Survivor's Manual for CIS 1210

Jeffrey Saito (Hitman7128)

Last Update: August 26, 2024

This is a course about Algorithms and Data Structures using the JAVA programming language. We introduce the basic concepts about complexity of an algorithm and methods on how to compute the running time of algorithms. Then, we describe data structures like stacks, queues, maps, trees, and graphs, and we construct efficient algorithms based on these representations. The course builds upon existing implementations of basic data structures in JAVA and extends them for the structures like trees, studying the performance of operations on such structures, and their efficiency when used in real-world applications. A large project introducing students to the challenges of software engineering concludes the course.

— *Description of CIS 1210 in course catalog*

Contents

§1 Introduction

§1.1 How to Use This Guide

Like the prerequisites state, you must have taken both CIS 1200 and CIS 1600 before taking this course. This course combines the programming aspect of CIS 1200 with the conceptual knowledge of CIS 1600. Most of this guide will go over the conceptual aspect and break it down (especially the math heavy parts) in Layman's terms and a relatable, informal tone.

For each section of content, I will have the long version and then the summary "TL;DR" version. The long version is meant for when you want to inform yourself of the material ahead of time, or the content is new for you and things don't fully make sense yet. The summary version is meant for review or to get the core ideas in the midst of the sea of content.

Be on the look out for these boxes:

Warning 1.1. Warning boxes usually will contain an important caveat regarding the concept, like an oversight you can make if you're not careful. Pay attention to these when I use it.

Review 1.2. Review boxes are for when I think it would be wise to review a concept if you forgot it. Concepts very often build off one another in this course with some from CIS 1600 and CIS 1200.

Advice 1.3 — Advice boxes offer additional clues on how to use a particular concept, especially for homework problems or on an exam.

Lemma 1.4

Lemmas or theorems are generally claims we have to prove in order to make progress and help us better understand why a concept may work.

Example 1.5

An example box will have a problem (generally one from class or recitation) that will illustrate the concept in action by working to solve the problem.

§1.2 General Course Advice

- First thing's first, **this course is challenging**. Don't come in with hubris and thinking you'll never need help. The TAs are there to help you.
- Whether you like my (or the professor's) thorough explanations or not, all the intuition you can get for the material matters. If all you want is the summary, that would be like skipping all the lectures and only attending recitation.
- Sometimes the proofs taught in class will make you think "How is anyone supposed to derive that completely from scratch?" Focus less on that and more on understanding why it works. Also, try to extract the intuition from said proof so it can be put to use on other problems.
- There are two types of homeworks: written and programming. Written homeworks are a problem set, similar to that of CIS 1600, while programming homeworks are similar to those in CIS 1200. Sometimes, the homework for the week will be half written and half programming.
- This goes without saying, DO NOT procrastinate on the homeworks. If you don't know where to start, at the very least, write down material that you have recently learned that seems relevant or could lead to a solution. Then, see where that gets you and go to office hours if you're still stuck.

- For written homeworks where they ask you to design an algorithm, be careful of oversights that can make the algorithm faulty. If this happens, they'll take off a hefty 15 or so points. I know this because of 2 instances where they thought my algorithm was faulty and took off that many points, but I explained in a regrade request why the algorithm is correct and got the points back.
- For programming homeworks, make sure your test cases not only cover most of your code (they check for code coverage of your test cases anyway) but also that you're fairly confident your code works in all situations. You usually only get 2 free submissions per programming assignment. Also, make sure to read the Javadocs in the stub files carefully as it may have instructions that the write up on the course page doesn't have.
- Be wary of a monster programming homework with 8 parts about two-thirds of the way into the course. It's almost double the length of a normal homework, but they give you about 2 weeks on it for that reason.
- If you didn't think all the details mattered for the material, you will once you take the exams.
- You should strive to have the intuition of the material ingrained in your brain as you go. You don't want to be cram memorizing the week before an exam. But it's perfectly fine if some details slipped your mind and you need a refresher the week before.
- When they design the exams, they assume everyone knows the basics of the material and what was taught in class. What the professor wants to do is separate mediocrity from true expertise by seeing who knows the ins and outs of the material rather than who can memorize a bullet point list.
- So if you memorize everything but don't really understand why things work the way they do, you'll have trouble on exam questions where they make you apply the knowledge you already know. In that case, you'll get a decent grade but not a great one.
- It's worth keeping in mind exams will be about 65% of your grade (percentage could vary depending on the course iteration).
- Do keep in mind that the course is curved. So try to get as many points as you can on an exam, but don't have a panic attack if you're not able to solve everything. The test section has statistics posted so you have an idea of what the average could be, but test difficulty will change depending on the course iteration.

§1.3 What is an algorithm?

Imagine you need to get something done according to some rules. Like you have a physical dictionary and need to look up a word, say the word "computer." You know it's in alphabetical order. That's an essential aspect to making a process to finding your word. Maybe you flip to a random page and see words beginning with 'd.' You know you're too far to the right because 'd' comes after 'c.' So you need to start flipping to the left. But now, you end up at words starting with 'b,' which comes before 'c.' So you're too far to the left. From there, you can slowly but surely narrow down where the word is in the dictionary until you find it. This is an algorithm at play: you use a set of rules to accomplish a task.

But efficiency is a key component. An inefficient algorithm that works with a lot of data could literally take decades to complete. What good is that if you have to wait that long for the result when there could be a much faster algorithm? But also, what good is an algorithm if your computer doesn't have enough memory for it to execute?

§2 GCD

Here, we'll get a first taste of what algorithm efficiency is all about. First, we define the GCD or greatest common divisor of two integers. The GCD of two integers a and b is the largest positive integer d such that d divides both a and b.

Review 2.1. Recall from CIS 1600 that an integer m divides n if and only if $n = km$ for some integer k.

§2.1 Naive Algorithm

In general, a "naive algorithm" is usually a simple algorithm that will accomplish the task. However, it doesn't leverage any shortcuts to make the work efficient. When working with larger numbers, this can be a massive problem.

We know the gcd of a and b is essentially the same as the gcd of b and a. So we can, without loss of generality, assume $a \geq b$ (because if $a < b$, we can just swap the values of a and b without issue). We know anything that divides some integer n cannot be greater than n itself. So it follows that any common divisor of a and b cannot be greater than b.

So the naive way is to check each integer from 1 to b, see which ones divide both a and b, and output the largest that divides both.

This gets the job done with something like $\gcd(160, 120)$ without taking too long, since the numbers are small enough. But what about $\gcd(33554435, 33554432)$? Or when working with even bigger numbers? Is there a more efficient way?

§2.2 Euclid's Algorithm and Its Proof

The notation $x \pmod{y}$ is essentially saying, the remainder when x is divided by y.

Lemma 2.2 (Euclid's Algorithm)

For positive integers a and b such that $a \geq b$, $\gcd(a, b) = \gcd(b, a \mod b)$

Proof. Let's prove this.

We know mod is basically remainder. Think about division back in elementary school where we have a quotient and remainder. So we can write $a = bq + r$ for integers q and r with $0 \leq r < b$. Essentially, q and r are the quotient and remainder, respectively, when a is divided by b.

Take a look at the equation $a = bq + r$. It's pretty clear intuitively that if some integer divides both b and r, then it must divide a. So any common divisor of b and r also divides a. You can prove it mathematically using the formal definition of divisibility. Anyway, that result we got means that $\gcd(b, r)$ (which is a common divisor of b and r) divides both a and b. So $\gcd(b, r) \leq \gcd(a, b)$ because $\gcd(b, r)$ is a common divisor of a and b, but it is at most the literal largest common divisor of a and b.

We can also rearrange the equation to $a - bq = r$, where it becomes clear that any common divisor of a and b also divides r. Do the same logic to get $\gcd(a, b) \leq \gcd(b, r)$.

The only way for $\gcd(b, r) \leq \gcd(a, b)$ and $\gcd(a, b) \leq \gcd(b, r)$ to hold simultaneously is if they're equal. So $\gcd(a, b) = \gcd(b, r)$, which completes the proof. □

So basically, we can keep using this lemma to keep changing the gcd into something else, until it becomes a base case.

I'll just tell you that the base cases are $\gcd(x, 0) = x$ for any positive integer x and $\gcd(x, 1) = 1$ for any positive integer x.

It's unexpected that you would come up with this entire proof from scratch, having never seen it before. But do understand the steps because you can apply a similar process to similar problems. The idea was to consider the two gcds separately and extract information on what relevant variables they divided.

§2.3 How fast is Euclid's Algorithm?

First, we have to prove a lemma:

Lemma 2.3

If $r \equiv a \pmod{b}$ with $a \geq b$, then $r < \frac{a}{2}$.

The intuition is that when you divide a by b, the remainder is not going to be that big compared to b. And since $b \leq a$, it is going to be even smaller compared to a. Like imagine if $a = 94$ and $b = 10$, in which $r = 4$. So r is smaller than b, which in turn is smaller than a. But want to see the math in action? I've got that covered for you.

Proof. We have 2 cases to consider:

Case 1: $a \geq 2b$

Remember that r is the remainder when a is divided by b. So since r is a remainder, $r < b$ or $b > r$.

Then, $a \geq 2b > 2r \implies a > 2r \implies r < \frac{a}{2}$.

Case 2: $a < 2b$

We know $a \geq b$, so $b \leq a < 2b$. Clearly, these bounds mean that when a is divided by b, the quotient will be 1. So $a = b + r$, which rearranges to $r = a - b$. But $a < 2b$ or $-b < -\frac{a}{2}$, meaning that $r = a - b < a - \frac{a}{2} = \frac{a}{2} \implies r < \frac{a}{2}$. □

So applying the Euclidean algorithm once on a and b means we transform $\gcd(a, b)$ to $\gcd(b, r)$, where $r \equiv a \pmod{b}$. By the lemma, $r < \frac{a}{2}$. Applying it a second time means we transform $\gcd(b, r)$ to $\gcd(r, s)$, where $s \equiv b \pmod{r}$. By the lemma, $s < \frac{b}{2}$. So after applying Euclid's algorithm twice, we went from $\gcd(a, b)$ to $\gcd(r, s)$, where $r < \frac{a}{2}$ and $s < \frac{b}{2}$. Essentially, we halved both inputs in 2 steps.

§2.4 Comparing Runtimes

Definition 2.4 (Log function (informal definition))**.** For an integer n, the log function $\log_2(n)$ denotes the number of times you have to halve n until you get a number less than 2.

For example, $\log_2(16) = 4$ because $16 \to 8 \to 4 \to 2 \to 1$, so we have to halve 16 a total of 4 times until it becomes less than 2.

This function is extremely important throughout the course, but we'll get to that later.

Anyway, we end once we hit a base case. We could get lucky in the process and at some point, it becomes computing $\gcd(c, 0)$ for some integer c. If this happens, we're done because as stated earlier, $\gcd(c, 0) = c$. But what if we're not so lucky and have to keep halving? Well, even in the worst case scenario where we have to keep halving, if we halve the inputs enough times, it will eventually get to less than 2, which means it gets to 1, and that's a base case.

How many times do we have to do this? Well, we do 2 steps to halve both inputs. By the definition of the log, this means we have to do this a total of $2\log(b)$ times, worst case scenario.

Now, let's just use a concrete example just to show how much of a difference the two algorithms make.

Suppose we needed to calculate $\gcd(33554435, 33554432)$. In the naive way, we'd have to test all the way up to 33554432 for both inputs, so that's $2 \cdot 33554432 \approx 67$ million times. Yikes! And that only gets worse the higher the numbers are. I'll tell you that $\log_2(33554432) = 25$. So we only need to do at most $2 \cdot 25 = 50$ steps with Euclid's, because the inputs halve each time, which allow us to more quickly reach a base case. Much better!

We say that Euclid's algorithm is $O(\log b)$, while the bad method is $O(b)$. What does the big O mean? We'll get to that later.

§2.5 Summary

We used some divisibility tricks in order to prove a useful lemma about gcd and then make a recursive algorithm out of it. We also saw how much of a difference we can make in the number of steps we have to do by implementing a more efficient algorithm.

§3 Crash Course to Runtime and Its Notation

Runtime will be a concept that will follow you around this course from start to finish. Essentially, with an algorithm and input of size n, we define a function $T(n)$ that models the algorithm's runtime in terms of n. However, in this course, we don't care too much about $T(n)$ exactly but rather, bounds on it and the rate of T's growth as n grows. For example, if T is constant regardless of n, we don't care about what constant it is equal to; just the property that T is a constant function.

Obviously, the smaller the runtime, the better because that means the algorithm is faster.

§3.1 Definitions and Seeing Them in Use

This section is unavoidably math heavy, but I will do my best in using Layman's terms. We need to get some important definitions over with though.

Definition 3.1 (Big O notation)**.** A function $f(n)$ is $O(g(n))$ (for some function $g(n)$) if and only if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all integers $n \geq n_0$.

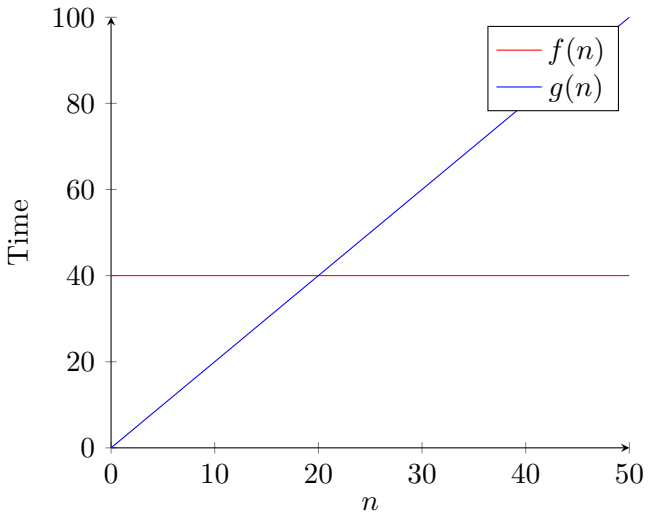
For example, the function n is $O(n^2)$ because $n \leq n^2$ for all integers $n \geq 1$. Also, $2n$ is $O(n)$ because if $c = 2$, then $2n \leq cn$ for all $n \geq 1$. Essentially, when we say $f(n)$ is $O(g(n))$, we're saying the rate of growth of f in the long run is equally as ideal or more ideal than the rate of growth of g in the long run. In other words, the growth rate of g poses as an upper bound for the growth rate of f.

Here's the intuition behind the constant c. Basically, multiplying any function by a constant doesn't change the general pattern in its rate of growth. Multiply a constant function by a constant? Still a constant function. Multiply a linear function (a function in the form $f(x) = ax$ for a constant a) by a constant? It will still be a linear function and grow like one. When we analyze runtimes, we don't care about these constants, just how increasing n effects the behavior of the growth of the runtime. For example, if an algorithm's runtime is linear, we don't care if it's $4n$ or $9n$ or $7128n$, just that it's linear. So having the c with $f(n) \leq cg(n)$ is essential to allow us to say constant factors in f don't matter because we could choose c appropriately to counter out any effects those constants may have on the left hand side of the inequality. We saw this in our $2n$ vs n example earlier where we used $c = 2$ to counter the 2 constant on the left.

