges into C. Similarly, the second $\leq$ is an equality *only when* all the edges out of C are saturated. □

You can also think of this in a real-world situation: you (meaning, all of C) are sending supplies out to someone else (C^c). You start with n supplies, and end up at the end of the day with zero supplies. At some point, people mistakenly send you a total of m supplies back, but you promptly send those out again to end up with zero supplies at the end of the day. In sum, then, you have sent a *net* of n supplies out, including l directly out and m that you double counted because you got them sent back to you, i.e., $n = l - m$. This is exactly where the final equality in the above algebra comes from.

4.5.7 Max Flow, Min Cut Strong Duality Theorem

Theorem 31. *When Ford-Fulkerson exits, there is a cut with identical crossing capacity to the weight of the flow and hence the flow is maximal and the cut is minimal.*

Proof. We showed in Lemma 6 that when Ford-Fulkerson exits there exists a cut where all the exiting edges are saturated and entering edges are avoided, hence, by Lemma 7 the flow has the same weight as the cut has crossing capacity.

But also by Lemma 7, we know no other flow can have higher weight (because its weight is at most the capacity of the cut), and similarly no other cut can have lower capacity. So these two are indeed optimal. □

4.5.8 Running Time?

How long does Ford-Fulkerson take to run? It depends on which path through the residual graph that you choose. You'll evaluate some different options on HW7 (if you feel like doing so); for now, it's sufficient to know that a good choice (often referred to as the Edmonds-Karp algorithm) is to pick the path in the residual graph that uses the fewest number of edges; for integer-valued weights, this leads to a polynomial-time max-flow algorithm.

4.5.9 Post-Lecture Ed Notes

Note max-flow is sort of a "baby version" of some broader problem called linear programming. The duality between max-flow and min-cut is (essentially) an instance of this broader duality between a linear programming maximization problem and its dual minimization problem. I'll add a problem working through this to the homework.

Also note that the restriction that no antiparallel edges exist is relatively easy to do away with; in the above code, I've handled this by allowing multiple parallel edges in the residual graph and kept track of the meaning of each of those edges to disambiguate. Alternatively, Erickson and CLRS have transformations that allow you to reduce one version of the problem to the other.

Name: _____ **Stanford ID Number:** _____

4.5.10 Lecture 8/5 EC Question

(This is a somewhat open-ended question.) At the start of today, we assumed the graph had no *antiparallel edges*, i.e., if $E(a \to b) \neq 0$ then $E(b \to a) = 0$. What part of our lecture relied on this fact, i.e., which step would be complicated by the existence of a, b with $E(a \to b) > 0$ and $E(b \to a) > 0$?

4.6 Lecture 8/7: More Classic Algorithms

(This lecture is adapted closely from problems and chapters in CLRS.)

4.6.1 DP: Rod Cutting

Suppose you have a large metal rod of length n, and a machine that can cut any rod into smaller pieces (assume all lengths are whole numbers, and you can only cut into whole-number-length pieces). You check on Craigslist, and a lot of people are offering money for rods of different sizes. Let P_s be the market price people will pay you per rod for a rod of length s.

Problem 4. *Assume you are going to cut your rod into pieces and sell all the pieces at market price. What sizes of pieces should you cut to maximize your earnings?*

Let's look at a tiny example: you have a rod of length 2, and the market prices are $P_1 = 2$ and $P_2 = 3$ dollars. You have two options:

1. Cut into two rods of length 1 each, for a total of $2P_1 = 4$ dollars, or
2. Leave it as one rod of length 2, for a total of $P_2 = 3$ dollars.

In this scenario, cutting the rod into two pieces of length 1 is the optimal solution.

A Recursive Solution

Think on your own!

Let's think recursively about this. There are n possible sizes s for the first rod we cut, and if we commit to the first cut having length s, then the best thing we can do is optimally cut and sell the remaining $n - s$ length of rod.

This leads to the recursive algorithm (note we're only showing how to compute the price of the optimal cut, not how to compute the cut itself, but you can modify the algorithm to do so):

```
def best_cut_price(rod_length, prices):
    best_so_far = prices[rod_length]
    for first_cut in 1, 2, ..., rod_length-1:
        total_price = prices[first_cut] + best_cut_price(rod_length - first_cut)
        if total_price > best_so_far:
            best_so_far = total_price
    return best_so_far
```

(TPS: what does the recursion tree look like? What's the running time?)

It *at least* recurses on $n - 1$ and $n - 2$ each time, so it takes at least $\sqrt{2}^n$ time.

Memoization

But, many of those recursive calls are duplicates! So if we memoize, we'll save time. It's fine to just write it like this:

```
@MEMOIZE
def best_cut_price(rod_length, prices):
    best_so_far = prices[rod_length]
    for first_cut in 1, 2, ..., rod_length-1:
        total_price = prices[first_cut] + best_cut_price(rod_length - first_cut)
        if total_price > best_so_far:
            best_so_far = total_price
    return best_so_far
```

Or if you want to be super explicit:

```
memo = {}
def best_cut_price(rod_length, prices):
    if (rod_length, prices) in memo: return memo[(rod_length, prices)]
    best_so_far = prices[rod_length]
    for first_cut in 1, 2, ..., rod_length-1:
        total_price = prices[first_cut] + best_cut_price(rod_length - first_cut)
        if total_price > best_so_far:
            best_so_far = total_price
    memo[(rod_length, prices)] = best_so_far
    return best_so_far
```

(TPS: what is the memoized running time?)

Well, each recursive call recurses on a strictly smaller problem, so we just need to ask how many *unique* inputs are encountered; the duplicates will be memoized when we we need them. The prices dictionary never changes, so it's just the rod lengths, which can be $1, 2, \ldots, n$. So there are n unique calls, each one doing $O(n)$ work, for a total of $O(n^2)$.

Bottom-Up

It's good practice to try and translate this into a bottom-up solution, but it won't change the running time.

4.6.2 DP: Common Subsequence

A string q is a *subsequence* of a string s if q can be formed by dropping characters from s. For example, `abc` is a subsequence of `faaaeoebhuac`.

Many different strings can share a subsequence, e.g., `abc` is a subsequence of both `fabec` and `alphabetic`.

Problem 5. *Given strings s and t, find the* longest *string q such that q is a subsequence of both s and t (this is called the* longest common subsequence*, or LCS).*

A Recursive Solution

(Think on your own/TPS for a while.)

We make the following claims (saying 'the' LCS but there may be multiple):

1. If both s and t start with the same character, i.e., $s_1 = t_1$, then the LCS starts with s_1 followed by the LCS between $s_{2\ldots}$ and $t_{2\ldots}$.

2. Otherwise, the LCS is either the LCS of $s_{2\ldots}$ and t, or the LCS of s and $t_{2\ldots}$.

Both of these are nonobvious enough that you should really prove them. Here's a proof for the first claim; I suggest trying to prove the second as practice (it's in CLRS if you get stuck!).

Proof. Suppose $s_1 = t_1$ and q is an LCS of both s and t. If $q_1 = s_1$, then we're done. Otherwise, q consists entirely of a subsequence between $s_{2\ldots}$ and $t_{2\ldots}$; but then $s_1 \circ q$ is a subsequence between s and t of strictly larger length, so we have a contradiction and hence this case cannot happen. □

(Insert example here, something like: $s =' axbxc'$, $t =' ayybyycy'$. They claim there's an LCS shared between s and t that does not start with $s_1 =' a'$, e.g., bc. But then you could prepend $s_1 =' a'$ onto this to get abc which is a subsequence, longer than the so-called LCS!)

This observation leads to the following recursive program:

```
def lcs(s, t):
    if s == "" or t == "": return ""
    if s[0] == t[0]: return s[0] + lcs(s[1:], t[1:])
    a = lcs(s[1:], t)
    b = lcs(s, t[1:])
    return longest of a, b
```

Runtime pre-memoization

Suppose both s and t start off size n.

(TPS about the runtime pre-memoization.)

At each step you (in the worst case) recurse on two problems, each one smaller. This leads (not exactly, but close enough) to a binary tree having depth n, hence $\approx 2^n$ time. Not great!