Here's the intuition behind the constant n_0. When we analyze function runtimes, we want to look at the big picture, or which will be better/worse in the long run rather than a few small inputs. Take the set of functions $f(n) = 40$ and $g(n) = 2n$. Consider this table with a few small inputs for n:

n	1	5	10	15
$f(n)$	40	40	40	40
$g(n)$	2	10	20	30

If this is all we looked at regarding f and g, we would see f has worse time and foolishly conclude g is the better function. But we don't care about some cherry picked, small inputs. We care about looking at f and g in the long run. In fact, let's graph the two functions:



It looks like $g(n)$ only starts to become worse than $f(n)$ after a certain point. In fact, if g simply had a worse growth rate than f in general, g would always have to show its "true colors" as the worse function after n becomes large enough. The only reason g didn't show its true colors right away is because it had a headstart in less time over f with the small inputs. But the headstart wore off later.

That's why we define a cutoff point n_0 and only care about $n \geq n_0$. This circumvents these headstarts and allow us to focus more on a function's true colors and how it behaves in the long run.

Now, let's return to the original definition of big-O notation with the $f(n) = 40$ and $g(n) = 2n$ example above. So g's linear behavior is going to be worse than f's constant behavior in the long run, so we expect $f(n)$ to be $O(g(n))$. In fact, for all $n \geq 20$, we can see that $f(n) \leq g(n)$. So our choice of constants is $c = 1$ and $n_0 = 20$ to conclude that $f(n)$ fits the definition of $O(g(n))$.

Now, for definitions involving similar language and variables, but with crucial distinctions:

Definition 3.2 (Big Omega notation)**.** A function $f(n)$ is $\Omega(g(n))$ (for some function $g(n)$) if and only if there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all integers $n \geq n_0$.

Essentially, if $f(n)$ is $\Omega(g(n))$, the growth rate of g poses as a lower bound (instead of upper) for the growth rate of f. For example, $2n$ is $\Omega(n)$ and is also $\Omega(40)$.

Definition 3.3 (Big Theta notation)**.** A function $f(n)$ is $\Theta(g(n))$ (for some function $g(n)$) if and only if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$

Essentially, if $f(n)$ is $\Theta(g(n))$, the growth rate of f and g are more or less the same. For example, $2n$ is $\Theta(n)$ (because it is $O(n)$ and $\Omega(n)$).

Warning 3.4. Once you get the hang of these notations, it should become easy to simplify big-O of some expression without much justification. But be warned that on homework or midterms, they may ask you to explicitly find constants c and n_0.

We should practice with problems involving explicitly finding c and n_0.

Example 3.5 (Class)

Prove that $\frac{n^2}{5} + 500n + 1000$ is $\Theta(n^2)$, while finding constants c and n_0.

So we need to prove $\frac{n^2}{5} + 500n + 1000$ is both $O(n^2)$ and $\Omega(n^2)$, by definition of big Theta. Note that the values of the constants we choose for O can be different from those we choose for Ω, as long as they work in both individual cases.

First, let's do $O(n^2)$. We need to find a c and n_0 such that

$$\frac{n^2}{5} + 500n + 1000 \leq cn^2$$

for all $n \geq n_0$. One thing we do know is, for all positive integers n, we have $n \leq n^2$ and $1 \leq n^2$. So $500n \leq 500n^2$ and $1000 \leq 1000n^2$ for all $n \geq 1$. So in fact, for all $n \geq 1$,

$$\frac{n^2}{5} + 500n + 1000 \leq \frac{n^2}{5} + 500n^2 + 1000n^2 \leq 1500.2 \cdot n^2.$$

Then, c and n_0 naturally come out: we have $c = 1500.2$ and $n_0 = 1$.

Now, we do $\Omega(n^2)$. This is even easier: since $500n + 1000$ is always positive for all $n \geq 1$, we know $500n + 1000 \geq 0$. Thus,

$$\frac{n^2}{5} + 500n + 1000 \geq \frac{n^2}{5}.$$

So our constants are $c = \frac{1}{5}$ and $n_0 = 1$.

This proves $\frac{n^2}{5} + 500n + 1000$ is both $O(n^2)$ and $\Omega(n^2)$, so it is $\Theta(n^2)$. ■

Let's try a harder example, but this will necessitate induction.

Example 3.6 (Lecture Notes)

Prove that $\log_2(n)$ is $O(n)$, while finding constants c and n_0.

Before we get into the proof, let me just say, this result will be extremely useful for you from now until the end of the course. DON'T FORGET IT.

Anyway, this one looks harder. In the prior example, proving it was $O(n^2)$ and $\Omega(n^2)$ wasn't too hard because both $\frac{n^2}{5} + 500n + 1000$ and n^2 were sort of, in the same "family." But not here.

Irrespective, we want to find positive constants c and n_0 such that $\log_2(n) \leq cn$ for all $n \geq n_0$.

Here's a key observation: we can cherry pick a value of c and n_0 to our liking in making things work out, like we did last example.

I gave a hint towards induction, but in trying to work stuff out, let's try to do the inductive step and pick c that makes it convenient.

So the inductive step would involve proving

$$\log_2(k) \leq ck \implies \log_2(k+1) \leq c(k+1)$$

for some arbitrary k greater than the base case (which we'll get to later). We're assuming $\log_2(k) \leq ck$ is true, so let's make it look more like what we want to prove. We can do so by adding $\log_2(k+1) - \log_2(k)$ to both sides to get:

$$\log_2(k+1) \leq ck + \log_2(k+1) - \log_2(k).$$

Take my word on this that $\log_2(x) - \log_2(y) = \log_2(\frac{x}{y})$ for any positive real numbers x and y (we'll get to log rules later). So we have

$$\log_2(k+1) \leq ck + \log_2\left(1 + \frac{1}{k}\right).$$

Now, if we can prove that $ck + \log_2\left(1 + \frac{1}{k}\right) \leq c(k+1)$, we'd be done with the inductive step because then,

$$\log_2(k+1) \leq ck + \log_2\left(1 + \frac{1}{k}\right) \leq c(k+1) \implies \log_2(k+1) \leq c(k+1).$$

So we work with

$$ck + \log_2\left(1 + \frac{1}{k}\right) \leq c(k+1) \implies \log_2\left(1 + \frac{1}{k}\right) \leq c.$$

We know k is an arbitrary integer larger than the base case, so k could be ridiculously big. However, $\frac{1}{k}$ gets smaller as k gets bigger. In fact, for $k \geq 1$, it's easy to see that $1 + \frac{1}{k} \leq 2$ (because equality holds at $k = 1$, but the left hand side keeps getting smaller and smaller after that). If you halve something that's less than or equal to 2, you're guaranteed to get something less than 2 after that one halve, so by the informal definition of a log, we have $\log_2(1 + \frac{1}{k}) \leq 1$ for $k \geq 1$. So we can choose $c = 1$ with the caveat that $k \geq 1$ for the inductive step to work.

That's the induction step covered and now, we need to make sure the base case holds, which will be $n = n_0$ (basically the "cutoff" point where we want to start examining the functions). So we would need $\log_2(n_0) \leq n_0$, since we chose $c = 1$. The good thing is n_0 doesn't have to be optimal; it just has to satisfy the base case and work with the prior caveat that $k \geq 1$ for the inductive step to work (imagine if our prior caveat was $k \geq 101$ for the inductive step to work. Then, we couldn't just choose $n_0 = 97$ for example because then the inductive step fails for $k = 98$). Anyway, we know $\log_2(2) = 1$ (halve it once to get it under 2). So $n_0 = 2$ works as a base case. And when we do the inductive step for an arbitrary integer $k \geq n_0 = 2$, we don't violate the caveat, so we're good.

Now that we found convenient constants to work with, we can rewrite the induction proof into something more familiar by inserting in the values of those constants:

Solution:

Let $c = 1$ and $n_0 = 2$. I will show that $\log_2(n) \leq n$ for all integers $n \geq 2$.

Base Case:

Since $\log_2(2) = 1$, we have $\log_2(2) \leq 2$. Check.

Inductive Step:

Assume that $\log_2(k) \leq k$ holds for an arbitrary integer $k \geq 2$. I will show this implies that $\log_2(k+1) \leq k + 1$. If $\log_2(k) \leq k$ holds, then

$$\log_2(k) + (\log_2(k+1) - \log_2(k)) \leq k + (\log_2(k+1) - \log_2(k))$$

$$\implies \log_2(k+1) \leq k + \log_2\left(1 + \frac{1}{k}\right).$$

We know for $k \geq 1$ (which will always hold for $k \geq 2$), we have $1 + \frac{1}{k} \leq 2$. Then, $\log_2(1 + \frac{1}{k}) \leq 1$. Thus,

$$\log_2(k+1) \leq k + \log_2\left(1 + \frac{1}{k}\right) \leq k + 1,$$

completing the induction.

Advice 3.7 — When given a homework problem where you have to find the constants, this thought process will help where you try to do the inductive step and choose a c that makes it work out. Then, you choose n_0 so that the base case works. You do have to go back and write it up like a normal induction proof, but at least you will know specific values for the constants that work.

Example 3.8 (Class)
True or false: 3^n is $O(2^n)$.

Let's go about this trying to find c and n_0 until we either find it, or we reach a contradiction. So we need $3^n \leq c \cdot 2^n$ for all $n \geq n_0$. This rearranges to $(\frac{3}{2})^n \leq c$ for all $n \geq n_0$.

Here's the thing: we can choose c to be whatever we want, but if it turns out we cannot choose an appropriate n_0 to fit the definition of big-O after choosing c, our choice of c has failed. For example, suppose we chose $c = 400$. It turns out the inequality holds for $n = 1, 2, 3, ..., 14$, but for all $n \geq 15$, it no longer holds. So there is no n_0 for which it will hold for all $n \geq n_0$ because the n it fails for are limitless.

The issue here is that, take my word on this, the function $(\frac{3}{2})^n$ can always get as big as it wants if n is large enough. So even if we choose c to be 250 million or whatever, c is still a fixed value at the end of the day, since it's some constant. And guess what! The function $(\frac{3}{2})^n$ will eventually catch up to c (even if c is 250 million) and then the inequality no longer holds for n after that.

So we reached a contradiction, where we try to choose c but by doing so, we learn there is no n_0 (or cutoff point, essentially) where the inequality will hold for all future n past some n_0. This is because the left hand side will always catch up to c.

Thus, 3^n is NOT $O(2^n)$. In fact, 3^n is $\Omega(2^n)$.

§3.2 General Facts about Runtime

Now that we have the hang of runtime, I'll give you some useful facts that will be helpful to eyeball bounds on certain functions.

- For a degree d polynomial in n, meaning a function in the form $f(n) = a_d n^d + a_{d-1} n^{d-1} + ... + a_1 n + a_0$ for constants a_d, a_{d-1}, ..., a_1, a_0 with $a_d > 0$, we have $f(n)$ is $\Theta(n^d)$.
- Any nth degree polynomial is always $O(m^n)$, when $m > 1$ is some constant. So any nth degree polynomial is always $O(1.5^n)$, for instance.
- Logs are always $O(n)$ (proven earlier)

So we have this hierachy where logs are better than polynomials, but polynomials are better than exponential functions.

§3.3 Constant Time Operations

Here are some examples of constant time operations (meaning these run in $\Theta(1)$):

- Checking if two primitive data types (boolean, char, int, etc.) are equal to the same value
- Accessing an element at a certain position in an array
- Adding two numbers (adding n numbers is NOT $\Theta(1)$ though)

- Comparisons like "is $x > y$?"

These will be useful for you to know when you need to analyze the runtime of your algorithm. Just take my word on these; don't try to seek out an explanation.

§3.4 Summary

- We learned about runtime and how we define different notation to signify upper bound, lower bound, or same ballpark in growth.
- c is used in these definitions to dilute the effects of constants in front of the functions because we don't care about them when measuring growth.
- n_0 is used so we can see a function's true colors in the long run and not make faulty assumptions based on a couple of small inputs.
- Sometimes there are cases where you don't have to do too much work to get c and n_0. Other times, you have to do the math or even induction. But with induction, you can try to choose c and n_0 to make the process convenient.

§4 Log Rules

I taught you an informal definition of logs, specifically $\log_2$. Now, here's the formal definition:

Definition 4.1 (Log function (formal definition))**.** The inverse of an exponential function: for a base b and positive real number x, we have that $\log_b(x)$ is equal to the real number r such that $b^r = x$.

The base can be any positive real number besides 1. Most commonly in this course, you'll see $b = 2$. An example with another base: we know that $3^4 = 81$, so we can say $\log_3(81) = 4$.

Here are some properties of logarithms:

- $\log_b(b^x) = x$, essentially the fundamental definition of logarithms
- $\log_b(x) + \log_b(y) = \log_b(xy)$ (DO NOT USE THIS IF THE BASES ARE NOT THE SAME) Very useful for combining logarithms.
- $\log_b(x) - \log_b(y) = \log_b(\frac{x}{y})$
- Change of base formula from base b to a new base c: $\log_b(x) = \frac{\log_c(x)}{\log_c(b)}$. Since $\log_c(b)$ is essentially a constant when b and c are both fixed, this means changing the base of a log is just multiplying by some constant.
- $\log_b(x) = \frac{1}{\log_x(b)}$, uses the above formula by changing from base b to base x
- $b^{\log_b(x)} = x$, useful for getting rid of logarithm terms in the exponent
- $\log_b(c^x) = x \cdot \log_b(c)$, useful for getting rid of exponents inside log terms or putting them inside to later combine log terms.
- $x^{\log_b(y)} = y^{\log_b(x)}$, occasionally useful if y and x are better off switching places
- If $b > 1$ is fixed, while x is changing, the value of $\log_b(x)$ increases as x increases.
- $\log_b(1)$ is always 0. Using the above point, this means that $\log_b(x) > 0$ when $x > 1$ (so long as $b > 1$).

While there is no shame in referring to these from time to time, you should ideally try to memorize them, especially for the closed notes midterms. To make it easier to memorize, it helps to understand the intuition. With the addition property, it comes from the fact that we add exponents when multiplying, like $2^3 \cdot 2^2 = 2^5$, so $\log_2(2^3) + \log_2(2^2) = \log_2(2^5)$, which becomes $3 + 2 = 5$.

Also, the base tends to be irrelevant with logs in this course, as long as it is a fixed number for all terms you're using it with. So really, there's no difference between using $\log_2$ or $\log_3$, as long as all relevant terms are assumed to be all base 2 for example, and you don't try to combine $\log_2$ and $\log_3$ terms.

Example 4.2 (Recitation)

Prove that $\log(n!) = \Theta(n \log n)$, while finding constants c and n_0. Assume base 2 for all the logarithms.

We know $n! = n \cdot (n-1)!$ and $1! = 1$ from CIS 1600. Let's use our trusty log rules, specifically that addition property one. Using this, we know $\log(n!) = \log(n) + \log((n-1)!)$. Taking this a step further, $\log(n!) = \log(n) + \log(n-1) + \log((n-2)!)$. Do this enough times until we get to the base case for factorial, and we get

$$\log(n!) = \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(2) + \log(1)$$

for an arbitrary positive integer n. Now we'll get both upper and lower bounds on this.

First, an upper bound. We know log increases as its input increases by a given property I listed. So for the sake of upper bound, we can bump all terms up to n:

$$\begin{aligned}\log(n!) &= \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(2) + \log(1) \\ &\leq \underbrace{\log(n) + \log(n) + \log(n) + \cdots + \log(n) + \log(n)}_{n \text{ of these}} = n \cdot \log(n).\end{aligned}$$

Since n is an arbitrary positive integer, the above holds for all such n. So we showed $\log(n!)$ is $O(n \log n)$ with $c = 1$ and $n_0 = 1$.