Runtime post-memoization

Now imagine we memoize the code:

```
@MEMOIZE
def lcs(s, t):
    if s == "" or t == "": return ""
    if s[0] == t[0]: return s[0] + lcs(s[1:], t[1:])
    a = lcs(s[1:], t)
    b = lcs(s, t[1:])
    return longest of a, b
```

We always recurse on a smaller input, so let's ask how many unique inputs are possible. Well, each input is some *suffix* of s and suffix of t. There are only $O(n)$ suffices of a string of length n, so there are only $O(n^2)$ possible unique inputs. Each such unique call does $O(1)$ work other than recursing, for a total of $O(n^2)$ time. Much better!

4.6.3 Greedy and DP: (fractional) knapsack

Suppose you break into a store selling expensive raw materials. There are n raw materials, numbered $1, 2, \ldots, n$. Each one has a *weight* w_i indicating how many pounds of that material are in the store. Each one also has a *market price* $p_i \geq 0$ indicating the market price per pound of that material.

You brought with you a 'knapsack' (bag) that can carry up to W pounds of materials; you want to grab materials in a way that maximizes the total market price without going over your weight limit W.

Problem 6. *How much of each material should you take?*

Fractional Knapsack

First, let's assume you brought with you a very powerful knife that can cut through bars of raw material. This allows you to take *fractions* of each, i.e., you can take only one pound of material 1 even if there are 10 pounds in the store. This variant of the problem is called *fractional knapsack*.

Represent your knapsack as a function k, where $k(i)$ is the number of pounds of material i you will to take.

Greedy Algorithm

(Think)

- Initial solution: empty set
- How to extend? Take as much of the next-most-expensive material as you can fit.

- Repeat until you fill up your bag.

In fact, this works! We need the following lemma:

Lemma 8. *Suppose some (fractional) knapsack k can be extended to a knapsack having optimal price. Let k' be the knapsack after taking as much of the most expensive remaining material you can. Then, k' can still be extended to an optimal knapsack.*

Proof. (We'll assume for this proof that all the items have distinct prices; it's good practice to rewrite the proof to also work in the case where some items have the same price.)

Let m be the most expensive material and x the amount of it you can take before either running out or filling your knapsack completely.

Let k_* be the optimal knapsack extending k and $\Delta k = k_* - k$ be the amount of each material added to k to form k_*. First, note the optimal knapsack is always filled because prices are positive. Second, note if $(\Delta k)(m) \geq x$ we're done, because then k_* is already an extension of k'.

Otherwise, suppose $(\Delta k)(m) < x$. Then, there must be some material j such that $(\Delta k)(j) > 0$, i.e., that material was added to the knapsack to form k_*. But m was the most expensive material remaining, hence $p_j < p_m$. In other words, we would make *more money for the same weight* by taking more of m rather than $(\Delta k)(j)$ of j, i.e., k_* would not have been optimal.

(End here on whiteboard.)

A bit more algebraically, define k'_* to be identical to k_* except taking more of m instead of j, i.e., such that $k'_*(m) = k_*(m) + \epsilon$ and $k'_*(j) = k_*(j) - \epsilon$ for some ϵ small enough that neither $k'_*(j)$ goes negative nor $k'_*(m)$ goes above w_m. Then, we see that the amount of money we get is

$$\begin{aligned}
\sum_i p_i k'_*(i) &= p_m k'_*(m) + p_j k'_*(j) + \sum_{i \notin \{j,k\}} p_i k'_*(i) \\
&= p_m(k_*(m) + \epsilon) + p_j(k_*(j) - \epsilon) + \sum_{i \notin \{j,k\}} p_i k_*(i) \\
&= \epsilon(p_m - p_j) + \sum_i p_i k_*(i) \\
&\gneq \sum_i p_i k_*(i),
\end{aligned}$$

i.e., the knapsack k'_* is better than the knapsack k_*, a contradiction with the assumption k_* was optimal. □

0–1 Knapsack

Now, let's consider a slightly different scenario: suppose you forgot your knife, so you cannot subdivide each material: you either take all w_i of it, or none of it.

The same greedy algorithm will not work (you'll see why on the practice problem!). But we can still do dynamic programming!

A Recursive Solution If you have no space, return empty sack. Otherwise, either take the first item or don't, and recurse on the remaining items with however much space you have left.

```
def no_knife_knapsack(n_materials, weights, prices, capacity):
    if not weights: return 0

    without = no_knife_knapsack(n_materials - 1, weights[1:], prices[1:], capacity)
    if weights[0] > capacity: return without
    with_ = weights[0] * prices[0]
    with_ += no_knife_knapsack(n_materials - 1, weights[1:], prices[1:],
                               capacity - weights[0])
    return largest of with_, without
```

This is a binary recursion tree that will give roughly 2^n time complexity.

Memoization But there are many repeated states and it always recurses on a smaller problem, so memoize! Let n be the number of materials and c be the initial capacity. Then you have only $O(nc)$ unique inputs.

4.6.4 Post-Lecture Ed Notes

During the lecture, there were two interesting suggestions for greedy algorithms for rod cutting and LCS.

For rod cutting, it was suggested to make the cut of highest price first, and then repeat on the rest of the rod. This does not always produce the optimal cut, and the code above has an example of this.

For LCS, it was suggested to iterate through both arrays in lockstep, stop once you find a character in t that you've already seen in s, then add this character to the LCS and continue. Surprisingly, this does not always find the LCS! A small counterexample is: "xbax" and "ax"; the above algorithm will find the LCS "x" but indeed it should be "ax".

The basic insight behind that greedy algorithm is correct: find the "first" character that both share. But the problem is the definition of "first" here is ambiguous: first in s, or first in t? I think if you find the first shared character in s, and the first shared character in t, and then try both suffixes and take the best one, you should get a correct algorithm (but I haven't thought through it, so take this with a grain of salt!). But I don't think you can get away with not checking both of those branches.

If you're interested in other LCS algorithms, Wikipedia suggests: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=878178`

Name: _____ **Stanford ID Number:** _____

4.6.5 Lecture 8/7 EC Question

Suppose you are in the 0–1 knapsack scenario, i.e., you forgot your knife and cannot subdivide the materials. In lecture we said the greedy algorithm doesn't necessarily work in this setting. Try running the greedy knapsack algorithm on the following items, **assuming an initial capacity of** 2 **pounds.** Did it find the optimal knapsack?

Material	Weight w_i	Price p_i
1	1.5 pounds	\$2.00 per pound
2	1 pound	\$1.90 per pound
3	1 pound	\$1.90 per pound

4.7 Week 7 Problem Sheet

Read This First: These problems, especially for 8/5, are meant to go beyond the scope of this class; unless you are already extremely confident in algorithm design I suggest you spend your time either doing problems from online/the textbooks or from HW6 rather than start on these problems.

4.7.1 Lecture 8/5: Max-Flow, Min-Cut

Problem 60: Suppose the edge capacities are all nonnegative integers and let F be the weight of the maximum flow. Prove in the Ford-Fulkerson algorithm that no matter what choice of path from s to t you make in the residual graph, Ford-Fulkerson stops after at most F iterations and hence takes $O(F(n+m)\log n)$ time.

(Better choices of path in the residual graph lead to much better running time; I'll try to extend this problem to work through those, but if you can't wait, see the Erickson textbook!)

Problem 61: The *max-linear programming (max-LP) problem* takes as input a matrix A, vector b, and another vector c. The goal is to find vector x such that $Ax \leq b$. Furthermore, $c \cdot x$ should be maximal among all vectors satisfying $Ax \leq b$. A number of polynomial-time and "smoothed polynomial-time" (we haven't covered this yet) algorithms are known for LP solving, including the simplex algorithm, Karmarkar's algorithm, and the ellipsoid method.

Prove that you can *reduce* max-flow to max-LP in polynomial time. In other words, describe an algorithm that turns an instance of the max-flow problem into a matrix A and vectors b, c such that the maximum value of $c \cdot x$ is the weight of the max flow and from the maximal x you can extract the max flow. *(Assume you don't know how to solve max-flow directly, i.e., don't just run Edmonds-Karp.)*

LPs are somewhat more flexible than max-flow. For example, show how to use an LP encoding to solve a variant of max-flow where nodes are allowed to 'burn' at most 1 unit of supplies.

Hint: Hint(s) are available for this problem on Canvas.