Now, a lower bound. We know each term in the sum is nonnegative because $\log(1) = 0$ and increasing the input will result in the log being a positive value. So deleting some terms from the sum is only going to decrease its value for a lower bound. Thus, let's delete everything less than $\log(n/2)$:

$$\begin{aligned}\log(n!) &= \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(2) + \log(1) \\ &\geq \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(n/2+1) + \log(n/2).\end{aligned}$$

Similar to what we did previously, we can bump down the values in each log term for a lower bound:

$$\begin{aligned}\log(n!) &\geq \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(n/2+1) + \log(n/2) \\ &\geq \underbrace{\log(n/2) + \log(n/2) + \log(n/2) + \cdots + \log(n/2) + \log(n/2)}_{n/2 \text{ of these}} = \frac{n}{2} \cdot \log(n/2).\end{aligned}$$

Since we're using base 2, using that division/subtraction log property:

$$\log(n!) \geq \frac{n}{2} \cdot \log(n/2) = \frac{n}{2} \cdot (\log(n) - \log(2)) = \frac{n}{2} \cdot (\log(n) - 1).$$

Note that $\log(4) = 2$. Since we know about log's increasing properties, we know $\log(n) \geq 2$, when $n \geq 4$. So assume $n \geq 4$. Then, $\frac{\log(n)}{2} \geq 1$. This becomes $\log(n) - 1 \geq \frac{\log(n)}{2}$. Back to our work,

$$\log(n!) \geq \frac{n}{2} \cdot (\log(n) - 1) \geq \frac{n}{2} \cdot \frac{\log(n)}{2}.$$

But this works only when $n \geq 4$. So we can set $n_0 = 4$ and $c = \frac{1}{4}$ to say $\log(n!)$ is $\Omega(n \log n)$.

Thus, we can conclude that $\log(n!)$ is $\Theta(n \log n)$, since we showed both O and Ω. Side note, remember how we worked in base 2, but one of those properties said changing the base is essentially multiplying by a constant? That means the Θ holds, even if chose some other base besides 2, since we don't care about constants in Θ analysis.

Extra side note, you know how we deleted everything less than the $n/2$ term when doing the Ω bound? That technique is helpful in general with these bounds because you can try to get stuff in the desired behavior you want for that bound. We'll see something later where we will want to use this technique.

§5 Binary Search (and the real start of algorithm learning)

§5.1 Introduction

You may have heard of this algorithm before, but I'll go over it, so we're all on the same page. Now that we have some of the basics down on runtime, we can get into the structure of how algorithms are taught. There are three key components they look for with algorithms, particularly on the homeworks:

- The algorithm itself. They usually want you to write it in plain English and no psuedocode. There is psuedocode in the lecture notes, but my guide will dumb it down so it is easy to understand and follow along.
- A proof of correctness. This is basically showing why the algorithm returns what we want it to do. There is some nuance to how they may want you to do it, but we can get in to that later.
- Runtime. Basically justifying that the algorithms runs in the given runtime. At the point they start making you do this on homework, you don't need to be super precise on details. For example, you don't need in-depth justification on why $O(2n)$ is $O(n)$, but it never hurts to ask on the class board privately.

Anyway, let's get in to a simple example of an algorithm. Say we have an array of size n that we know is sorted in increasing order. How do we check if a particular number is in the array?

Well, we could go through each of the n cells in the array. Worst case, we go through the entire array and don't find our target at all. Each cell requires constant time work (checking if the number there is equal to the one we're looking for) and we do this n times. So our runtime is cn for some constant c, meaning this naive algorithm is $O(n)$.

But we can do better than this, especially with leveraging the fact that the array is sorted in increasing order.

> **Review 5.1.** If you're rusty on recursion from CIS 1200, review it now. A number of algorithms here on out will rely on it. Remember the idea is, it's a function that calls itself to solve a similar problem but of smaller size to make it easier to work out the final task.

§5.2 The Algorithm

So we have two inputs: our array and a target element, which is the element we're searching for in the array. Here's the algorithm:

Base Case: If the array is of size 1, check if the lone element is equal to our target element. If it is, return true. Otherwise, return false.

Otherwise, we have an array of size greater than 1. Check the element in the very middle of the array. Then, we do one of the three things:

- If this middle element is equal to our target element, we found it, so we return true.
- If the middle element is less than the target, recursively call the algorithm on the portion of the array to the RIGHT of the middle.
- If the middle element is greater than the target, recursively call the algorithm on the portion of the array to the LEFT of the middle.

§5.3 In Action

Let's suppose we have this sorted array and we want to search for 7:

0	1	2	3	4	5	6
1	3	4	5	7	11	15

Okay, so at the middle of the array is 5. This is less than our target, so we consider the right half of the array (positions 4-6) and recurse there. So it boils down to this:

4	5	6
7	11	15

Check the middle, it is 11, bigger than our target. So we recurse on the left half (position 4).

4
7

We're at a base case (1 element). So we just check this lone element. Yes, it is 7, our target, so we return true.

§5.4 Proof of Correctness (and intro to PoC via Induction)

> **Review 5.2.** Review induction and strong induction if you're rusty on those. Recursive algorithms rely on them.

So here's the informal way to think about it, considering the example we just did. The middle element is 5. We know the array is sorted in increasing order. So we know everything to the left of 5 (in the middle) is smaller than 5 because the array is sorted. So anything to the left of the middle is smaller than 5, which is smaller than 7, so we don't care about it.

Now, 7 could be in the right half. We don't know yet. But we essentially boiled it down to checking whether it is in the right half, which is why we recurse on that right half.

Similar logic holds if it turned out the middle was larger than the target, in which the right half is irrelevant and we only care about the left half.

It also shouldn't be too hard to see why the base case works (if it's 1 element, either the target is that lone element or it's not). Same with when we get lucky and the middle just so happens to be the target.

How THEY want you to prove a recursive algorithm though is through induction. I'm not going to go through all of it but just enough of the idea. First, the base case of our proof by induction is checking that the algorithm works for an array of size 1. This is essentially checking that the base case of the recursive algorithm does what we want (as will be what to do in resolving the base case for pretty much every proof of a recursive algorithm in this course).

Then, we need to do the inductive step in showing it works for an array of size $n = k$, where $k > 1$ is arbitrary. We'll use strong induction. So we justify why we search a specific half of the array in the cases as we did, instead of doing something crazy like randomly picking the half each time. But here's the inductive part: when we make the recursion, it boils down to the inductive hypothesis holding for a smaller n because we're essentially doing the same problem with a smaller size. We know the algorithm will work for a smaller n by the inductive hypothesis, so this implies it holds for $n = k$. We used strong induction here because the smaller array isn't necessarily $k - 1$, more like $k/2$, so we need to use strong induction to make the proof work.

Also, with this example, you now more clearly see the parallels between induction and recursion.

> **Advice 5.3 —** When saying to work with an array of size $k/2$, don't worry about k being odd (where $k/2$ won't be an integer).

§5.5 Runtime

So we can agree that before we do the recursive call, everything is of constant work, right? We check if we're at a base case, then we check the middle element, and compare it to the target. All constant work.

We could end early if we're lucky and find the target in the middle, but worst case, we'll have to recurse on one half of the array (in other words, solve a subproblem of size $n/2$). So if $T(n)$ is a time function for worst case scenario, we can say $T(n) = c + T(n/2)$. We know the base case is when we have size 1 and that's clearly of constant time, so $T(1) = d$ for some constant.

Anyway, with the recurrence $T(n) = c + T(n/2)$, we can substitute n with $n/2$ so we get $T(n/2) = c + T(n/4)$. Plugging this back in means $T(n) = c + (c + T(n/4)) = 2c + T(n/4)$. But we can plug in $n/4$ into the original recurrence to get $T(n/4) = c + T(n/8)$, so we have $T(n) = 2c + T(n/4) = 2c + (c + T(n/8)) = 3c + T(n/8)$.

Again, if we get lucky and find it in the middle in any of the recursive calls, we finish early. But to simulate the worst case scenario, we assume it has to go all the way to base case, which is when it becomes $T(1)$. Notice that we keep halving the input, like going from $n/2$ to $n/4$ to $n/8$ and we add c each time. So at worst, we add c as many times as it takes to halve n until it gets to 1. This is $\log_2(n)$ times. So we can say $T(n)$ is roughly $c \log_2(n)$ meaning $T(n)$ is $O(\log n)$. Better than $O(n)$, for sure, so we didn't have to check everything in the array.

This subsection was not rigorous and formal, but we'll have techniques later to more formally solve recurrence relations. Hopefully, you get the general idea.

However, remember Binary Search only works for sorted arrays. And an array may not always be sorted. Well, in the next section, we'll learn how to sort an array.

§5.6 Summary

- We saw how recursion is used in a simple algorithm to find a specific element in an array.
- We basically use the middle element to determine which half of the array our target has to be in, should it be present in the array.
- We got a taste of a Proof of Correctness, where we have to reason through our steps.
- We saw how induction, specifically strong induction, is the primary tool for proving a recursive algorithm because it naturally has to deal with a similar situation but a smaller size.
- We saw how binary search is $O(\log n)$, which is better than checking every element one by one.

§6 Insertion Sort

§6.1 Introduction

So we need a sorted array to use binary search. And how do we sort an array, since not all of them will be that way? Well, let's see one such example of a sorting algorithm!

§6.2 The Algorithm

Suppose the array is of size n. Suppose the positions of the array are labeled 1 to n from left to right (so it doesn't start at 0).

Run this loop for j from 2 to n, incrementing j by 1 each time:

- Consider the jth element in the array.
- If it's smaller than the element to the left of it, have it switch places with the element on its left.
- If it's still smaller than the element to the left of it, make a switch again. Keep making switches with the element on the left, until we reach a point where it is bigger than the element to the left of it, or it reaches the left end of the array.

§6.3 In Action

Let's sort this array with insertion sort:

1	2	3	4	5	6
5	2	4	6	1	3

So first, we start at $j = 2$, so we go to position 2 and see the 2 there. Then, we compare it with 5 on its left. Since 2 is smaller than 5, we make the switch. Now, 2 is at the left end of the array, so we don't make any more switches. Here's what the array looks like now:

1	2	3	4	5	6
2	5	4	6	1	3

So now, we move on and increment j, so $j = 3$. We see 4 at position 3, which is less than 5. So we swap 4 and 5. But 4 is greater than 2, so we don't make another swap. Here's what the array looks like now:

1	2	3	4	5	6
2	4	5	6	1	3

Now, $j = 4$. 6 is greater than 5, so we don't make any swaps. The array didn't change.

You can see where this is going, so here's what it looks like after $j = 5$:

1	2	3	4	5	6
1	2	4	5	6	3

And $j = 6$:

1	2	3	4	5	6
1	2	3	4	5	6

The array is sorted. Do you see why it's called insertion sort after this example? We basically take each element one by one and find the appropriate place in our line so far to insert it, so stuff is in the correct order.

We could also implement this recursively, where the base case is when $n = 1$ and we just do nothing, since it's already sorted. Otherwise, we recursively sort the first $n - 1$ elements and figure out how to insert the nth element.

§6.4 Proof of Correctness (and intro to Loop Invariants)

Take a good look at the array again after $j = 2$:

1	2	3	4	5	6
2	5	4	6	1	3

Notice those 2 shaded cells and how 2, 5 are in order.

Now, look at what the array was after $j = 3$:

1	2	3	4	5	6
2	4	5	6	1	3

Notice those 3 shaded cells and how 2, 4, 5 are in order.

Is this a coincidence? No. Think of it sort of like induction.

Our base case is before we do anything, the first element on its own is already sorted.

The inductive step comes from the fact that, if the first i elements are already sorted and we're working on $j = i + 1$, the first $i + 1$ elements will be sorted after. This is because the "inductive hypothesis" implies that the first i elements are sorted. We find the appropriate place to insert the new element (which is the first time when the left element is less than the new element after the swaps). The swaps cause everything within the original i elements that is bigger than the new element to be on the right of the new element without destroying the relative order of the original elements. Everything on the left is in order, too. Thus, the $i + 1$ elements are in order. So this acts like induction where we show how one step being true implies the next step is true.

Basically, we showed there is this invariant where the first i elements are sorted after the iteration for $j = i$ is over. To formalize a loop proof in this class, they want to see a loop invariant, which has 3 main things:

- Initialization: Show that the invariant is true before the loop even starts. This is sort of like the base case. For insertion sort, this is when we showed the 1st element is already sorted.
- Maintenance: Show that if the invariant is true when the ith iteration ends, it still holds true after the $(i + 1)$th iteration ends. This is like the inductive step. For insertion sort, this is when we showed inserting the new element in the way we did still keeps everything in order.
- Termination: That the loop gives us what we want at the end. For insertion sort, the initialization and maintenance steps ensure the new element we add maintains order. We end with the nth iteration where we have n elements in order. In other words, we have the sorted array, which is exactly what we want.

§6.5 Runtime

Okay, so we can agree that the source of the work done in insertion sort is through checking whether we need to make swaps and actually doing swaps, right? Making a single check is constant work because we just compare two variables. Making a single swap is also constant work: just make a temporary variable to save the values being swapped and then do the swap. So now, it boils down to how many checks and swaps we actually do.

To organize it, let's count the number of checks and swaps separately, by iteration. So how many happen during the jth iteration? Guess what. It depends on the original array! Say like we had the numbers 2, 3, 4, 5 sorted so far in an array. If we wanted to insert 6, we don't have to make any swaps. But if we wanted to insert 1, we would have to make multiple swaps. Okay, so even for arrays of the same size, there could be very different runtimes! So instead, let's consider bounds on the number of checks and swaps total.

Let's define C_j and S_j to be the number of checks and swaps, respectively, in the jth iteration. Let's let a and b be the constants representing the time for one check and the time for one swap, respectively. The runtime of insertion sort will be $T(n) = \sum_{j=2}^{n}(a \cdot C_j + b \cdot S_j)$. This is because in the jth iteration, we do C_j checks, which take a time each, so that's $a \cdot C_j$ time total. Similar thing goes for $b \cdot S_j$. And then of course, we sum over all iterations.

Let's suppose it's the jth iteration, where the first $j - 1$ elements are sorted and we're inserting a jth element. No matter what, it has to be checked with the element on the left. So $C_j \geq 1$ always. But if it's greater than that element, no swaps occur. So $S_j \geq 0$ always.

Now, let's consider worst case scenarios. Remember we keep making swaps until we either hit the left end, or we check the left element and see that it is less than the current element. If the latter is true, the iteration ends early. But in the worst case, we keep having to make swaps until we hit the left end. So we

have $S_j \leq j-1$ because we make $j-1$ swaps in the case we have to do it until we hit the left end. Also, $C_j \leq j$ when that happens because we keep checking with the element on the left, with the last check occurring when we're on the left end of the array.

So we have our formula $T(n) = \sum_{j=2}^{n}(a \cdot C_j + b \cdot S_j)$. If we assume the best case scenario, which is when C_j and S_j are as low as they can be, we get

$$T(n) = \sum_{j=2}^{n}(a \cdot 1 + b \cdot 0) = \sum_{j=2}^{n}(a) = a(n-1).$$

So the best case is $\Theta(n)$, which would occur when we don't have to make swaps, so something like $1, 2, 3, 4, 5, 6$.