Problem 62: *(Parts of this problem, especially the proof of strong duality, are adapted from Prof. Friedrich Eisenbrand's linear optimization course notes.)*

The max-flow/min-cut lemma/theorem is a special case of the much larger phenomenon of *duality*. In fact, *every* max-LP problem has some dual min-LP problem. We call the max-LP problem "primal" and the corresponding min-LP problem "dual." Specifically, if the primal problem is to maximize $c \cdot x$ subject to $Ax \leq b$, then the dual problem involves minimizing $b \cdot y$ subject to $A^t y \geq c$. Weak duality says the optimal value for the primal problem is at most the optimal value for the dual; strong duality says they are equal.

1. Prove *weak duality*: if $Ax \leq b$ and $A^t y \leq c$, then $(c \cdot x) \leq (b \cdot y)$.

2. Prove or assume the following fact, paraphrased from Prof. Eisenbrand's lecture notes: Given any compact set $X \subseteq \mathbb{R}^n$ and continuous function $f : X \to \mathbb{R}$, there exist points $x_1, x_2 \in X$ with $f(x_1)$ maximal on X and $f(x_2)$ minimal on X. (Requires some real analysis.)

3. Prove or assume the following fact, paraphrased from Prof. Eisenbrand's lecture notes: Given any closed and convex $K \subseteq \mathbb{R}^n$ and a point $x_0 \in \mathbb{R}^n \setminus K$, there exists some α, β such that $\alpha^t x > \beta$ for all $x \in K$ and $\alpha^t x_0 < \beta$. *(In other words, you can always find a hyperplane separating the point x_0 from any closed, convex set that it does not belong to.)* (Requires some linear algebra and geometry.)

4. Prove or assume the harder part of Farkas's lemma from Prof. Eisenbrand's lecture notes: for any real-valued matrix A and vector b, if there is no solution $x \geq 0$ to $Ax = b$, then there exists some λ such that $\lambda^t A \geq 0$ and $\lambda^t b < 0$.

5. Prove or assume the harder part of what Prof. Eisenbrand calls the "second variant of Farkas's lemma:" if $Ax \leq b$ has no solution then there exists $\lambda \geq 0$ such that $\lambda^t A = 0$ and $\lambda^t b < 0$.

6. Prove the other direction of the second variant of Farkas's lemma: if there exists $\lambda \geq 0$ with $\lambda^t A = 0$ and $\lambda^t b < 0$, then $Ax \leq b$ has no solution.

7. Finally, prove the main part of strong duality: let x_0 be a solution maximizing $c \cdot x$ subject to $Ax \leq b$, and y_0 a solution minimizing $b^t \cdot y$ subject to $A^t y \geq c$. Prove $c \cdot x_0 = b^t \cdot y_0$.

(Pre-hint: there are a few major tricks used. First, if you have two constraints $Ax \leq b$ and $A'x \leq b'$, you can represent both constraints by a single matrix equation $(A/A')x \leq (b/b')$ where A/A' means stack A ond top of A'. Second, if you have constraints $Ax \leq b$ and $A'x' \leq b'$ on two sets of variables, you can represent both by the single equation $(A|A')(x/x') \leq (b|b')$, where $(A|A')$ means stacking A to the left of A'. Third, $Ax \leq b$ has a solution if and only if $Ax - Ay + z = b$ has a solution with $x, y, z \geq 0$. Finally, if x_0 is the optimal value of a max-LP defined by A, b, c, then, for every $\epsilon > 0$, there is no solution to the constraints $Ax \leq b$ and $c^t x \geq c^t x_0 + \epsilon$.)

Hint: Hint(s) are available for this problem on Canvas.

Problem 63: In lecture, we saw that the weight of the max flow was always at most the weight of any min cut. In part (1) you encoded max-flow as an LP. In part (2) you saw the value of the max LP solution is always at most the value of any solution to its dual. Connect these two results, by interpreting the dual of the LP in part (1) as an encoding of the min-cut problem. (This problem is open-ended. You might not be able to perfectly interpret it in this way; try to make as direct a connection as you can. I highly suggest working with a concrete example.)

Problem 64: Continue doing practice problems from the web and the textbooks. Erickson has a whole chapter on applications of max-flow.

Lecture 8/7: More Example Problems

Problem 65: Suppose you have a set of n variables $x_1, \ldots, x_n$ and m constraints, each constraint being of the form $x_i - x_j \leq c$ for two variables x_i, x_j and constant c.

1. Describe how to encode the constraints as a graph such that, if the graph has a negative-weight cycle, then the constraints are definitely *unsatisfiable*, i.e., there is no assignment to the variables making all constraints true.

2. Now, prove if there are no negative-weight cycles in the graph, then there is a satisfying assignment to the constraints.

3. Explain how to modify an algorithm we've seen to determine if the constraints are satisfiable.

4. If the constraints are satisfiable, explain how to use an algorithm we've seen to find a satisfying assignment.

5. Explain how to also support constraints of the form $x_i \geq c$ and $x_i \leq c$.

Hint: Hint(s) are available for this problem on Canvas.

Problem 66: Continue doing practice problems from the web and the textbooks.

4.8 Lecture 8/12: Other Famous Algorithms

4.8.1 "Service Algorithms"

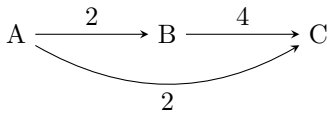
There are an important class of algorithms that are mostly used to reduce other problems to; i.e., it doesn't sound like a very interesting problem to solve, but it turns out you can solve lots of other problems using it as a tool.

Linear Programming

One of the most immediate is *linear programming* (LP):

Problem 7. *Given a matrix A and two vectors b and c, find a vector x such that $Ax \leq b$ and $x \cdot c$ is as large as possible.*

It turns out this is a *very* useful problem to be able to solve. For example, you can use LP algorithms to solve max-flow! Consider the following graph:



You can represent a flow in this graph as a vector $x = [x_1, x_2, x_3]$, where x_1 is the flow $A \to B$, x_2 is the flow $B \to C$, and x_3 is the flow $A \to C$. We know the flow has to be positive, hence $x_1 \geq 0$, $x_2 \geq 0$, and $x_3 \geq 0$. We also know the incoming and outgoing flows of everything other than A, C have to be zero, i.e., $x_1 - x_2 = 0$. We can write all of these constraints in the following matrix inequality:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & -1 & 0 \\ -1 & 1 & 0 \end{bmatrix} x \leq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Any solution to that inequality gives us a flow, and we specifically want the *max flow*, i.e., the one maximizing $x_1 + x_3$, which we can represent as $[1, 0, 1] \cdot [x_1, x_2, x_3]$.

So applying an LP solver to that input will find the max flow for us.

The nice thing about LP solvers is that they're even more general than max-flow: we could, for example, ask an LP solver to give us the maximum flow where the flow from $A \to B$ is exactly twice the flow from $B \to C$!

LP Solving Algorithms and Smoothed Analysis

There are a few major LP solving algorithms:

1. The simplex method: exponential time worst case, but polynomial time *smoothed analysis* (see below).
2. Ellipsoid method: polynomial time worst case, but worse than simplex in practice.
3. Interior points method: polynomial time worst case, competitive-ish with simplex in practice. (Patent-encumbered when first developed as Karmarkar's algorithm.)

The most interesting observation is that the "best in practice algorithm" (simplex) actually has quite bad worst-case behavior. To understand why it performs so well in practice, the *smoothed analysis* method was developed. The motivation for smoothed analysis is somewhat similar to average-case analysis (it can be slow on very unlikely inputs, as long as it's fast for "common" inputs), except in this case we don't really know what a realistic distribution on inputs should be. Smoothed analysis of the simplex method says, *no*

matter what input A, b, c you pick, most of the inputs around it are easy to solve. Symbolically, this looks roughly like:

$$\forall A, b, c, \mathbf{E}_\epsilon[\text{time to solve } (A+\epsilon_A)x \leq (b+\epsilon_b) \text{ maximizing } (c+\epsilon_c)\cdot x] = O(\text{polynomial}).$$

In other words, for every possible input, if you change the input values a tiny bit, it is very likely that that new input is solved very fast. Many LP problems come with some natural noise anyways, since they're adapted from real-world measurements, so this is a fairly clean explanation for the performance of the simplex method.

The max-flow/min-cut duality we saw last week is also generalized into the LP world; see HW7 for a walkthrough of that.

Classes: CS 261 seems to cover LP well, but doesn't seem to cover the simplex method.