If we assume C_j and S_j are as high as they can be, it becomes

$$T(n) = \sum_{j=2}^{n}(a \cdot j + b \cdot (j-1)) = a \cdot (2+3+4+\cdots+n) + b \cdot (1+2+3+\cdots+(n-1)).$$

Remember the sum of the first i positive integers is $\frac{i(i+1)}{2}$ from CIS 1600. So

$$T(n) = a \cdot \left(\frac{n(n+1)}{2} - 1\right) + b \cdot \frac{(n-1)n}{2}.$$

This is $\Theta(n^2)$, which would occur when we have to keep making swaps until we reach the end, so something like $6, 5, 4, 3, 2, 1$.

As you can see, there's no one size that fits all with the runtime of insertion sort, not when there's this disparity in the best and worst cases. And yes, that means for an arbitrary array we need to sort, sometimes it's close to the best case, other times it's close to the worst case. All we did is just find bounds.

§6.6 Summary

- We saw our first sorting algorithm, which is done by appropriately inserting numbers in an already sorted portion of the array.
- We saw how a loop invariant is used to show how the first j elements are sorted every step of the way.
- It turned out there was no one runtime to describe all cases of insertion sort. The best case is $\Theta(n)$ but the worst case is $\Theta(n^2)$.

§7 Runtime of Recurrences

§7.1 Introduction

So we saw binary search earlier, a recursive algorithm. Well, we need to have a strategy to get the runtime of a recursive formula because there could be variation in said formula. First things first, here's some important formula:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$1 + r + r^2 + \cdots + r^n = \frac{r^{n+1}-1}{r-1}$$

$$1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1 \quad \text{(A result of the formula above by using } r = 2\text{)}$$

These are usually provided for you on a test, but helpful to remember they exist so you know to go and reference it when need be.

§7.2 Expansion Method

Let's go back to the recurrence for the worst case for binary search (meaning the algorithm goes all the way down until we reach a base case without ending early). We don't care about how we derived the recurrence right now, just what it is.

$$T(n) = \begin{cases} T(n/2) + c, & n > 1 \\ k, & n = 1 \end{cases}$$

where c and k are some constants. If you recall, what I did was I kept plugging the recurrence into itself. To refresh your memory, we have this equation,

$$T(n) = T(n/2) + c \quad (1)$$

But we can plug $n/2$ for n in (1) to get

$$T(n/2) = T(n/4) + c \quad (2)$$

Plug this back in to (1) to get

$$T(n) = (T(n/4) + c) + c = T(n/4) + c + c \quad (3)$$

We plug $n/4$ for n in (1) to get $T(n/4) = T(n/8) + c$. Plugging this in to (3) implies

$$T(n) = (T(n/8) + c) + c + c \quad (4)$$

Monitoring our progress,

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + c + c \\ &= T(n/8) + c + c + c \end{aligned}$$

As you can see, we're expanding the equation out by feeding the recurrence in to itself, hence, the colloquial name for the technique. We can pretty easily see that the general form will be $T(n) = T(n/2^i) + i \cdot c$ at this point.

> **Advice 7.1 —** You can use induction here to prove this, but they don't require it. The pattern could easily start to emerge if you expand enough times.

Anyway, we know this recursion has a base case. With the $n/2$ going to $n/4$ then $n/8$, it gets smaller each time. So if we were to, in theory, expand it enough times, we would reach the base case. And that base case occurs when the input is 1. So we want to set $n/2^i = 1$ because that's when the expansions reach the base case. Solving for i,

$$n/2^i = 1 \implies n = 2^i \implies i = \log_2(n).$$

So plugging this in to the general form means

$$T(n) = T(n/2^i) + i \cdot c = T(1) + \log_2(n) \cdot c = k + c \cdot \log_2(n).$$

So $T(n)$ is $\Theta(\log n)$.

Example 7.2 (Class)
Find a Θ bound for $T(n)$ given the recurrence relation

$$T(n) = \begin{cases} 3T(n/2) + kn, & n > 1 \\ c, & n = 1 \end{cases}$$

Okay, I'll let you know this one is not as pretty and is math heavy, but bear with me. Let's again, do our technique where we plug the recurrence in to itself. Multiplying stuff out is fine, but let's refrain from adding.

$$\begin{aligned}
T(n) &= 3T(n/2) + kn \qquad (1)\\
&= 3\left(3T(n/4) + \frac{kn}{2}\right) + kn\\
&= 9T(n/4) + \frac{3kn}{2} + kn \qquad (2)\\
&= 9\left(3T(n/8) + \frac{kn}{4}\right) + \frac{3kn}{2} + kn\\
&= 27T(n/8) + \frac{9kn}{4} + \frac{3kn}{2} + kn \qquad (3)
\end{aligned}$$

From (1), (2), and (3), we can guess the pattern:

$$\begin{aligned}
T(n) &= 3^i T(n/2^i) + \left(kn + \frac{3kn}{2} + \frac{9kn}{4} + \cdots + \left(\frac{3}{2}\right)^{i-1} kn\right)\\
&= 3^i T(n/2^i) + kn\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^{i-1}\right)
\end{aligned}$$

We want to see the point at which we expanded enough times to hit a base case. So we want $n/2^i = 1$ or $i = \log_2(n)$, once again. So let's do some simplification, first using one of the formula I provided earlier in the section to sum up the powers of $\frac{3}{2}$:

$$\begin{aligned}
T(n) &= 3^i T(n/2^i) + kn\left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^{i-1}\right)\\
&= 3^{\log_2(n)} \cdot T(1) + kn\left(\frac{(3/2)^i - 1}{3/2 - 1}\right)\\
&= 3^{\log_2(n)} \cdot c + 2kn\left(\left(\frac{3}{2}\right)^i - 1\right)\\
&= 3^{\log_2(n)} \cdot c + 2kn\left(\frac{3^{\log_2(n)}}{2^{\log_2(n)}} - 1\right)\\
&= 3^{\log_2(n)} \cdot c + 2kn\left(\frac{3^{\log_2(n)}}{n} - 1\right)
\end{aligned}$$

Recall one of our log identities: $x^{\log_b(y)} = y^{\log_b(x)}$. This means $3^{\log_2(n)} = n^{\log_2(3)}$, which means the n can be swapped to a more convenient place.

$$\begin{aligned}
T(n) &= 3^{\log_2(n)} \cdot c + 2kn\left(\frac{3^{\log_2(n)}}{n} - 1\right)\\
&= n^{\log_2(3)} \cdot c + 2kn\left(\frac{n^{\log_2(3)}}{n} - 1\right)\\
&= n^{\log_2(3)} \cdot c + 2k \cdot n^{\log_2(3)} - 2kn\\
&= (c + 2k) \cdot n^{\log_2(3)} - 2k \cdot n
\end{aligned}$$

Since $\log_2(3) > 1$, the $n^{\log_2(3)}$ is the dominant term over n because of a larger exponent. So this is $\Theta(n^{\log_2(3)})$. This one was a doozy.

Advice 7.3 — I would always try to write down at least 2 or 3 expansions when I need to show my work. Don't always try to combine terms because a pattern like c, then $c+2c$, then $c+2c+4c$, then $c+2c+4c+8c$ would be much easier to see if you don't combine terms. If you did combine terms, you would get c, $3c$, $7c$, $15c$ in which the pattern is not as obvious.

§7.3 Master Theorem

Theorem 7.4

Suppose a recursive algorithm had a time function $T(n)$, and $T(1)$ is a constant. If the recurrence is

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k)$$

for some constants a, b, and k, then we have three cases:

- If $a > b^k$, then $T(n)$ is $\Theta(n^{\log_b a})$
- If $a = b^k$, then $T(n)$ is $\Theta(n^k \log_b n)$
- If $a < b^k$, then $T(n)$ is $\Theta(n^k)$

You will not be expected to memorize this, and you will be given this on tests. The proof? Not until CIS 3200.

Warning 7.5. On the homeworks and tests, you may get recurrence relation problems where they want you to use expansion and not Master Theorem. In that case, expand and DO NOT CITE THIS THEOREM AT ALL (or you may get 0 credit). Doesn't hurt to check in with the TAs and professors when in doubt.

Advice 7.6 — However, when trying to do an algorithm problem, this theorem is incredibly useful for checking the runtime of a recursive algorithm without having to use expansion. You may be able to cite it when doing a runtime analysis for designing an algorithm. Even in cases where they force you to use expansion, you can use the theorem to check your work.

§7.4 Summary

- We basically repeatedly substitute the recurrence in to itself and expand out the terms each step of the way.
- After enough times, we notice a general pattern of what the equation looks like the ith time we plug it in to itself.
- At that point, we can set i so that it reaches a base case, which is something we know the runtime of.
- Then, we unavoidably have to do some algebra. We do have formulas related to summations, should we need them.
- Master Theorem can help check our work or be useful later when we need to justify the runtime of a recursive algorithm but don't need to go through the trouble of expansion.

§8 Iterative Code Analysis

§8.1 Introduction

For loops are incredibly common in programming, so we ought to know how to analyze their runtime.

Let's look at a simple example:

```
for (int i = 1; i <= n; i++) {
    print("1210");
}
```

Well this just prints "1210" n times. Remember that printing is constant time. So this is $\Theta(n)$.

But here's a not so simple example:

```
for (int i = 1; i <= n; i++) {
    for (int j = i; j <= n; j++) {
        print("1210");
    }
}
```

Well, this is annoying. Not only do we have an extra inner loop, but the loop with j starts at a different value, depending on i.

Also, who says i has to only be increasing by 1 each iteration? We can have something like this:

```
for (int i = 1; i <= n; i = 2*i) {
    for (int j = i; j <= n; j++) {
        print("1210");
    }
}
```

This time i doubles each iteration instead of increasing by 1, so that's even nastier to deal with.

Well, we're going to learn some techniques for these iterative snippets of code.

§8.2 Superset-Subset Method

Let's get right in to an example where I explain the thought process.

Example 8.1 (Class)

Find a Θ bound on the runtime of this code snippet:

```
for i <- 1 to n (incrementing i by 1 each time)
    for j <- 1 to i*i (incrementing j by 1 each time)
        for k <- 1 to j (incrementing k by 1 each time)
            print("1210")
```

Yuckiness aside, let's think about this. So we're printing something (constant time) some number of times. So it boils down to how many times it runs. We know how nested for loops work: we keep incrementing inner loops until they reach their limit. Once that happens, we go back to the previous loop and increment there before restarting the inner loop.

First off, we'll have $i = 1$, $j = 1$, then $k = 1$ because those are the starting values for each variable. So with the given values, the j loop runs from 1 to 1^2. In other words, we only have $j = 1$ for now. Then, we have k running from 1 to j, which in this case, only runs from 1 to 1, since j is limited to 1. So the k loop reaches its limit immediately, but then going back, so does the j loop. So we only do $(i, j, k) = (1, 1, 1)$.

This is the end of the $i = 1$ iteration, so we increase i by 1, as stated in the for loop.

Now, $i = 2$ and we restart the j and k loops at their starting values. So we have j running from 1 to 4, and then k running from 1 to j. So we do $(i, j, k) = (2, 1, 1)$ then $(2, 2, 1)$, $(2, 2, 2)$, $(2, 3, 1)$, $(2, 3, 2)$, $(2, 3, 3)$, $(2, 4, 1)$, $(2, 4, 2)$, $(2, 4, 3)$, $(2, 4, 4)$. That marks the end of the $i = 2$ iteration.

You're starting to get the picture: every time we print, there's some triple (i, j, k) associated with it. So we want to count the number of such triples.

How about not count the number of triples directly, and instead, get bounds on them?

Let's try to get a superset of the triples, which serves as an upper bound. So in the for loop, we know $1 \leq i \leq n$ and $1 \leq j \leq i^2$ and $1 \leq k \leq j$.

Let's think about the j loop. When $i = 1$, we have j going from 1 to 1. When $i = 5$, we have j going from 1 to 25. When $i = n$, we have j going from 1 to n^2 (which is as big as the bounds can be).

Here comes the trick: the largest the stopping point is for the j loop is n^2, for all i that we work with. So let's deliberately extend the upper bound to be n^2 for the j loop, regardless of i. So now, we have a new set of triples where $1 \leq i \leq n$ and $1 \leq j \leq n^2$ and $1 \leq k \leq j$. Already much nicer to work with since the j loop's upper bound doesn't depend on i (since we know i changes occasionally).

Warning 8.2. You have to make sure when you extend the bounds, it actually results in the duration of the loop increasing for every i we work with and not decreasing any of them. Otherwise, we wouldn't have a superset.

If we said the stopping point for the j loop is $n^2/2$ for all i, that's a problem. Because when $i = n$, the j loop goes from 1 to n^2. But the new stopping point decreases the j loop for $i = n$, so some of the original triples won't make it into the "superset," and this defeats the purpose of a superset.

Now, the k loop. When $j = 1$, we have k going from 1 to 1. When $j = 7$, we have k going from 1 to 7. When $j = n^2$ (the max), we have k going from 1 to n^2. Again, let's deliberately extend the upper bound for the k loop to be n^2, regardless of the value of j.

So now, we have a new set of triples where $1 \leq i \leq n$ and $1 \leq j \leq n^2$ and $1 \leq k \leq n^2$. With our method of deliberately extending the loop duration (and never shrinking it), each of the original triples is covered in this new set. So this new set is a superset of the original triples. And it isn't too hard to calculate the runtime of this. By multiplication rule, it is $n \cdot n^2 \cdot n^2 = n^5$. But this superset is only an upper bound on the original set, so we can say the original set is $O(n^5)$.

We need an Ω bound too though because we're trying to prove a Θ. Good news is, we can play a similar trick. This time, we take the original bounds and decrease them instead of extend them. Again, the original loops were $1 \leq i \leq n$ and $1 \leq j \leq i^2$ and $1 \leq k \leq j$.

Let's suppose we changed the i loop to $n/2 \leq i \leq n$. We're deliberately cutting off the triples involving $i = 1$, $i = 2$, $\cdots$, $i = n/2 - 1$, but that's okay when this new set we're making is going to be a subset of the original one that's easier to count.

So our new set involves $n/2 \leq i \leq n$ and $1 \leq j \leq i^2$ and $1 \leq k \leq j$. Let's see how the j loop works as we go through all values of i.

When $i = n/2$, we have j going from 1 to $n^2/4$. When $i = n/2+1$, we have j going from 1 to $(n/2+1)^2$. When $i = n$ (the largest it can be), we have j going from 1 to n^2. Let's deliberately shrink the j upper bound to always be $n^2/4$, since for every i, the j loop either stops at $n^2/4$ or goes past it. So now we have $n/2 \leq i \leq n$ and $1 \leq j \leq n^2/4$ and $1 \leq k \leq j$.

Let's work on the k loop. For $j = 1$, we have k going from 1 to 1. For $j = 2$, we have k going from 1 to 2. For $j = n^2/4$ (the highest j can be), we have k going from 1 to $n^2/4$. Let's deliberately shrink the k upper bound to 1, since for every j, the k loop either stops at 1 or goes past it. So now we have $n/2 \leq i \leq n$ and $1 \leq j \leq n^2/4$ and $1 \leq k \leq 1$. The length of the i loop is $n/2$, the j loop is $n^2/4$ and the k loop is 1. So this is $n^3/8$ overall and since this is a subset serving as a low bound for the original code, this means the original code is $\Omega(n^3)$.

But combined with $O(n^5)$, this isn't very informative. We ideally wish we could get $\Omega(n^5)$ instead. Did we mess up? Well, let's look at our bounds again for the subset. We had $n/2 \leq i \leq n$ and $1 \leq j \leq n^2/4$ and $1 \leq k \leq 1$. Remember how in our superset we had $\Theta(n^2)$ for the k loop, but here, we have $\Theta(1)$ for the k loop? That seems to be a discrepancy we should address.