MILP

Suppose you have an LP and want to restrict the possible values for some of your variables, e.g., say the flow from A to B has to be either zero or one; a value like 0.5 isn't allowed. The result is a *Mixed-Integer Linear Program*, and in general solving them is NP-hard. There are a variety of techniques for solving these, most notably the Branch-and-Bound and Branch-and-Cut techniques, which essentially try to massage an LP solver's output to meet the constraints of the problem.

Classes: MSE 111/211 seems relevant, EE364B seems to cover branch-and-bound.

Satisfiability and CDCL

If you're not interested in finding the optimal solution, just finding any solution, then LP and MILP solvers are part of a larger class of *satisfiability algorithms*. Satisfiability algorithms solve the following problem:

Problem 8. *Given a* logical formula $P(x_1, \ldots, x_n)$*, find values for* $x_1, \ldots, x_n$ *that makes* $P(x_1, \ldots, x_n)$ *true.*

For example, given a sudoku board, you could make a logical formula that encodes "x_1 is different from everything else in the first row and first column, ...," i.e., all the constraints of the sudoku board.

There are a lot of different algorithms for solving these sorts of problems, going by names like constraint programming, answer set programming, SMT, and CDCL. In general, these algorithms take exponential time on worst-case inputs but are extremely efficient on many real-world instances. If you have a problem that seems to be of this form ("find a solution to these constraints"), you should try using one of them.

Classes: CS257 is a good choice.

4.8.2 "Translate-Then-Operate"

If you have a slow operation that you need to do many times, it can sometimes be easier to first transform the input into a form where that operation becomes more efficient, do the operation as many times as needed, and then transform it back.

A first example for this is *taking the logarithm* if you need to multiply numbers very frequently. Multiplying numbers can be inefficient on some hardware compared to adding them, and can also sometimes have worse precision. So if you need to compute, say, $a \cdot b$, $a \cdot c$, and $a \cdot b \cdot c$, you could first compute the logarithms of a, b, c and then compute $a \cdot b$ as $e^{\log(a)+\log(b)}$. Depending on the cost of computing logarithms, addition, multiplication, and exponentiation, this can sometimes be more efficient and/or more precise.

A similar situation occurs when you need to multiply numbers in a finite field, i.e., compute ab modulo p for some large prime p. Computing the remainder modulo p is very inefficient on most modern computers, but if you first translate a and b into *Montgomery form*, it makes that operation much more efficient. You can then translate back from Montgomery form to the 'normal' int form after you've done your multiplications.

Finally, translating sequences into the frequency domain using a Fourier transform can make computing convolutions more efficient.

4.8.3 Multiplying: Numbers, Matrices, and Polynomials

Lots of people are interested in multiplying things quickly. Those things could be matrices, really large numbers, or really large polynomials.

There are a variety of algorithms for this; in the matrix space, Strassen's algorithm is a divide-and-conquer algorithm that requires only about $O(n^{2.8})$ time (i.e., scalar multiplications) compared to the $O(n^3)$ time of the naïve approach (traffic director). There are (asymptotically) even better algorithms known, and determining the "matrix multiplication exponent" is a very big problem in complexity theory since matrix multiplication is a bottleneck in a lot of other algorithms. Wikipedia says the current best is $O(n^{2.371552})$.

On many computers nowadays multiplying two ints takes $O(1)$ time. But this is possible because ints are limited; on a 64-bit machine an int usually can store at most something like the value 2^{64} before overflowing. If you're instead working with values on the order of $2^{2^{64}}$, say, then you need to start worrying about the time it takes to multiply such numbers. The "grade school" algorithm for this takes $O(n^2)$ time, but there's a divide-and-conquer algorithm (Karatsuba multiplication) that shows you can actually do it in about $O(n^{1.58})$ time (according to Wikipedia). Faster algorithms are known, and this is also an area of active research. Many of them use "Fast Fourier Transforms."

Finally, if you have two polynomials with lots of terms, you might want to find the coefficients of the product polynomial. This again has many faster-than-the-naïve-algorithm solutions, many also using fast fourier transforms.

4.8.4 String Search

Given strings s and t, find out whether s occurs as a substring of t. The naïve algorithm for this takes $O(|s||t|)$ time, but you can do it in $O(|s| + |t|)$ time using an algorithm like Knuth-Morris-Pratt (KMP).

4.8.5 Stable Marriage

Wikipedia describes it like so:

> Given n men and n women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. When there are no such pairs of people, the set of marriages is deemed stable.

There are some cute solutions to this problem, most notably one called Gale-Shapley.

4.8.6 Guaranteed Near-Optimality of Greedy-Like Algorithms

There are some cases where, even though the greedy algorithm does not necessarily find the optimal solution, you can prove that it finds something *close to* the optimal solution. Things to search in this vein include "greedy optimization of submodular functions" and "Markov-Chain Monte Carlo" (not exactly a greedy algorithm, but similar).

4.8.7 Other Searching and Sorting

Many of the graph algorithms we saw can be sped up by using a *Fibonacci heap* instead of a 'normal' heap. Fibonacci heaps let you do everything *except for* delete-root in $O(1)$ amortized time; delete-root is still $O(\log n)$. So they wouldn't speed up heapsort, but they speed up things like Dijkstra's algorithm in some scenarios.

If you leave the realm of comparison-based data structures, and assume, e.g., that your values are integers in some range, then you can do some sorting and searching operations more efficiently. In the sorting world, algorithms like counting sort can sometimes give better time for sorting, and data structures like van Emde Boas trees can do better than AVL trees, if the values in question come from a small range of integers.

Finally, when we studied graph algorithms we assumed you were given a single, static graph. But in real life, these graphs change over time: roads get built, flights get added, and friendships get made. There's a large research area dealing with *dynamic graph algorithms*, which support those scenarios more efficiently.

4.8.8 Post-Lecture Ed Notes

There was a question asking whether the decision tree for a sorting algorithm can ever have more than $n!$ leaves, and I think the answer is basically no. If you restrict the tree to just the nodes that can actually be reached on some execution for some input permutation of $\{1, \ldots, n\}$, I think I agree with the questioner that you must have exactly $n!$ leaves. (If you build the tree for more possible inputs, e.g., with repeats like $[1, 1, 3]$, then there may be more than $n!$ leaves. But we didn't do that, so I think saying exactly $n!$ leaves is correct.) In any case, all we needed for the proof was that there are at least $n!$ leaves, but still great to think through these things !

4.9 Lecture 8/14: Review Day

4.9.1 Announcements

1. Cheat sheet for the final: one 8.5x11 inch page, both sides, handwritten.
2. Exam starts at 8:30 AM on Saturday for in-person students. Lasts 3 hours.
3. SCPD students have until 8:30 AM Sunday to start the test.
4. Policy for the final exam: must show up within 30 minutes of the start time (i.e., by 9 AM).

4.9.2 Algorithm Analysis Matrix

Might help if we do it in this order:

- If your algorithm makes internal decisions (such as the pivot, or hash function, etc.), how are those decisions made?
  - The *best* possible decision?
  - The *worst* possible decision?
  - A *randomly chosen* decision? (In this class, we're calling an "expected time" analysis.)
- If your algorithm takes an input, which input are you computing the running time for?
  - The *best* possible input?
  - The *worst* possible input?
  - A *randomly chosen* input? (In this class, we're calling an "average case" analysis.)
- If your algorithm can support multiple operations in a row, are you interested in:
  - The time for a single operation in isolation?
  - The time for a sequence of operations? ("Amortized analysis.")
- If your algorithm makes internal decisions (pivot, hash function, etc.) is the input:
  - Fixed ahead of time? (We've been calling this "black-box analysis.")
  - Change in reaction to those internal decisions? (We've been calling this "white-box analysis." There are different levels of information you might allow the user to get about the internal decisions, e.g., do they know the hash function entirely? Or just what items got inserted to which bucket?)

This is a pretty big matrix of options, but for the most part, it's just saying that *the analysis question you need to answer depends on the scenario you're analyzing.* The terms above are just words that help us quickly refer to certain types of scenarios, but, e.g., on the exams, I think we've been pretty consistent about usually describing the exact scenario when it's relevant.