But why was our k bound in the subset so awful? Well, it's because the minimum possible value of j forced it to be that way. Because $j = 1$ was a possibility, we have to deal with k going from 1 to 1, which is not much room to work with. And we couldn't extend this bound in a subset, only decrease it. So when we made the subset, the $j = 1$ pulling this k loop down meant all the others like $j = n^2/4$'s k loop had to be pulled down to 1 with it.

Well, we could cut off small j from the lower bound to give more breathing room in the k loop. Let's go back to the step where we determine the j loop: $n/2 \leq i \leq n$ and $1 \leq j \leq n^2/4$ and $1 \leq k \leq j$. Well, how about changing j's bounds so that it is $n/2 \leq i \leq n$ and $n^2/8 \leq j \leq n^2/4$ and $1 \leq k \leq j$. This is still a subset, but we increased the lower bound on j to give more breathing room on the k loop. Now, let's see how the k loop works.

For $j = n^2/8$, we have k going from 1 to $n^2/8$. For $j = n^2/8 + 1$, we have k going from 1 to $n^2/8 + 1$. For $j = n^2/4$, we have k going from 1 to $n^2/4$. Now, we can set the cutoff for the upper bound on k to be $n^2/8$, since for each j we work with, the k loop either stops at $n^2/8$ or goes past it. So now we

have $n/2 \le i \le n$ and $n^2/8 \le j \le n^2/4$ and $1 \le k \le n^2/8$. When computing this, it comes out to $n/2 \cdot n^2/8 \cdot n^2/8 = n^5/128$. This is a subset of the original, so we know the original is $\Omega(n^5)$. Much better!

Warning 8.3. Similar to my warning for superset, for subset, make sure you're never increasing the duration of any iteration, or you'll be including triples in the "subset" that aren't original ones. That would defeat the purpose of a subset.

So the Ω and O helps us conclude that the original code is $\Theta(n^5)$.

Advice 8.4 — I think the hardest superset-subset problem is the first one you see. Once you've seen one, others will start to feel natural but can still be tricky.

Superset is usually easy because you just have each upper bound be the highest it can be.

Subset is tricky, but you'll want to pay attention to the runtimes of each individual loop and use them to model the runtimes of your subset. For example, we have k be $\Theta(n^2)$ in the superset, so we ideally wanted k to be $\Theta(n^2)$ in the subset, too. Hence, why we wanted the length of the k loop to be something like $\frac{n^2}{2}$ or $\frac{n^2}{5}$. Also, we had to carefully position the i and j loops so the k loop works out. This comes with practice and seeing examples.

Here's a practice example that will be easier to reason with now that you have seen one example:

Example 8.5 (Recitation)

Find a Θ bound on the runtime of this code snippet:

```
for i <- 1 to n (incrementing i by 1 each time)
    for j <- i to n (incrementing j by 1 each time)
        for k <- i to j (incrementing k by 1 each time)
            print("1210")
```

Superset is easy: make each loop run the largest it can be. Have i go from 1 to n. Then j goes from 1 to n. Then, k goes from 1 to n. So that's n^3, meaning the original code is $O(n^3)$.

Subset is tricky, but let's think about this carefully. So we have the j loop starting at some value after i and going to n. Then, the k loop runs for values between i and j. We want each loop length to be $\Theta(n)$, just like our superset.

We want the j range to be values all after i, but the k range to be values between i and j. So let's have i go from 1 to $n/3$, then j go from $2n/3$ to n, then k go from $n/3$ to $2n/3$. Convince yourself this is actually a subset with the intuition of decreasing loop bounds. This works out to $\Omega(n^3)$.

So then the original code is $\Theta(n^3)$.

§8.3 Table and Summation Method

Well, in that example before, we had each loop incrementing by 1 each time. But that won't always be the case, as we'll see in an example.

Example 8.6 (Recitation)

Find a Θ bound on the runtime of this code snippet:

```
for (int i = 4; i < n; i = i*i) {
    for (int j = 2; j < sqrt(i); j = 2j) {
        print("1210");
    }
}
```

I'm just going to tell you: the superset-subset method isn't going to work well here when the loop's iteration method isn't increasing by 1 each time but doing wacky stuff instead. So we're going to have to do some math! But before that, let's consider what is going on here.

So we start at $i = 4$. Let's suppose some time passes and the j loop completes, so we need to change i. We change it to $4 \cdot 4$ or $i = 16$, as instructed by the for loop. Then, run the j loop again with this new value of i. Suppose some time passes and the j loop completes again. Then, we change i to $16 \cdot 16 = 256$. So we keep doing this process of running the j loop with a value of i and modifying i with the condition provided. But we stop when $i < n$ no longer holds.

But let's examine the value of i at each iteration number. First iteration, i is $4 = 2^{2^1}$. Second iteration, i is $16 = 2^{2^2}$. Third iteration, i is $256 = 2^{2^3}$. we notice a pattern: in general, during the xth iteration of i, the value of i is 2^{2^x}. How would you notice this? By squaring i each time, we keep multiplying the exponent by 2, so the powers of 2 in the exponent accumulate. Let's make a table to organize our data:

iteration # of i	1	2	3	$\cdots$	x
Value of i	2^{2^1}	2^{2^2}	2^{2^3}	$\cdots$	2^{2^x}

We can also examine the j loop's values as it goes through iterations. First iteration, j is 2. Second iteration, it is $2 \cdot 2 = 4$. Third iteration, it is $2 \cdot 4 = 8$. So for the yth iteration of j, the value of j is 2^y.

iteration # of j	1	2	3	$\cdots$	y
Value of j	2^1	2^2	2^3	$\cdots$	2^y

Just like the last example, it boils down to how many times we print "1210." Print is of constant time, so let's just say it takes time 1. We will convert i and j to x and y because, unlike i and j, we have x and y incrementing by 1 each time. So we can write the runtime in the form

$$\sum_{x=1}^{U} \sum_{y=1}^{V} 1,$$

where U and V represent upper bounds for x and y, respectively. In particular, U is the value of i when it reaches its upper bound and remember that the value of i depends on x. Similarly, V is the value of j when it reaches its upper bound, with the value of j depending on y. This sum is something that is feasible to evaluate, most of the time.

So let's turn the for loops in to sums. Let's suppose we're at the xth iteration for i and then we're running the j loop until it hits its limit. So we stop when $j = \sqrt{i}$ roughly (the upper bound). Remember that $i = 2^{2^x}$, where x represents the iteration number for i. So we do the first iteration of j, then the second, then the next, until we stop. The stopping point is when 2^V (the value of j at the Vth iteration) hits $\sqrt{i} = \sqrt{2^{2^x}}$. We want to find this value of V as that is our upper bound. So we equate: $2^V = \sqrt{2^{2^x}}$ or

$$V = \log_2(\sqrt{2^{2^x}}) = \frac{1}{2}\log_2(2^{2^x}) = \frac{1}{2} \cdot 2^x = 2^{x-1}.$$

The inner loop basically prints as many times until the iteration number hits V, the upper bound, so we can represent this inner loop as

$$\sum_{y=1}^{V} 1 = \sum_{y=1}^{2^{x-1}} 1.$$

So given the value of x, the inner loop runs in that time. But the value of x goes from 1 to 2 to 3 and so on until it too reaches its limit. So we need to figure out when that is. Suppose U is the stopping point for x. Remember the stopping point for i here is when it gets to n. Then, we need 2^{2^U} (the value of i at the Uth iteration) to be equal to n. So

$$2^{2^U} = n \implies 2^U = \log_2(n) \implies U = \log_2(\log_2(n)).$$

So we run x from 1 to U. Combining the inner loop sum with these bounds on x implies the total runtime is

$$\sum_{x=1}^{U} \sum_{y=1}^{V} 1 = \sum_{x=1}^{\log_2(\log_2(n))} \sum_{y=1}^{2^{x-1}} 1.$$

No way around it now, we have to do the math. Inner sum is easy, just add up as many 1s as there are within the bounds for y:

$$\sum_{x=1}^{\log_2(\log_2(n))} \sum_{y=1}^{2^{x-1}} 1 = \sum_{x=1}^{\log_2(\log_2(n))} 2^{x-1}.$$

Now, we shift the index and use sum of powers of 2 formula to get

$$\sum_{x=1}^{\log_2(\log_2(n))} 2^{x-1} = \sum_{x=0}^{\log_2(\log_2(n))-1} 2^x = 2^{\log_2(\log_2(n))} - 1 = \log_2(n) - 1.$$

So the code is $\Theta(\log(n))$. Phew!

> **Advice 8.7 —** Personally, I would always use superset-subset for when each loop is incrementing by 1 each iteration. If any of the loops are anything other than incrementing by 1 each time (like doubling), then do table and summation.

§8.4 Summary

This section was unavoidably tricky and it may take time to sink in. I explained stuff in detail so that you have more intuition on the steps and can be able to tackle similar problems.

- We saw how, for tricky nested for loops, we don't actually have to compute the exact number of how many times it runs. We can instead, find a superset by extending the duration of the loops into something that's easy to calculate. Then, we similarly find a subset by diminishing the duration of the loops.
- In times where the loops don't increment by 1 each time, we have to get in to some gritty math. But what we did though is figure out how the variables behave in each loop to get a general pattern of what their values will be at a certain iteration. Then, we change the sum to be in terms of iteration numbers.
- We also use the stop conditions on each loop to find upper bounds for the sums. In particular, the outer loop's upper bound depends on n. The inner loop's upper bound depends on the value of i, which depends on the value of x (the iteration number for the outer loop).
- Rewriting stuff in terms of x and y instead of i and j gets us something we can sum.

§9 Merge Sort

> **Warning 9.1.** From here on out, my algorithm explanations, proof of correctness, and runtime will be less formal than before. Meaning, the purpose isn't for rigor: it's to get you to understand the idea of how it works. If I broke down every detail to the simplest level, we would be here for much longer than we need to when I can just spoonfeed you the intuition. But do go and try to read the formal proofs in the lecture notes when you have the intuition down as that will improve your understanding of the algorithms. It's just that those formal proofs feel unmotivated and convoluted without fundamental, intuition in Layman's terms first.

§9.1 Introduction

So insertion sort had a worst case time of $\Theta(n^2)$. Not ideal, so can we do better?

First, I need to introduce a sub algorithm called "Merge."

Given two sorted arrays, we want to combine them (merge) to form a single sorted array. Here's an algorithm to do that, where A and B are the original sorted arrays. Let C be the new sorted array that we want.

- If one of the two arrays are empty, take the leftmost element from the non-empty array and add it to the end of C.

- Otherwise, if both arrays are non-empty, compare the leftmost elements of A and B. Whichever is smaller, delete it from its respective array and add it to the end of C.
- Rinse and repeat until both A and B are empty.

Warning 9.2. In Java, you can't actually "delete" elements from the array as easily like I did. I only did so to make it easier to explain. How you would actually implement it is have some sort of pointer that moves to the right in the original arrays as you "delete" elements to know where you are in each array. Anything left of said pointers, you already added to the new array and no longer care about.

Let's do a quick demonstration with sorting these two sorted arrays:

1	2	3	4
8	11	17	30

1	2	3	4
1	2	18	19

We compare the leftmost elements of both. So 1 vs 8, 1 is smaller, so we add 1 to C and delete it from the second array. Compare leftmost elements again: 2 vs 8, 2 is smaller, so we add 2 to C and delete it from the second array. Now, 8 vs 18, 8 is smaller, so we add 8 to C and delete it from the first array. Repeat this process until we have all elements in C and it looks like this:

1	2	3	4	5	6	7	8
1	2	8	11	17	18	19	30

The proof of correctness for merge basically relies on the fact that both arrays are sorted. Anything to the right of the leftmost element is bigger than it. So the smallest element among both A and B has to be the leftmost element of either A and B; anything else is necessarily greater than the respective leftmost elements, so thus, is not a contender for smallest element among both arrays. So using this fact, we can make a loop invariant where every step of the way, we choose the smallest element among both arrays and maintain order in the progress for C. I don't need to spell out the full formal details but rather the full intuition.

Runtime? So transferring a single element from either of the two arrays to C is constant time: doing some comparison stuff, "deleting" it from the respective array, and moving it to C. We make as many transfers as there are elements in both arrays. So this is $\Theta(n+m)$, where n and m are size of A and B, respectively.

But merge only works for two sorted arrays! Well, what if we could recursively sort arrays and merge them into a sorted array? Let's see how that works in merge sort!

§9.2 The Algorithm

So we want to take an unsorted array as an input and the sorted version of it as an output. Here's the algorithm:

Base Case: If the array is of size 1, we just return it.
Otherwise, divide the array in to two halves:

- Recursively call merge sort on the left half of the array
- Recursively call merge sort on the right half of the array
- Merge the two sorted halves

§9.3 In Action

I'll skip all the details involving merge, as my focus is more on the recursive action going on. Let's see how merge sort sorts this array:

1	2	3	4	5	6	7	8
8	10	3	7	1	2	11	4

- So we divide into two halves: $8, 10, 3, 7$ and $1, 2, 11, 4$. We recurse on the first half. We'll get to the second half later.
- So now, we divide the half into halves: $8, 10$ and $3, 7$. We recurse on the first half. Again, we'll get to the second half later.
- So now, we have 8 and 10, individually. We hit our base case for both. Now, we start working our way up. We merge 8 and 10, which gets us $8, 10$.
- When we broke $8, 10, 3, 7$ into two halves, we now have the recursive work done for the first half. We just need to do the second half: $3, 7$.
- We break $3, 7$ down, hit the base case, and merge them to $3, 7$.
- So now, we merge $8, 10$ and $3, 7$ (the two parts we got recursively). This comes to $3, 7, 8, 10$.
- When we broke the original array in half, we now have the recursive work done for the first half. Now, we have to do the recursive work for the second half. I'll spare you the details and say the second half works out to $1, 2, 4, 11$.
- So now, we did the recursive work on both halves of the original array and got $3, 7, 8, 10$ and $1, 2, 4, 11$. Now, we merge them to get $1, 2, 3, 4, 7, 8, 10, 11$.

§9.4 Proof of Correctness

This is basically the same strong induction idea back from binary search. To give the intuition, obviously an array of size 1 is sorted, so that's the base case.

The inductive step comes from dividing the array in to two halves and recursively sorting them (because the halves are arrays of size less than n, the strong inductive hypothesis means both will be sorted correctly). So the two halves are sorted after the recursive calls. We know merge works on two sorted arrays, so we get the array of size n sorted.

§9.5 Runtime

So let $T(n)$ be the time for merge sort. We first check if we're in the base case, which is of constant time. Otherwise, we recurse on both halves of the array, which means we solve two subproblems of size $\frac{n}{2}$. Then, we merge, which is of time $\Theta(n)$, as reasoned by our runtime analysis for merge. The constant time check for the base case doesn't matter too much when we have a $\Theta(n)$. So we say

$$T(n) = 2T(\tfrac{n}{2}) + \Theta(n).$$

By Master Theorem, this is $\Theta(n \log n)$. No need to go through the trouble of expanding it when they don't require us to do so. Since $\log n$ is better than n, this is better than $\Theta(n^2)$, which is insertion sort's worst case. So we found a more reliable sorting algorithm in terms of time!

> **Warning 9.3.** Don't get the idea that insertion sort is obsolete just because merge sort has a better runtime than it. We'll see later that merge sort has flaws of its own.

§9.6 Summary

- We introduced a sub-algorithm called merge, which takes two sorted arrays and merges them into a single sorted array.
- We use merge to create a recursive sorting algorithm. Here, the array is divided in to two halves which are recursively sorted. Then, the two halves are combined with merge into a single sorted array.
- Merge sort's time of $n \log n$ is better than insertion sort's worst case time of n^2.

§10 Divide and Conquer

§10.1 Introduction

So we know how merge sort works, right? We divide the problem in to two subproblems, then recursion solves the two subproblems, and then we combine the two solved subproblems. This is called a divide and conquer algorithm. It has 3 main steps:

1. Divide: This is dividing the problem into multiple subproblems (usually 2). In merge sort, this is when we divided the array in to halves.
2. Conquer: This is recursively solving each subproblem. In merge sort, this is when we made recursive calls to both halves of the array.
3. Combine: This is combining the solved parts. In merge sort, this is merging the two sorted arrays.

Note that not all divide and conquer algorithms have a combine step. For example, with binary search, after some work, the recursive call was simply going to one of the two halves of the array, completely disregarding the other half.

Advice 10.1 — In that regard, whenever you have a problem involving "find an instance of something in an array" and they want $O(\log n)$, that's usually going to be something along the lines of binary search: you want to check the middle and use that information to conclude you only care about one side of the array and recurse on that, while completing ignoring the other side. I wouldn't say this is always the case though.

Advice 10.2 — When the problem wants $O(n \log n)$, recall Merge Sort's recursion of $T(n) = 2T(n/2) + \Theta(n)$. So you can interpret this as having to recurse on both sides of the array and having $\Theta(n)$ work in combining the two solved portions. This isn't always the case, but may be a good starting point.

Advice 10.3 — Alternatively, when it is $O(n \log n)$, you can use Merge Sort once on the array without exceeding the time. The problem may be much easier when the array is sorted, so if you realize that's the case, do it.

Advice 10.4 — Use Master Theorem to verify your runtime for divide and conquer algorithms! Saves you time from expanding.

Warning 10.5. Remember that when you make a recursive call to one side of the array, you no longer have access to the other side of the array within the recursive work! An example would be if we needed to work on the subproblem provided by the left half of the array but for some reason in the subproblem, we needed the last value in the original array. We wouldn't be able to get that last value directly when working on the subproblem, so we'd need to use a parameter to pass the important information along recursive calls in that situation. After the subproblem is over, you get access to the whole array again, like when we did the merge after solving the two subproblems with merge sort.

§10.2 Problem Solving

Example 10.6 (Recitation)

You are given an integer array $A[1..n]$ with $n \geq 3$ and the following properties:

- Integers in adjacent positions are different
- $A[1] < A[2]$
- $A[n-1] > A[n]$

A position i is referred to as a local maximum if $A[i-1] < A[i]$ and $A[i] > A[i+1]$.
Example: You have an array $[0, 1, 5, 3, 6, 3, 2]$. There are multiple local maxes at 5 and 6.
Design an $O(\log n)$ algorithm to find a local maximum and return its index.

Let's understand the problem first. I propose to think of the array as a sequence of length $n-1$, where the sequence consists of up arrows or down arrows.

Have an up arrow at the ith position in the sequence when $A[i] < A[i+1]$ (modelling an increase) and a down arrow at the ith position in the sequence when $A[i] > A[i+1]$. So for the example array $[0, 1, 5, 3, 6, 3, 2]$, the arrow sequence would be $\uparrow\uparrow\downarrow\uparrow\downarrow\downarrow$. We don't have time to determine all the arrows because that would require $O(n)$ work, worse than the target runtime of $O(\log n)$; it's merely an easier way to think about the problem.

We're given $A[1] < A[2]$, meaning the first arrow in the sequence is $\uparrow$. We're also given $A[n-1] < A[n]$, meaning the last arrow in the sequence is $\downarrow$.

A local maximum occurs when we have $\uparrow$ then $\downarrow$, consecutively in that order.

We know the arrow sequence starts with $\uparrow$ and ends with $\downarrow$. So there MUST be some point where the sequence transitions from $\uparrow$ to $\downarrow$; it's inevitable considering the arrows at the endpoints of the sequence. But now, we have to figure out where. So the fact that a sequence starting with $\uparrow$ and ending with $\downarrow$ must have a transition point from $\uparrow$ to $\downarrow$, keep that fact in mind because it will be essential for our proof of correctness.

Also, remember we don't need to find all local maximums, just one. Let's determine the arrow in the middle. We can do this by checking two integers in the middle of the array at consecutive positions.

Let's say the middle arrow is $\uparrow$. So we know the arrow sequence is going to be $\uparrow$ (first arrow), something, $\uparrow$ (in the middle), something, $\downarrow$ (last arrow).

Remember how every sequence starting with $\uparrow$ and ending with $\downarrow$ must have a transition point? So we know the second half of the array has a local maximum considering what we know about the arrows. So we want to search there. As for the first half, well, there COULD be a transition point. But for all we know, it could just all be up arrows, so there is no guarantee we find a transition point there. So we don't want to bother searching that half of the array and focus on the half that we know has to have a transition point.

But what about when the middle arrow is $\downarrow$? By similar reasoning, we want to search the first half and not bother with the second half.

As for the base case, that's when $n = 3$ and we have two arrows. Since the first arrow is $\uparrow$ and the second (which is the last) arrow is $\downarrow$, we have a transition point in front of us.

And there we have it, that's our algorithm.

Proof of correctness? Same old strong induction that we did for binary search. Base case is $n = 3$. Then, the inductive step relies on that fact where if the sequence starts with $\uparrow$ and ends with $\downarrow$, there's a transition point somewhere. Then, the two cases on the middle arrow, along with the recursion with the inductive hypothesis means it works out.

Runtime? We do constant amount of work to get the middle arrow and use that information to then solve 1 subproblem of size $T(n/2)$. Base case is obviously constant work, so we have $T(3) = k$ and $T(n) = c + T(n/2)$ for $n > 3$. Master Theorem implies $T(n)$ is $O(\log n)$. Alternatively, notice this is binary search's runtime.

Example 10.7 (Recitation)

You are given an integer array, with both positive and negative elements. Design an $O(n \log n)$ algorithm to return the sum of the continuous subarray with the maximum sum.

This is a hard problem, but let's think about it. In general, the largest continuous subarray is either going to be in one of three possibilities:

- It's entirely within the left half of the array
- It's entirely within the right half of the array
- It's partly in the left half and partly in the right half

Partly in the left half and partly in the right half would be something like this:

1	2	3	4	5	6
0	−1	5	11	4	−2

Anyway, we know that the largest continuous subarray is in one of those three categories. Let's find the largest for each respective category and compare the 3 at the end. The first two, we can get by recursively solving the problem on the two halves of the array, separately.

As for the subarray spanning on the middle? Not as easy to see, but we can cut a subarray spanning on the middle through the middle. So $A[3..5]$ in the example can be split in to $A[3..3]$ and $A[4..5]$. Not to mention the left half of any middle array must have the middle itself (3, in this case). The right half of any middle array must have the middle itself too (4, in this case).

So here's what we do to get the best subarray spanning on the middle. Consider the sum for $A[n/2, n/2]$, then $A[n/2-1, n/2]$, then $A[n/2-2, n/2]$, and so on until $A[1, n/2]$. Basically we check each subarray that starts at the middle and protrudes leftward. Keep track of the best sum we find as we go through them. So we take the best one, say $A[i, n/2]$.

We also do the same for the subarrays that start at the middle and protrude rightward: check $A[n/2+1, n/2+1]$, then $A[n/2+1, n/2+2]$, then $A[n/2+1, n/2+3]$, and so on until $A[n/2+1, n]$. Again, keep track of the best one of these and say the best one is $A[n/2, j]$.

We combine the best one that starts at the middle and protrudes leftward (which we called $A[i, n/2]$) with the best one that protrudes rightward (which we called $A[n/2, j]$) and combine the two to get $A[i, j]$, which necessarily passes through the middle of the array.

So we found the best subarrays for the three categories, we just compare the 3 to find the overall maximum.

As for the base case, that's when we have 1 element. Either we include it in the subarray, or we don't. If it's negative, don't include it. If it's positive, we include it.

Proof of correctness? Base case is pretty self-explanatory on why it works. Don't include an unnecessary negative that brings the sum down, but definitely include a sole positive.

As for inductive step, we again know the best subarray is in one of the three categories. The recursive call and inductive hypothesis correctly gets us the best in the first two categories. As for proving we get the best in the third category, well, there's a part that protrudes leftward from the middle (call this part L) and there's a part that protrudes rightward from the middle (call this part R). Clearly, L has no effect on R, so they're independent. To optimize the overall middle array, both L and R should be optimal. And we optimize it by going through all the possibilities and cherry picking the best one.

So we correctly obtain the best from all three categories and all that's left is to compare them. This proves the inductive step.

Runtime? Well, when we get the best from both halves, that's two subproblems of size $\frac{n}{2}$. As for the best subarray spanning the middle, we check $n/2$ different subarrays protruding leftward and $n/2$ different subarrays protruding rightward. We can save time on sums by reusing previous work: the sum for $A[m-1, n/2-1]$ for example is the sum for $A[m, n/2-1]$ plus $A[m-1]$. So we're really just adding $n/2$

numbers together and doing some constant work in checking if we found a new best when going through the leftward protruding parts. Same with the rightward protruding parts.

So finding the best subarray spanning the middle is $\Theta(n)$ work. So we have $T(n) = 2T(\frac{n}{2}) + \Theta(n)$. This is merge sort's time, which is $O(n \log n)$.

§10.3 Summary

- We've seen before this section how algorithms like binary search and merge sort break the problem in to parts that progressively get easier to solve.
- Divide and conquer captures the general strategy behind those two algorithms. We divide the problem in to subproblems and then recursively solve those subproblems. Sometimes, we may combine the solved subproblems.
- We saw a problem where we only cared about one side of the array, so we didn't have a combine step.
- We also saw a problem where we needed to solve the subproblems provided by both sides of the array and combine them.

§11 Quick Sort

§11.1 Introduction

So Merge Sort's $n \log n$ runtime is cool and all. But there's an issue with it. It'll take up a lot of memory. Think about it: you have a problem of size n. You break that up in to two subproblems of size $n/2$ and need to allocate memory for the arrays corresponding to the would-be results of these subproblems. Then you break all of those in to two subproblems, each of size $n/4$ and need to allocate separate memory for all these subproblems. And so on.

The issue is that these subproblems don't get fully resolved until we break the subproblems into a small enough size that we reach a base case. So in breaking the problems into subproblems constantly, we use up a lot of memory, especially for a massive value of n.

Maybe we can try to get a sorting algorithm in $n \log n$ time that is in-place? Meaning we don't have to create these additional arrays when making recursive calls and we just do all the swapping and action in our original array?

§11.2 Main Idea

Here's a demonstration. We have this array. I'm going to choose an element at random from this array to be our "pivot," say 7 for this demonstration:

1	2	3	4	5	6	7
6	19	3	10	7	11	8

We want to switch the pivot with the last element in the array just to get it out of the way for now. Then, we consider the stretch of the remaining numbers. We label the left end with an L pointer and the right end with a right pointer.

L					R	
6	19	3	10	8	11	7

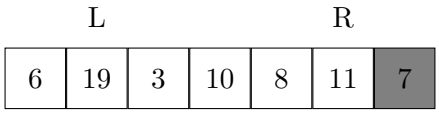
We're going to keep running this loop, until the L is directly to the right of R:

- Check the element where the L pointer is

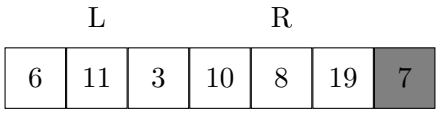
- If the element is less than the pivot, move the L pointer 1 slot to the right
- If the element is greater than the pivot, swap the numbers at the L and R pointers and then move the R pointer 1 slot to the left.

When the above loop ends, the L and R pointers will have crossed. Insert the pivot in between the crossed pointers.

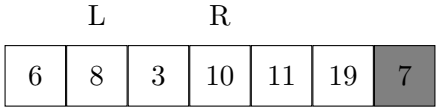
Let's work through the example. 6 is less than the pivot, just move the L pointer to the right:



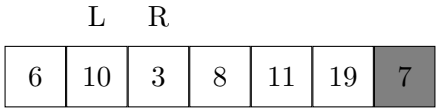
19 is greater than the pivot, swap 19 and 11 and move the R pointer to the left:



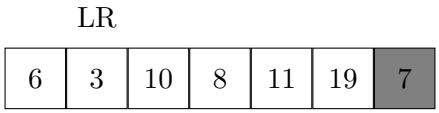
11 is greater than the pivot, so swap and move R to the left:



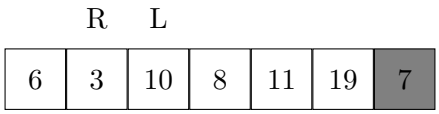
Swap again:



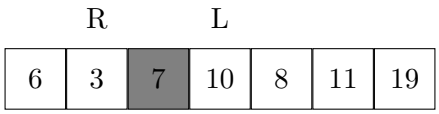
and again:



3 is less than the pivot, so we just move L to the right:



The two pointers have crossed, so we insert 7 back in between them:



The essential gist is we pick a pivot element. Then, we get everything less than the pivot to the left of the pivot in the array and everything greater than the pivot to the right of the pivot in the array.

The L and R pointers are a way to monitor our progress. Through a loop invariant, you can show anything strictly to the left of the L pointer is less than the pivot at every step. Similarly, anything strictly to the right of the R pointer is greater than or equal to the pivot at every step. Once the two pointers cross, you know you have everything less than the pivot on the left and everything greater than the pivot on the right, so you insert the pivot in that location.

I claim the pivot is now in the correct spot in the array and this would be the case for any choice of an array and pivot. Why? Because we already identified everything less than the pivot and greater than the pivot. If the array were in order, going left would mean smaller numbers and going right would mean greater numbers. So in the final sorted array, everything less than the pivot is left of the pivot and everything greater than the pivot is right of the pivot. So the pivot doesn't need to even move from this spot, and it boils down to sorting the two halves the pivot divides the array in to.

So we can use this idea in to an algorithm called Quick Sort. So we choose a pivot and partition the array about the pivot. This means we get everything less than the pivot to the left of it and everything greater than the pivot to the right of it, like we did in the example. Then, the pivot divides the array in to two halves, where we recursively solve each half. Eventually, every element will be a pivot and thus, get put in the right place. The base case would be if we have 1 element in the array and it would be already sorted, so we do nothing.

The neat thing about this algorithm is, it is in-place, meaning we didn't have to create any additional arrays through recursive calls: we do all the sorting within the original array. This means quick sort doesn't require much additional memory beyond the original array.

There are two types of quick sort:

- Deterministic: We pick the pivot based on some rule we decide on (like always the first element in the array, or always the middle)
- Randomized: We pick the pivot at random.

In proof of correctness and runtime though, there is no difference between deterministic and randomized. Because the first element being a pivot is equally likely to be as good or bad as the second element, or the third element, or the whatever element, considering the random nature of an arbitrary array.