4.9.3 First Quiz Topics

Big-O

(From the 6/26 lecture:)

Definition 23. *For any natural-valued function $f(n)$ we can define the set of functions $O(f(n))$ like so:*

$$O(f(n)) = \{f_0(n) \mid \exists C_{f_0}, N_{f_0}, \forall n \geq N_{f_0}, \quad 0 \leq f_0(n) \leq C_{f_0} f(n)\}.$$

And this convention:

Convention 2. *When we have an equation with asymptotic notation on either (or both) sides of the equals sign, we mean the following: if every asymptotic set on the left hand side is replaced by* any *of its members, then there* exists *a member of each asymptotic set on the right-hand side that can replace it and make the equation true.*

Hence, for example,

$$O(f(n)) + O(g(n)) + h(n) = O(k(n))$$

means, for any *choice of* $f_0(n) \in O(f(n))$ *and for* any *choice of* $g_0(n) \in O(g(n))$ *there* exists *a choice of* $k_0(n) \in k(n)$ *such that*

$$f_0(n) + g_0(n) + h(n) = k_0(n).$$

Let's work through Homework 1.1.1. It asks to prove carefully that $O(f(n)) + O(g(n)) = O(f(n) + g(n))$.

Proof. By the convention in class, we must show that for any $f_0(n) \in O(f(n))$ and $g_0(n) \in O(g(n))$, there exists some $k_0(n) \in O(f(n)+g(n))$ such that $f_0(n)+g_0(n) = k_0(n)$. In particular, take $k_0(n) \coloneqq f_0(n)+g_0(n)$, i.e., it suffices to show $f_0(n) + g_0(n) \in O(f(n) + g(n))$.

By definition of big-O, there exist C_f, C_g, N_f, N_g such that $0 \leq f_0(n) \leq C_f f(n)$ for $n > N_f$ and $0 \leq g_0(n) \leq C_g g(n)$ for $n > N_g$. Now let $N_{f+g} = \max(N_f, N_g)$ and $C_{f+g} = \max(C_f, C_g)$. Adding the two earlier inequalities gives us

$$0 \leq f_0(n) + g_0(n) \leq C_f f(n) + C_g g(n) \leq C_{f+g} f(n) + C_{f+g} g(n) \leq C_{f+g}(f(n) + g(n))$$

for all $n > N_{f+g}$, hence $f_0(n) + g_0(n) \in O(f(n) + g(n))$ as desired. □

Sorting algorithms

- Selection sort: simple, but $\Theta(n^2)$ best and worst case.
- Insertion sort: much better best case ($\Theta(n)$) but worst case and average case both still bad ($\Theta(n^2)$).
- Merge sort: good best- and worst-case ($\Theta(n \log n)$) but is requires $O(n)$ extra space.
- Quick sort: good *expected* time on all inputs ($\Theta(n \log n)$) but if you get really unlucky with pivot choices it can be as bad as $\Theta(n^2)$; also still $O(\log n)$ space usage.
- Heap sort: only $O(1)$ extra space, and guarantees worst-case $\Theta(n \log n)$, fully deterministic! But not quite as efficient in practice (bad cache behavior and doesn't really take advantage of any existing sortedness in the array).

Maybe useful to look at insertion sort best and worst case.

Sorting lower bound

At the end of the sorting section, we proved that every comparison-based sorting algorithm takes *at least* $\Omega(n \log n)$ time. A "comparison-based sorting algorithm" is one that only interacts with its input by comparing two elements in the input array or moving them around. It can't, for example, add them.

The proof of the comparison-based sorting lower bound went like so:

1. For any input size n, we can turn any comparison-based sorting algorithm into a *decision tree*, where the interior nodes represent the comparisons made by the algorithm, the leaves represent the permutation of the input array that puts it in sorted order (i.e., the return value of the sorting algorithm), and edges represent what the algorithm does next depending on the outcome of a comparison.

   In particular, each input value corresponds to a single branch in the tree (from root to leaf), corresponding to the sequence of comparisons the algorithm makes on that input.

   This decision tree captures all the possible things the algorithm could do on all the possible inputs of size n.

   (Decision trees are hard to draw by hand because they're so large, but we posted a Python script that can draw them for you!)

2. Because the sorting algorithm can have $n!$ many different outputs, i.e., permutations that put it back in sorted order, there must be at least $n!$ many leaves in the tree.

3. But you can prove for *any* binary tree, if there are at least k nodes, then the depth must be at least $\Omega(\log k)$.

4. Hence, there must be a branch in the tree of length at least $\Omega(\log n!)$.

5. But each branch corresponds to the sequence of comparisons made when running the algorithm on some input, hence there must be some input that causes the algorithm to make at least $\Omega(\log n!)$ comparisons.

6. Each comparison takes at least $\Omega(1)$ time, hence the algorithm must take at least $\Omega(\log n!) = \Omega(n \log n)$ time on at least one input of size n.

Decision trees vs. Recursion Trees The decision trees described are very different from recursion trees. In a recursion tree, each node represents one recursive call to the function. In a decision tree, each (interior) node represents a comparison performed by the function.

4.9.4 Second Quiz Topics

In this section we started to look at data structures that could store sets of items.

Searching Algorithms

- BSTs: good average case time for inserting n items in random order, but bad worst-case time.

- AVL: $O(\log n)$ worst-case everything, but deletion was not easy (we didn't learn it) and needs at least two extra bits per node to store balance factor.

- 2–3 trees: $O(\log n)$ worst-case everything; if represented as a red-black tree needs only one bit extra per node. Can be extended to B-trees, which are most performant in practice. Deletion was slightly easier.

- Splay trees: $O(\log n)$ amortized. Very easy to implement many operations (split, join, delete, insert, ...). Performs even better if some nodes are more popular than others (but we didn't analyze this case closely).

- Hash maps: We saw both linear probing and separate chaining. We'll talk about time analysis later; the main thing to note is that if you choose the sequence of operations (what to insert/lookup) without knowing anything at all about the hash function, and then you pick the hash function at random, the expected amortized time per operation is $O(1)$.

- Union-find: $O(\alpha(n))$ amortized (we didn't talk much about what $\alpha(n)$ is, except that it's very close to 1). Relied on *path compression* and *join-by-weight*.

The first four are essentially comparison-based. Hash maps don't require comparisons, but they do require a hash function. Hash maps are generally faster (at least in terms of expected time), but they are somewhat less flexible (e.g., can't easily find "largest thing in the set less than x"). Union-find solves a different problem than the others; it keeps track of many different sets of items under the operations 'union' and 'same-set.'

Hash table collisions

(We didn't talk much in lecture about linear probing, so going to only talk about separate chaining here. Also going to assume a load factor ≤ 1.)

In retrospect, I think it's useful to separate out exactly what expected values we computed vs. in what scenarios they become useful. Let's say an "operation" is either an insertion or search for a specific value (deletions can be handled too, but we didn't spend too much time talking about that). Then, in lecture we saw the following:

- If you decide on a sequence of n operations, then pick a hash function at random, and then run the operations: the expected number of items in any given bucket is $O(1)$.
- If you decide on a sequence of n operations, then pick the hash function at random, and then run the operations: the expected number of items in **the most-filled bucket** is *not* $O(1)$.
- If you pick the hash function and *then* decide on the sequence of n operations knowing that hash function, then it's possible for the person who picks the sequence to ensure that some bucket has $\Theta(n)$ items in it.

Now, based on those facts, you can analyze a lot of different scenarios.

The main thing to consider is *whether the input is determined independent of the choice of hash function, or whether it can depend on the choice of hash function.* We've been calling the first case "black-box" and the second case "white-box" (though maybe we should think of it as a spectrum of "gray-box" scenarios depending on how much information about the hash function the user gets).

As a running example, let's consider a spellcheck program that takes a dictionary of n words, inserts them all into a hash table, and then checks each word in the user's n-word input to make sure it's in the dictionary.

- If the user has to provide their input before the program runs, e.g., they run the program like `spellcheck.exe dictionary.txt my_book.txt`, then the randomly chosen hash function will map every distinct word to each bucket with equal probability, and so each operation will take expected (amortized) time $O(1)$, for a total expected time of $O(n + n) = O(n)$.
- Suppose, however, the program is more interactive: after loading the dictionary, it stays open and lets the user specify words one-by-one. Also assume the user can reliably time how long each dictionary search takes. Then they could do the following:
  1. First, search for each one of the n dictionary words and find the word x that takes the longest to search for: this must be the one at the end of the longest chain.
  2. In particular, the expected lookup time for x will *not* be $O(1)$.
  3. Then, do n repeated searches for x.
  4. Notably, those n repeated searches will be *worse than* $O(n)$, so the total time for these $3n$ operations is *not* $O(n)$. (But it is still pretty good, but we haven't done enough analysis to give an upper bound on this.)
- Alternatively, suppose the program provides a debug mode: you can ask it to print out all the buckets before specifying what words you want to search for in the dictionary. Then the user could do the following:
  1. Find the item x at the end of the longest chain.
  2. Perform n repeated searches for x.
  3. Notably, those n repeated searches will each be *worse than* $O(1)$, so the total time for these $2n$ operations is *not* $O(n)$. (But it is still pretty good, but we haven't done enough analysis to give an upper bound on this.)