§11.3 Proof of Correctness

Same old strong induction. Base case of 1 element obviously works. When we use the pivot to partition the array, we reasoned earlier that everything to the left of it is less than the pivot and everything to the right of it is greater than the pivot. From that fact, we concluded the pivot is in the correct position in the array and that it boils down to recursively sorting the two halves of the array. By IH, the two halves will get recursively sorted.

§11.4 Runtime

> **Review 11.1.** We're going to use random variables and Bernoulli trials here, along with LoE. Go back and review that from CIS 1600 if you need to.

Okay, so we can all agree, quick sort, in essence, is just a bunch of comparisons and swapping. One comparison with the additional possibility that a swap follows it, is obviously constant time. So it boils down to how many comparisons we do.

Best case generally isn't very informative, as it doesn't really tell us how an algorithm may function in general. Instead, let's look at expected runtime.

To that end, let $y_1 < y_2 < \cdots < y_n$ be the n elements of the array in increasing order. Keep in mind, the originally array is, more often than not, not going to be sorted already.

For integers i and j with $1 \leq i < j \leq n$, let $X_{i,j}$ denote a random variable on how many times y_i is compared to y_j throughout the entire process. Note that we don't have $i = j$ because it doesn't make

sense to compare an element to itself in the context of quick sort. And $X_{j,i}$ is the same as $X_{i,j}$, so we do $i < j$. Anyway, the runtime is

$$T(n) = \sum_{1 \le i < j \le n} X_{i,j}$$

because this sum takes every pair of elements in the array, counts the number of times the two are compared, and contributes it to the total to get an overall count on the number of comparisons. Taking the expected value of both sides and using LoE (remember LoE allows us to separate expected value over sums!) implies

$$E[T(n)] = E\left[\sum_{1 \le i < j \le n} X_{i,j}\right] = \sum_{1 \le i < j \le n} E[X_{i,j}].$$

So now our goal is to find $E[X_{i,j}]$.

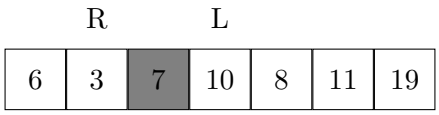
Remember how quick sort works? Whenever things are being compared, the pivot element is always involved. So a comparison between y_i and y_j would mean one of those two elements is currently the pivot.

I claim that if y_i and y_j are compared once, they'll never be compared again, meaning $X_{i,j}$ is at most 1. Why is this? Let's just say y_i is the pivot (the case of y_j being the pivot reasons similarly). We compare y_j to y_i (the pivot) inevitably and then move y_j to the appropriate half of the array. But then, after all comparisons with y_i are done, we recurse on both halves of the array formed by y_i. And when we recurse on the half containing y_j, we completely ignore y_i, as it is no longer part of either half we recurse on. So once we compare y_i and y_j once, we'll never do it again because of the recursion.

Okay, so $X_{i,j}$ is either 0 or 1, so it's essentially a Bernoulli random variable. Also, let $F_{i,j}$ be the event that $X_{i,j} = 1$ occurs. Now, it boils down to

$$E[T(n)] = \sum_{1 \le i < j \le n} E[X_{i,j}] = \sum_{1 \le i < j \le n} \Pr[F_{i,j}].$$

But also, because we recurse on both halves of the arrays individually, the two sides of the array won't interact with each other at all during the recursive work. Let's take that finished example for demonstration:



So we partitioned everything about 7 and recurse on both halves. That means 3 cannot interact with 8 anymore because they're on opposite halves. Same with 6 and 11 and every pair of numbers on opposite halves. Can we use this idea to get a better understanding of when $F_{i,j}$ occurs?

So suppose we have k so that $i < k < j$. If y_k is a pivot element before both y_i and y_j become one, what will happen is we partition the array about y_k and the order means y_i and y_j end up on opposite halves. Then, the recursion ensures we no longer have the chance to compare y_i and y_j to each other (and we don't need to because of the purpose of a partition).

However, if we have $k < i < j$, y_k being the pivot doesn't mean we permanently lose the chance to compare y_i and y_j because then, both y_i and y_j end up on the same side of the partition. Same with $i < j < k$.

Okay, think about CIS 1600 probabilities. So for $F_{i,j}$ to occur, y_i or y_j must be selected as a pivot before any y_k with $i < k < j$ is selected as a pivot. So there are $j - i + 1$ possible pivot elements in question here: y_i, y_{i+1}, y_{i+2}, $\cdots$, y_j. Each one is uniformly likely to be the pivot first, since we select pivots arbitrarily. Only when y_i or y_j is the first of those possible pivots, will it allow $F_{i,j}$ to occur, only 2 possibilities of those choices. So $\Pr[F_{i,j}] = \frac{2}{j-i+1}$. Now, for some gritty math. Note that summing over all pairs (i, j) with $1 \le i < j \le n$ is basically summing over all pairs with $1 \le i \le n-1$ and $i+1 \le j \le n$ (think of it like a for loop if it's hard to see).

$$E[T(n)] = \sum_{1 \le i < j \le n} \Pr[F_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \left(\frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n-i+1}\right).$$

Those who haven't taken CIS 1600 with Rajiv may not know this, but $\sum_{a=1}^{b} \frac{1}{a}$ is basically $\log b$ (the Harmonic series). The stuff inside the parantheses is basically double a harmonic series up to $n-i+1$, so

$$E[T(n)] = \sum_{i=1}^{n-1} \left(\frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n-i+1} \right) = \sum_{i=1}^{n-1} (\log(n-i+1))$$

$$= \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(2) = \log(n!) = \Theta(n \log n),$$

where we used a recitation problem saying $\log(n!) = \Theta(n \log n)$. That was a lot of work, but the takeaway is Quick Sort is expected $\Theta(n \log n)$. But that's only **expected**. What about worst case?

Well, it turns out the worst case is when the pivot is the largest or smallest element in the array. What will happen is we make all the $n-1$ comparisons between the pivot and all other elements and move stuff around. But all the elements will be in one side of the pivot. Then, we recurse on that side, which is an array of length $n-1$. So in essence, for the worst case, we have $T(n) = (n-1) + T(n-1)$. Base case is $T(1)$ is a constant. Expanding this gives

$$T(n) = (n-1) + (n-2) + \cdots + 2 + 1 + T(1) = \frac{(n-1)n}{2} + T(1).$$

This is $\Theta(n^2)$, which is insertion sort's worst case time. Okay, so the nice expected runtime comes with the issue of having a not so cool worst case time.

§11.5 Summary

- We saw how quick sort picks pivots at random. By moving elements around, we can get everything less than the pivot on the left and everything greater than the pivot on the right. This puts the pivot in the right spot in the array.
- We recurse on both sides to sort the halves to the left and right of the pivot.
- Quick sort is in-place, meaning we do all the action in the original array without much additional memory necessary.
- We found the expected runtime is $\Theta(n \log n)$, but the worst case is $\Theta(n^2)$.

§12 Selection

§12.1 Introduction

Suppose we want to find the smallest element in an array. Oh that's easy, just go through the array, keep track of the smallest we have encountered so far and return it at the end. Takes $O(n)$ time.

2nd smallest? I guess, so through the array once and find the smallest and get rid of it. Then, the 2nd smallest is the new smallest. Takes two $O(n)$ passes, so still $O(n)$.

What about the median? This is basically the $\frac{n}{2}$th smallest. Well, first pass, we find the smallest among n elements. Second pass, we find the smallest among $n-1$ elements. Third pass, we find the smallest among $n-2$ elements. And so on until we have to find the smallest among $\frac{n}{2}$ elements. How long will that take? Well, we have $\frac{n}{2}$ passes, and for each pass, we work with at least $\frac{n}{2}$ elements in the array, so that's already $\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$, which is $\Omega(n^2)$.

If we have to do that, we might as well merge sort, which takes $\Theta(n \log n)$ and get the median that way. But can we do better? Can we get the ith smallest element (for any i with $1 \leq i \leq n$) in $O(n)$ time? That's selection.

§12.2 The Algorithm

This is to find the ith smallest element in the array.

- Divide all n elements into groups of 5. Excess elements can go in their own group.
- Find the median element in each group of 5.

- Take the group consisting of the median element of each group. Find the median M of that group by recursively calling the algorithm.
- Partition the entire array about M like in quick sort.
- Let p be the resulting position number of M. If $i = p$, just return M. If $i < p$, recurse on the left half of the array, finding the ith smallest element. If $p < i$, recurse on the right half of the array, finding the $(i - p)$th smallest element.

Base case is when we have $n = 1$ and we just return whatever is the lone element in the array.

§12.3 Proof of Correctness

Why we go about a convoluted way in finding M to choose as a pivot isn't particularly important for proof of correctness. Moreso for runtime. Anyway, we know how partition works back from quick sort and the useful property of it: it gets everything in the array less than the pivot to the left of the pivot and everything greater than the pivot to the right of the pivot. The pivot is also in the correct spot if the array were to be sorted in order.

So if $i = p$, we know M, the pivot, is what we want because it's the pth smallest element by how a pivot works.

Otherwise, if $i < p$, we know the ith smallest element is less than the pth smallest element. By the pivot properties, that's to the left of the pivot, which is why recurse there.

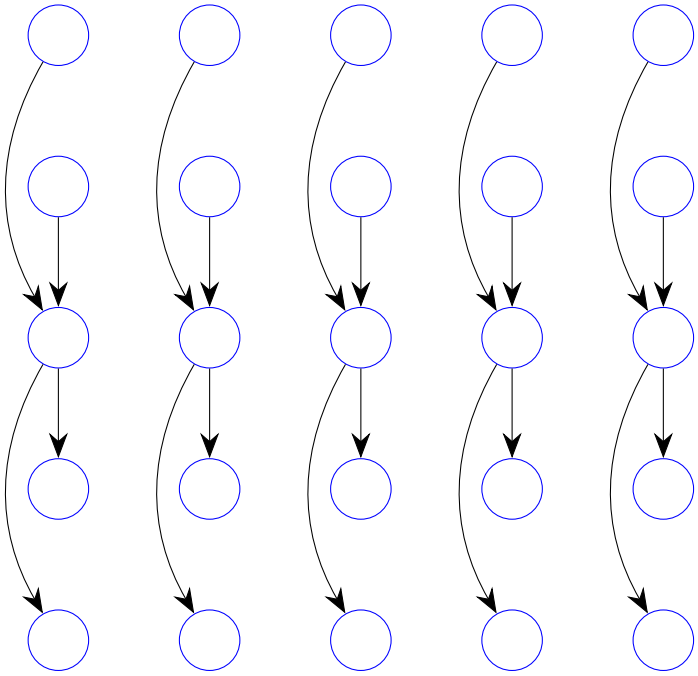
Otherwise, if $i > p$, we know the ith smallest element is greater than the pth smallest element. By the pivot properties, that's to the right of the pivot, which is why recurse on it. Except one thing. We completely lose access to the left side of the array during the recursion. So the ith smallest element in the whole array might not be the ith smallest element in the right half (with the left half deleted). But we know the pivot and everything to the left of it is less than anything in the right half, so deleting that turns the ith smallest element in the whole array to the $(i - p)$th smallest element in the right half.

So that justifies the recursive step, which is our inductive step. Base case is easy, so that's the strong induction needed for recursive algorithms.

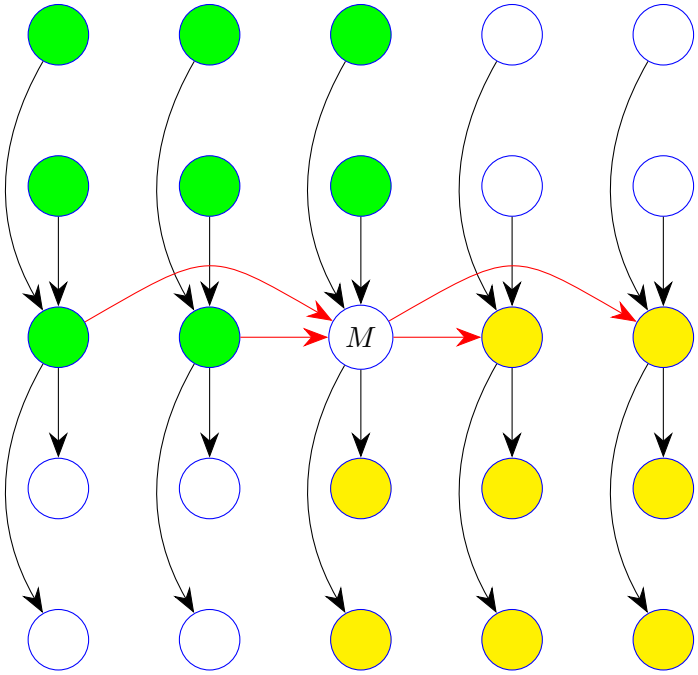
§12.4 Runtime

Let M be the median of the medians that we found as instructed in the algorithm.

Let's say we have 5 groups for demonstration without any leftover (leftover elements are negligible for runtime). In practice, we're going to have $\frac{n}{5}$ groups. I'll have groups represent columns. We can reorganize each column, so elements less than the median in its group are above the median in its column and elements greater than the median are below the median in its column. An arrow from one element a to another element b symbolizes $a < b$.



So the medians occupy the middle row. We figure out the median of the medians (which we called M). We can reorganize the columns so medians smaller than M are to the left of it and medians bigger than M are to the right of it. And we can draw arrows again.



Notice something about those elements I colored in yellow? Yes, we can get to all of them from M through a series of arrows, meaning these are all elements greater than M. How many of these are there? Well, they only show up for columns to the right of M and we have $\frac{n}{5}$ columns (corresponding to the number of groups). M being the median means it's at the halfway point, so $\frac{n}{10}$ columns contain these yellow elements. Each column that contains a yellow element has 3 of them (the 3 bottommost rows). So that's about $\frac{3n}{10}$ yellow elements, or elements we know are DEFINITELY larger than M.

Similarly, green elements are elements that are DEFINITELY smaller than M because you can follow a series of arrows to get to M. Similar logic means there are about $\frac{3n}{10}$ of these green elements.

So when choose M as a pivot, we know it has at least $\frac{3n}{10}$ elements larger than it and at least $\frac{3n}{10}$ elements smaller than it (by the green element and yellow element argument). So that means, worst case scenario, when we partition and do the pivot stuff, each side of the array can have at most $n - \frac{3n}{10} = \frac{7n}{10}$ elements (because the other side has to have at least $\frac{3n}{10}$ elements). So when we do the recursion, worst case, we solve a subproblem of size $\frac{7n}{10}$.

Enough blabbering, let's analyze what contributes to the time function $T(n)$:

- We find the median in each group of 5. Finding a median in a group of 5 is $O(1)$ because it doesn't depend on any variable. But we have $\frac{n}{5}$ groups, so this is $O(\frac{n}{5})$.
- We find the median of the medians, which is a subproblem of size $\frac{n}{5}$ because we have a median from each of the groups. This is $T(\frac{n}{5})$.
- We partition everything about the median of medians, which is $O(n)$ (check each element in the array and put it on the appropriate side in relation to the median of medians).
- The array is in two halves and we know the size of either side doesn't exceed $\frac{7n}{10}$. We recurse on one half, and worst case, it is $T(\frac{7n}{10})$.

So our runtime recurrence is $T(n) = O(\frac{n}{5}) + T(\frac{n}{5}) + O(n) + T(\frac{7n}{10})$. Solving this recursion formally is more trouble than it is worth and not particularly relevant. So I'll tell you $T(n)$ is $O(n)$.

Advice 12.1 — So we can get the median of an unsorted array in $O(n)$ time. This can be useful for partitioning when you don't care about the order within the partition.