- Finally, suppose the user modifies the program's source code so it doesn't use a random hash function, but rather always uses some fixed hash function $h(x)$. Also suppose the user controls the dictionary. Then the user could do the following:

  1. Make up fake words for the dictionary so that they all have the same hash value.
  2. Now there's a chain of size n; search for the last word in that chain n times.
  3. This takes $O(n^2)$ time!

Amortized Analysis

Many data structures have the following property: some individual operations can be slow, but any *sequence of many operations* is still fast.

This led to the following definition:

Definition 24. *An algorithm has worst-case running time $O(A(n))$ amortized over n operations if every sequence of n operations takes time $O(nA(n))$.*

(We may sometimes drop the "amortized over" statement and just say something like, it has amortized time $O(A(n))$. We generally assume the sequence of n operations starts with an empty initial state.)

The best example of this was inserting into a growing array, where whenever the array fills up we make it twice as large as it needs to be.

Most of the insertions have free space and hence take just $O(1)$ time to add the item at the end. But every now and then, an insertion will fill up the array, triggering a slow, $O(n)$-time resize. But these resizes get less and less frequent the more you insert, and in fact, we showed in that lecture that any sequence of n insertions takes at most $O(n)$ time total, i.e., $O(1)$ amortized time per insertion.

We saw two different ways to go about proving the $O(n)$ time total. The first was to directly reason about a sequence of n insertions and how long it would take. The second was the *potential method*, where we defined a potential function and proved two key facts about it:

1. The potential of any state in the sequence is never less than that of the initial potential.
2. For any operation in the sequence, the time it takes plus the change in potential is always at most $Cf(n)$, where C is a constant and $f(n)$ is the amortized time bound we're trying to prove (1 in this case).

As long as those two conditions were met, we saw that implies any sequence of n operations takes $O(nf(n))$ time and so the amortized time per operation is $O(f(n))$.

4.9.5 Graphs Section Topics

Min-Cut Part of Max-Flow/Min-Cut

Let's revisit the scenario of the US trying to disrupt USSR supply routes, and this time let's suppose their goal is the following: *blow up the smallest number of supply routes such that no goods can get from s to t.*

What they should do is the following:

1. Label all supply lines with a capacity of 1 and use Ford-Fulkerson to find a minimal s–t cut C. (To do this, find the max-flow and then take C to be everything reachable from s in the final residual graph — we proved in lecture this was a min cut.)
2. Then, blow up every edge exiting the cut C.

Because C is an s–t cut, to get any goods from s to t they would have to at some point exit the cut C, hence blowing up all the exiting edges totally stops goods from flowing from s to t.

Because the cut is minimal, this blows up the fewest number of edges possible to achieve that goal. Here's a slightly more rigorous proof of that fact:

Proof. Let C be a min s–t cut and E be the set of exiting edges from C. Suppose for sake of contradiction that there is some other set of edges E' where (i) $|E'| < |E|$ and (ii) after removing the edges in E' there is no remaining path from s to t. Then, consider the cut

$$C' = \{n \mid n \text{ is reachable from } s \text{ after removing edges in } E'\}.$$

This is an s–t cut by assumptions on E', and in particular, every exiting edge of C' must be in E' (if there were an exiting edge $a \to b$ not in E', then b would be reachable from s and hence $b \in C'$ and hence $a \to b$ would not be an exiting edge — contradiction). But this implies $|E'|$ is larger than the crossing capacity of C'. But then C' has lower crossing capacity than C, i.e., C was not a minimal cut. This is a contradiction that completes the proof. □

4.9.6 Post-Lecture Ed Notes

If you're interested in KMP string matching, I think SW seems to do the best job explaining it. There's also a nice intuitive introduction here: `https://gist.github.com/LeoRiether/c61b7f709b7826f47bc67f0c5d0d9b6b` but it's incomplete

There were some interesting questions about alternative settings for stable marriage; folks with those questions may find these related problems on Wikipedia interesting: `https://en.wikipedia.org/wiki/Stable_marriage_problem#Related_problems`

Here's more information about the real story behind the Simplex algorithm: `https://www.lancaster.ac.uk/stor-i-student-sites/ben-lowery/2022/03/linear-programming-and-the-birth-of-the-simplex-algorith`

4.10 Study Guide for Third Quiz (Graphs and Algorithm Design)

While we don't expect any changes, this document is only a best-effort estimate of the topics. Everything from lecture is in-scope.

Topics

The following topics are in-scope:

1. Know the graph terminology from lectures (like sparse, dense, directed, undirected, weighted, unweighted)
2. Know what the following algorithms do, how they work, how to apply them in different scenarios, and how much time they take: explore (BFS and DFS), toposort, Dijkstra, Bellman-Ford, Floyd-Warshall (don't need to know how it works unless we cover it on Wednesday), Prim, Kruskal, activity selection, Edmonds-Karp/Ford-Fulkerson (max-flow/min-cut; don't need to know running time).
3. Be able to solve problems using dynamic programming, and analyze the running time of your solution.
4. Be able to solve problems using greedy algorithms, and analyze the running time of your solution.

Having done homework questions (especially the textbook or online ones) might be useful in answering quiz questions, but we will write the quiz assuming you haven't done the homework.

Appendix A

Appendix: About the Logarithm

Originally, I expected we wouldn't need any nontrivial limits or facts about the logarithm/exponential function in this class. Over time, a few facts (especially the limit of $\left(\frac{n-1}{n}\right)^n$ and the asymptotics of the harmonic sequence) kept coming up again and again. At the time, I gave students the limits as something they were allowed to use without proof, but if I were to do it again, I would provide the following handout to explain the logarithm and the exponential functions.

Please note that doing this rigorously requires a level of calculus/analysis that is beyond the prerequisites for this course. In the below, I've tried to indicate where there are gaps, e.g., by using $\approx$ instead of writing out a limit.

(Warning: this content has not been thoroughly checked, and is probably even more bug-ridden than the rest of these notes!)

A.1 A Natural Definition of the Log

Three related objects appear with surprising frequency across mathematics and computer science. These are the exponential function $\exp(x)$, the constant e, and the natural logarithm $\log(x)$. All of these can be defined in terms of each other:

1. If you know the function $\exp(x)$, you can define $e \coloneqq \exp(1)$ and $\log(x) \coloneqq \exp^{-1}(x)$.
2. If you know what the value of the constant e is, you may define $\exp(x) \coloneqq e^x$ and $\log(x) \coloneqq \exp^{-1}(x)$.
3. If you know the function $\log(x)$, you can define $\exp(x) \coloneqq \log^{-1}(x)$ and $e \coloneqq \exp(1)$.

To be parsimonious, we would like to pick one of these three to serve as the base definition for all others. I know of no way to define $\exp(x)$ directly that does not require an understanding of limits and/or derivatives. Unfortunately, at the undergraduate level, too few students have taken a course in differential calculus for us to assume such knowledge in class. The second option is also not optimal, for two reasons: (1) even if you know e, defining e^x for nonrational x requires an understanding of limits, and (2) defining the value of e itself is nontrivial.

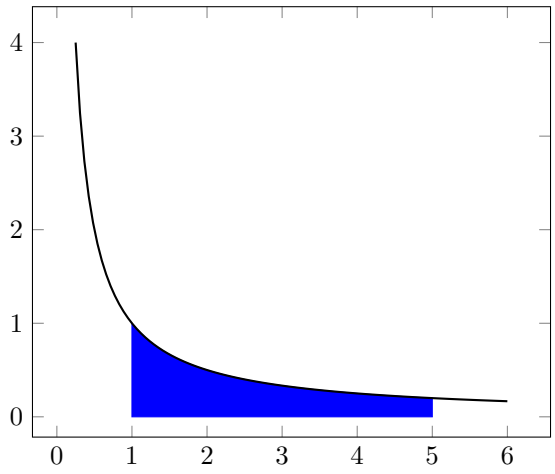
For these reasons, we will take the third road: define $\log(x)$ directly, then take $\exp(x)$ to be its inverse and e to be $\exp(1)$. This route allows us to trade differential calculus for integral calculus, i.e., trade talking about rates of change for talking about areas. I personally find areas to be less intimidating, and essentially let us do most of the calculus we need to do in a convincing (if not entirely rigorous) geometric way. It also has the nice benefit of making the $\log(n) \approx H_n$ connection immediately and visually intuitive. This is particularly helpful in computer science where H_n appears with high frequency.

A.2 A Definition of $\log(x)$

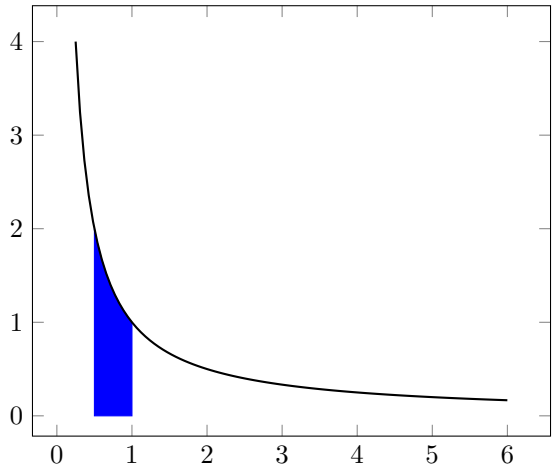
Without further ado ...

Definition 25. *The* natural logarithm $\log(x)$ *is defined to be the area under the curve* $\frac{1}{x}$ *in the range* $[1, x]$. (If $x < 1$, we take it to be -1 times the area in the range $[x, 1]$. The domain of the function is $(0, \infty)$.)

Hence, $\log(5)$ is defined to be the following area:



And $\log(0.5)$ is defined to be -1 times the following area:



A.3 Warmup: A Generalized Logarithm

In our definition of the logarithm, we rather arbitrarily decided to start at $x = 1$. What if we wanted the area under the curve $\frac{1}{x}$ in some range $[a, b]$? We could define the *range-logarithm* like so:

Definition 26. *The* range-logarithm $\log[a, b]$ *is the area under the curve* $\frac{1}{x}$ *from* $x = a$ *to* $x = b$.

In particular, with this notation, $\log(x) = \log[1, x]$.

Surprisingly, if you can compute the normal logarithm and you can divide, it's easy to compute the range logarithm!

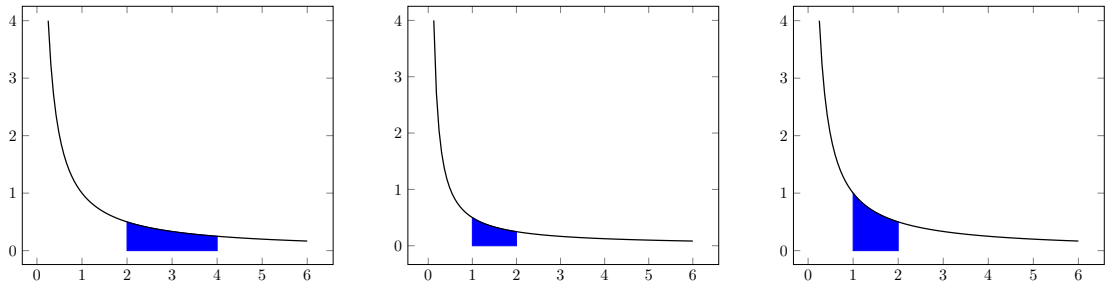
Theorem 32. *For any positive numbers* $a \leq b$, *we have* $\log[a, b] = \log(b/a)$.

To prove this theorem we'll use some geometric manipulations and see what happens to the area. We'll need this pretty intuitive axiom about area:

Axiom 1. *If some region has area* A *and you stretch it either vertically or horizontally by a stretch factor* c, *then the resulting region has area* cA.[1]

[1] This axiom can also be used to show that the logarithm is continuous!

Now, let's study the following three diagrams, with $[a, b] = [2, 4]$.



1. The first diagram shows $\log[a, b]$, i.e., the area under $\frac{1}{x}$ from $x = a$ up to $x = b$.

2. The second diagram shows what happens when we scale the graph down *horizontally* by a factor of a, i.e., the area under the curve $\frac{1}{ax}$ from $x = a/a$ up to $x = b/a$. If we call this area A, Axiom 1 tells us $A = \frac{1}{a}\log[a, b]$.

3. The final diagrams shows what happens when we scale the graph up *vertically* by a factor of a, i.e., the area under the curve $\frac{a}{ax}$ from $x = a/a$ up to $x = b/a$. If we call this area B, Axiom 1 tells us $B = aA$.

   (a) But, in fact, $\frac{a}{ax} = \frac{1}{x}$ and $a/a = 1$, so $B = \log(b/a)$!

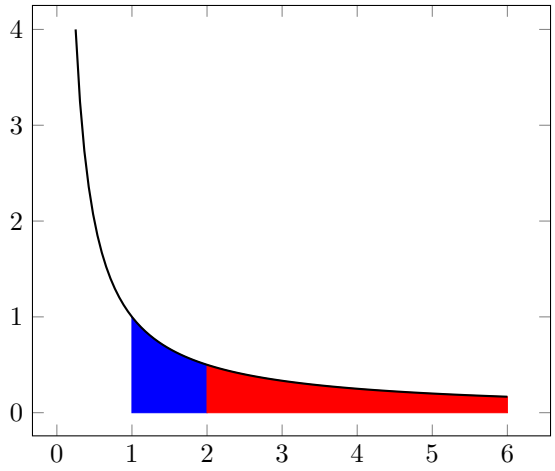
   (b) Furthermore, $B = aA = a\frac{1}{a}\log[a, b]$, so $\log(b/a) = B = \log[a, b]$, exactly as claimed.

A.4 A Fundamental Property of the Logarithm

Now let's look at one of the most exciting and useful properties of the logarithm: it turns multiplication into addition! More precisely, we'll prove:

Theorem 33. *For any real numbers a and b, we have* $\log(ab) = \log(a) + \log(b)$.

Let's draw the scenario with $a = 2$ and $b = 3$:



This drawing makes clear how we can split up $\log(ab)$ in terms of $\log[1, a]$ and $\log[a, ab]$; in particular,

$$\log(ab) = \log[1, a] + \log[a, ab] = \log(a) + \log(ab/a) = \log(a) + \log(b),$$

where the second equality follows from the warmup lemma we just proved!

A.4.1 Corollaries of this Fundamental Property

In fact, we can now see the following corollaries:

1. $0 = \log(1) = \log(b \cdot 1/b) = \log(b) + \log(1/b)$, hence $\log(1/b) = -\log(b)$.

2. $\log(a/b) = \log(a \cdot \frac{1}{b}) = \log(a) + \log(1/b) = \log(a) - \log(b)$.

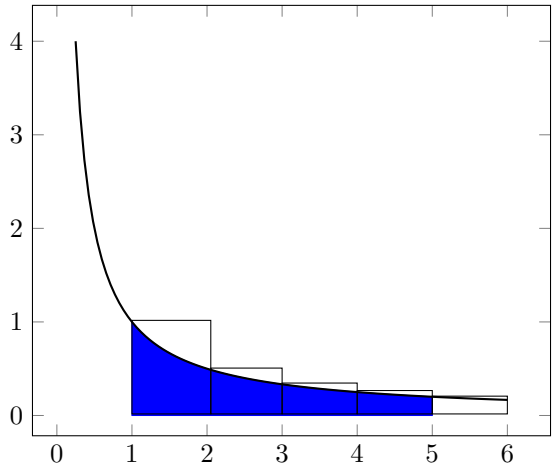
3. $\log(a^k) = \log(aa \cdots a) = \log(a) + \log(a) + \cdots + \log(a) = k \log(a)$ for integers k. (Exercise for reader: explain why this fact, along with $\log(ab) = \log(a) + \log(b)$, together imply $\log(a^k) = k \log(a)$ holds for all rational k. Then use the fact that area scales continuously with stretching to prove $\log(a^x) = x \log(a)$ for all real x.)

4. If $x < y$, then the area from 1 to y strictly contains the area from 1 to x. Hence log is strictly increasing.

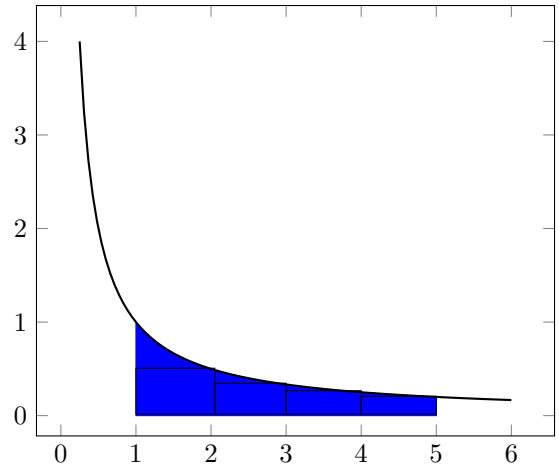
5. log is a bijection between $(0, \infty)$ and $\mathbb{R}$: it is injective because it is strictly increasing, and since $\log(2^x) = x \log 2$ for all x it is also surjective (onto).

A.5 The Harmonics

The following sequence shows up frequently in computer science: $H_n = \sum_{i=1}^{n} \frac{1}{i}$. Surprisingly, it's possible to prove $H_n = \Theta(\log n)$! The reason is that $\sum_{i=1}^{n} \frac{1}{i}$ is essentially a good, discrete approximation of the area under $\frac{1}{x}$. In particular, H_n is an *overapproximation* of $\log n$; the following diagram shows the five terms of $H_5 = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}$ shown as rectangles with the corresponding area drawn on top of the area defining $\log 5$.



From this diagram, we know immediately that H_n is larger than $\log n$. But *just how much* larger is it? It turns out: not more than 1 larger! To see why, make the same plot with two changes: drop the first box (i.e., plot $H_n - 1$) and shift every rectangle over by one spot.



This then shows that $H_n - 1$ is *smaller* than $\log n$; together we've shown

$$H_n - 1 \leq \log n \leq H_n,$$

which one can show implies $H_n = \Theta(\log n)$.

(1 here is clearly an overapproximation of $H_n - \log n$; the true value — in the limit, at least — is something called Euler's constant. Trying to better approximate this is a fun exercise — you might start by lower bounding the size of the white region in each box in the overapproximation diagram.)

A.6 Approximating $\log x$

In the previous section, we saw that H_n could be interpreted as an approximation of $\log n$ using rectangles of width 1. We can make a similar approximation with any width w; in particular the analogous bounds are

$$\sum_{i=1+w,1+2w,\ldots,1+kw} \frac{w}{i} \leq \log x \leq \sum_{i=1,1+w,1+2w,\ldots,1+kw} \frac{w}{i},$$

where k is the largest integer with $1 + kw \leq x$. This produces an approximation with error bounded by w, hence it allows you to compute $\log x$ to any desired precision.

A.7 What About $\exp(x)$?

Recall that we showed the logarithm was a bijection: that means we can define its inverse, which we'll call the *exponential function*.

Definition 27. $\exp(x)$ *is the inverse of the logarithm, i.e.,* $\log(\exp(x)) = \exp(\log(x)) = x$.

Let's investigate some properties of the exponential. Remember that logarithms turned multiplication into addition; exponents will do the opposite!

Theorem 34. $\exp(a)\exp(b) = \exp(a+b)$ *and* $\exp(a/b) = \exp(a-b)$.

Proof. The logarithm is a bijection, so it suffices to show $\log(\exp(a)\exp(b)) = \log(\exp(a+b))$. Indeed, this holds iff $\log(\exp(a)) + \log(\exp(b)) = \log(\exp(a+b))$, which holds iff $a + b = a + b$, which is true. Similar algebra expresses truthity of the other claim. □

In particular, then, we can characterize $\exp(n)$ as repeated multiplication:

Theorem 35. *If* n *is a natural number, then* $\exp(n) = \exp(1)^n$ *and* $\exp(-n) = \exp(1)^{-n}$.

Proof. Write $\exp(n) = \exp(1 + 1 + \cdots + 1)$ and apply previous theorem n times. Similar for $\exp(-n)$. □

In fact, this establishes that $\exp(r) = \exp(1)^r$ for any *rational* number r. The logarithm is continuous, so the exponent is as well, so we can see the exponential function is exactly $\exp(x) = \exp(1)^x$ for all real numbers x.

A.8 What About e?

The last theorem indicates there may be something special about $\exp(1)$, and indeed we give it a special name: $e \coloneqq \exp(1)$. Phrased in terms of logarithms, e is the unique number such that $\log(e) = 1$. Since $\log x$ is increasing, you can use the strategy in "Approximating $\log x$" above to compute e to any desired precision. When you do this, you'll find that the value of e is approximately 2.7.

A.9 Some Interesting Limits

The following limit shows up frequently:

Theorem 36. *For very large n, the quantity $\left(\frac{n-1}{n}\right)^n$ approaches $\exp(-1) = e^{-1}$.*

Proof. Applying logs to both sides, it suffices to show $n\log((n-1)/n)$ approaches -1. But $n\log((n-1)/n) = n(\log(n-1) - \log(n))$, hence this is asking us to determine the difference between $\log(n)$ and $\log(n-1)$. The magnitude of that difference is exactly $\log[n-1, n]$, i.e., the area under the curve $1/x$ in the range $[n-1, n]$. We can underapproximate it using a rectangle of length 1 and height $1/n$, so the magnitude is *at least* $1/n$. We can overapproximate it using a rectangle of length 1 and height $1/(n-1)$, so the magnitude is *at most* $1/(n-1)$. In total then, we have

$$-n/(n-1) \leq n\log((n-1)/n) \leq -n/n,$$

i.e.,

$$n\log((n-1)/n) \approx -1$$

for very large n, as desired. □

In fact, an analogous proof establishes the same fact for $\left(\frac{n+1}{n}\right)^n$; it approaches e.

Corollary 1. *For very large n and arbitrary constant c, the quantity $\left(\frac{n-c}{n}\right)^n$ approaches $\exp(-c) = e^{-c}$.*

Proof. (Thanks Anjiang!) Define $x = n/c$, then the above is $\left(\left(\frac{x-1}{x}\right)^x\right)^c$. But by the prior theorem the interior approaches e^{-1}, hence the quantity approaches e^{-c} as desired. □

A.10 Change of Base and $\log_2$

It's also very common in computer science to work with the *binary logarithm*, $\log_2$, defined as the inverse of 2^x. In other words, $2^{\log_2 x} = x$. When working under a big-O (or other asymptotic notation) we usually treat $\log_2$ and log identically. That is because they are simply a constant multiple from each other! More generally, define $\log_b$ to be the logarithm base b, i.e., the unique function such that $b^{\log_b x} = x$. Then we have the following theorem:

Theorem 37. *For any b and x,*

$$\log_b x = \frac{\log x}{\log b}$$

Proof. It suffices to prove $b^{\log x/\log b} = x$. In fact,

$$b^{\log x/\log b} = (e^{\log b})^{\log x/\log b} = e^{\log b(\log x/\log b)} = e^{\log x} = x.$$

□

A.11 Taylor's Version

It's popular to use the following equality: $\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$. Wikipedia, citing Rudin, makes the following argument for this fact. Recall we know, for large n,

$$\exp(x) \approx \left(\frac{n+x}{n}\right)^n.$$

Applying the binomial theorem to the right-hand side, we see (for large n)

$$\begin{aligned}
\exp(x) &\approx \sum_{i=0}^{n} \binom{n}{i} \frac{x^i}{n^i} \\
&= \sum_{i=0}^{n} \frac{n(n-1)\cdots(n-(i-1))x^i}{i!n^i} \\
&= \sum_{i=0}^{n} \frac{x^i}{i!} \left(\frac{n-1}{n}\frac{n-2}{n}\cdots\frac{n-(i-1)}{n}\right)
\end{aligned}$$

As n gets large, the coefficient $\frac{n-1}{n}\frac{n-2}{n}\cdots\frac{n-(i-1)}{n}$ on the ith term approaches one, hence

$$\exp(x) \approx \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

as desired. (This last step is a bit too handwavey; refer to Wikipedia/Rudin for a more rigorous analysis if you're interested.)