Say like you wanted to identify all elements less than the median and all elements greater than the median and put them in separate piles. You can do just that by identifying the median in $O(n)$ time and the partition work gives you the two piles.

§12.5 Summary

- We took the partition intuition back from quick sort and made an algorithm that gets us the ith smallest element in an array in $O(n)$ time
- We chose the median of the medians (we called this M) as our pivot.
- It turns out M has the property that at least $\frac{3n}{10}$ elements are bigger than it and at least $\frac{3n}{10}$ are smaller than it. This limits the size of the recursive subproblem we need to solve.

§13 Lists as a Data Structure

§13.1 Introduction

Review 13.1. You can review the Oracle JavaDocs for linked lists and array lists if you're rusty on them.

Back from CIS 1200, we used linked lists and array lists as essentially, resizable arrays. But now, let's not rely so much on Java's implementation of them and try to implement them from scratch, so we get a better grasp of runtime and space.

So a list as we know, is a finite sequence of ordered elements. In particular, the order aspect means each element has a position.

The list abstract data type needs to support the following operations:

- Append (add something to the back)
- Insert
- Delete
- Access

The array list and linked list both happen to be two implementations of a list.

§13.2 Operations

We know unlike an array, an array list can change in size even after it is declared. But how would we implement this? We would inevitably have to use an array, but move everything to a bigger array when the array fills up. That allows us to resize the array list.

So suppose we had this array list:

1	2	3	4
2	5	10	1

All's well and good until we want to add something else. So if we want to add 15, we have to copy everything to a bigger array and add 15 to the next empty slot:

1	2	3	4	5	6	7	8
2	5	10	1	15			

This resizing is going to be annoying, right? Well, we can do our resizings strategically.

If we resized from 4 to 5, we would have to resize again if we wanted to add another element. Having to resize every time we want to add something new to the array would be time consuming. If we resized from 4 to 1000, we won't have to resize again for a while. But that's a lot of space we're wasting on empty cells of the array that we might not ever use. So there has to be a balance between resizing large enough so that resizings don't happen too often (which saves time) but not resize to a ridiculously big array that wastes memory.

Anyway, let's think about runtime of adding a single element in an array list. If we don't have to resize, it's just $O(1)$. But when we do, it's $O(n)$ (where n is the current number of elements we have) because we have to copy n elements over to a new array and then add it. But resizing only happens every so often. So saying "adding an element in an array list is $O(n)$ worst case" omits the fact that the worst case only happens every so often. So we need a way to analyze runtime in a way that is more informative for this situation.

§13.3 Intro to Amortized Analysis

Definition 13.2 (Amortized Runtime)**.** A way of analyzing runtime of a single operation by doing n operations and averaging them. This helps get around when there is a worst case but it happens every so often and we know when and how often it happens.

It helps to give an example outside the context of computer science to understand this. Suppose I'm paying rent. I pay \$1 on every day except Sunday, but on Sunday, I pay \$22. Worst case, I pay \$22 one day, but that's only once every 7 days. Thinking about the average in the grand scheme of things over the week, it is

$$\frac{1+1+1+1+1+1+22}{7} = 4,$$

as my amortized pay because 6 times a week, I'm paying \$1 and on 1 day, I'm paying \$22. So looking solely at the worst case doesn't give us the big picture because that alone omits the fact that I'm not really paying anywhere as much on the other days, so the average in the long run is much lower than what the worst case may suggest.

Warning 13.3. Do NOT think amortized and expected runtime are the same thing! The key difference is for amortized, we usually know EXACTLY when the worst case happens AND that it doesn't involve any randomness, like with my rent example.

For expected runtime, it usually describes an algorithm that could be close to the best case today and then something close to the worst case tomorrow. Or vice versa. Or any sort of randomness. Who knows!

Anyway, let's look at an amortized analysis of the append operation to an array list. Suppose our rule for resizing is we start at an array of size 1 and double the size each time we need to resize. So it goes from 1 to 2, then 2 to 4, then 4 to 8, and so on.

Let's see how much time appending k elements to an initially empty array costs. Two things contribute to time: putting the element in the array when we have the space and resizing when we need to.

Well, putting an element to the array when we have the space is just $O(1)$, so simply adding the elements is $O(k)$.

And let's see how much time resizing adds. So we resize at 1, meaning we have to copy 1 element at that time. We resize at 2, where we copy 2 elements. And at 4, where we copy 4 elements. And so on. The last time we resize is at $2^{\log_2(k)}$. So adding all instances of having to copy elements over, we have

$$\sum_{i=0}^{\log_2(k)} 2^i = 2^{\log_2(k)+1} - 1 = 2k - 1.$$

This combined with the time it takes to put the k elements means the k append operations are $O(k)$. So the average is $O(1)$, so a single append operation is amortized $O(1)$.

§13.4 Runtime Analysis for Lists

Let's think about the other operations for an array list. Insert is worst case $O(n)$. Even with the possibility with resizing, what if we want to insert something at the front? We'd have to manually shift all the other n elements down. Remove? Same idea. Access? Luckily, that's $O(1)$ because it's an array.

As for linked list, we can think of this as a series of nodes. There is the pointer to the head, where we start. Each node has data of its own (like the entries in an array) and a next pointer directing us to the next node in the list. When we reached the end of the list, the last node's next pointer is simply set to null. Here's a simple linked list of size 5:

For now, assume we can only start a traversal of the linked list at the head. So if we need to access the end of it, we have to follow all the way through and can't skip to the end. Let's look at the operations.

Appending is $O(n)$, since we need to go all the way to the back. Insert is worst case $O(n)$ because we might need to go all the way to the back. Access is worst case $O(n)$ because again, we might need to go all the way back. And remove is worst case $O(n)$ for the same reason.

Operation	Append	Insert	Access	Remove
Array List	amortized $O(1)$	worst case $O(n)$	$O(1)$	worst case $O(n)$
Linked List	$O(n)$	worst case $O(n)$	worst case $O(n)$	worst case $O(n)$

It looks like an array list is better than a linked list in most things in terms of time. But that doesn't necessarily render a linked list as obsolete because we need to analyze space.

§13.5 Space Analysis

Let's just say all elements in question are of the same size, whether they are put in an array list or linked list. Let's say a single such element is of size E. We should consider what we need to allocate space for in both an array list and linked list.

Even if there are n elements in an array list, the actual array used to implement an array list is going to be much bigger than n because we resize, which produces unused space. Regardless of the space being used or not, they're each all just as big as a single element. So if our actual array is of size D, the amount of space we use for the array list is $D \cdot E$.

For a linked list, we have the elements and pointers to the next one. Each element has a pointer to the next element (including the last one, which just points to null to signify the end of the list). So if a pointer is of size P, a linked list of n elements is $n(P + E)$ space because for each element, we have the element itself and a pointer to the next.

I'll tell you this: for the most part, the linked list is going to be more space efficient than the array list. But there could be a time where the array list has less space than the linked list. So this would be when $n(P+E) > D \cdot E$ or $n > \frac{D \cdot E}{P+E}$. This is called the "break-even point." Maybe let's do a demonstration?

Suppose a pointer was 4 bytes and an element was 7 bytes, so $P = 4$ and $E = 7$. So the break-even point is when $n > \frac{7D}{11}$, as derived above. For n above the break-even point, the array list is better spacewise. For n below the break-even point, the linked list is better spacewise.

Here's the intuition behind this. Suppose our actual array was 256 so $D = 256$ and the break even point is roughly $n > 162$. Now, whether the array list has size 1 or size 100 or size 210, the actual array will always take up 256 spaces worth of elements (before we have enough elements to the point where we need to resize).

So if our list is of size 1, it's clear $D = 256$ for the array list is a massive waste of space and why the linked list is better. But when we add an extra element, the linked list would increase in space by adding a pointer and extra element. The array list, however, doesn't increase in space when we add an extra element because some of the previously unused space now gets allocated for the second element. That's the idea here: as we add more elements to the array list, we use up the previously wasted space, whereas we necessarily need more space for the linked list. So eventually, when the linked list gets big enough, we create more pointers and space for elements that it eventually makes up for the originally wasted space in the array list. With the numbers above, we hit this threshold (the break-even point) at $n = 163$ when the pointers for the linked list and elements outweigh all the 256 original slots in the array list (even with some slots being unused).

Also, to avoid wasting so much space like we did, we can implement a resize down when we have a ton of unused space. Suppose the actual array is of size D. If we find that we have less than $\frac{D}{4}$ elements, we should resize the array to size $\frac{D}{2}$ (halve it essentially).

§13.6 Summary

- We saw how the linked list and array lists would be implemented and how they work. In particular, an array list has to secretly resize if the actual array being used reaches its maximum capacity.
- We learned about amortized runtime, which thinks about the average in the long run because the situation is that the worst case happens only so often. For example, we may need to resize the array to be able to add a new element but only once in a while.
- We looked at space and how there are times where a linked list may be more efficient than an array list and vice versa, which is the break even point.
- An array list may waste a ton of space at first with unused elements, but it gets made up for as new elements are added and as a linked list requires more space for pointers and elements.

§14 Stacks and Queues

§14.1 Introduction

So we've learned about array lists and linked lists. Can we go a step further and use them to implement other data structures? Yes, we can. We'll see that with stacks and queues.

Definition 14.1 (Stack)**.** A list that follows a "last in, first out" property. That means, when you add a new element in to the list, it always goes to the end. However, when you want to remove an element from the list, you can only remove the element at the end.

The analogy for stacks is, you have a stack of plates. You can add extra plates on top of the stack. But it would be dangerous to try to pull out a plate from the middle or the bottom. So the only safe plate to remove is the one from the top and that's how you remove plates. It's a useful data structure when you have stuff and stuff added more recently has priority.

Definition 14.2 (Queue)**.** A list that follows a "first in, first out" property. That means, when you add a new element in to the list, it always goes to the end. However, when you want to remove an element from the list, you can only remove the element at the start.

The analogy for queues is like a real world queue. Don't want to cut the line, as that would be rude. So you go to the back. Then, some people may arrive behind you while people up front get called to do whatever they were waiting in line for. It's a useful data structure when you have stuff and stuff found first has priority.

Now that we're acquainted to the definitions, let's examine the implementation and functions of stacks and queues.

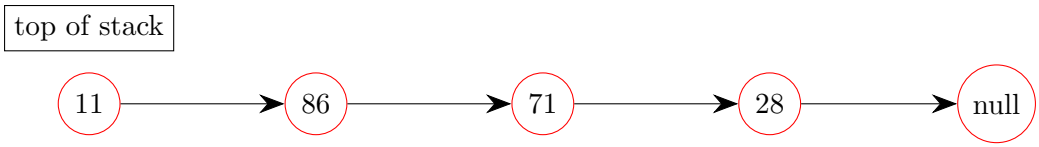
§14.2 Stacks

There are three operations that must be supported for it to be a stack:

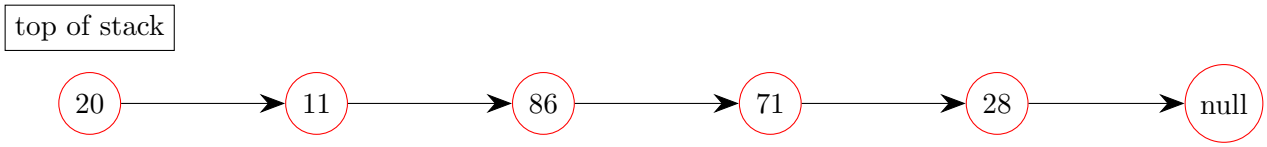
- Push: add an element to the end of the list
- Pop: remove the element at the end of the list
- Peek: return the element at the end of the list

We can implement a stack with either a linked list or an array list. Let's think about both.

So for a linked list, we know the action is happening at the end of the list, or the top of the stack. While it may seem weird at first, let's have the "end of the stack" (or the top of it) be the start of the linked list. This way, we can have a pointer to where the action is happening, and the pointer can take us to the top of the stack in $O(1)$ time.

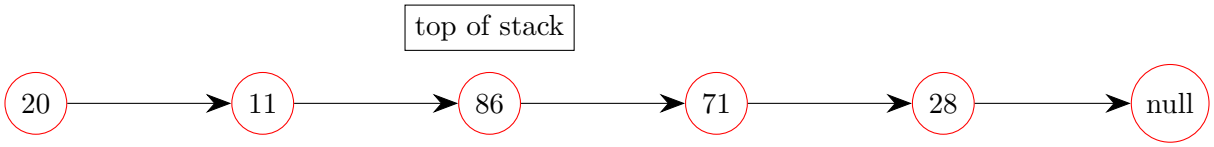


Say like we want to push 20. Easy way to implement this is by creating a new node whose next is the current top of the stack. Then, change the "top of stack" pointer so it goes to the new start:



This is all $O(1)$ time: adding a new node and pointer and redirecting the top of stack pointer.

As for pop, we can have the top of stack pointer move to the right instead of the left with the next pointers guiding us to the next plate in the stack. So if we wanted to pop twice, we'd move the top of stack pointer:



But realistically, we no longer need the 20 and 11 nodes and having them stay there is just a waste of space. So before moving the top of stack pointer, we'd want a temporary variable storing the node we're about to destroy, move the top of stack pointer to the next node, and then destroy the unwanted node by setting it to null:

One push is $O(1)$: just redirect a pointer and destroy the node behind it after.

As for peek, that's easy. Just see what node the top of stack pointer (because the pointer should always be at the end of the list) takes you to and return the value there. That's $O(1)$ as well.

Using an array list is a bit trickier but still manageable. We can have some sort of counter that keep tracks of how many elements we have as we add or remove them, so we always know the position of the last element and get there in $O(1)$ time. Push is just append, while pop is just removing something in the last slot (which luckily, doesn't force you to shift any of the elements over). And peek is easy: just use the counter to get the position of the last element and go there.

As for runtime, push is amortized $O(1)$ (because it is append in an array list, which may run in to resizing issues). Pop is remove without shifting elements but also runs in to resizing issues (resize down), so this is also amortized $O(1)$. Peek is regular $O(1)$.

Also, for both implementations of a stack, we can have an isEmpty and size method. Both can easily be done through a counter monitoring the number of elements we have and thus, we just check a variable. For isEmpty, we would check if the counter is 0, while size, we just return the counter. Both are $O(1)$ time.

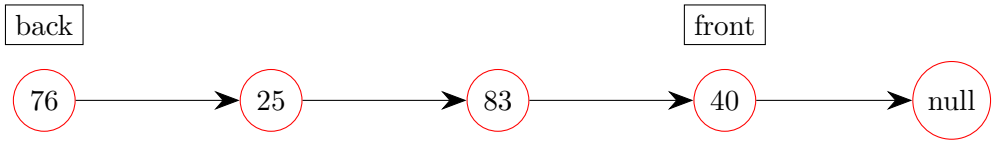
§14.3 Queues

There are three operations that must be supported for it to be a queue:

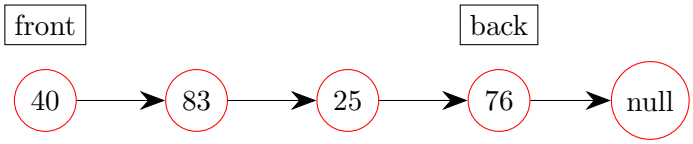
- Enqueue: add an element to the end of the list
- Dequeue: remove the element at the start of the list
- Peek: return the element at the start of the list

Again, we think about how we would implement this in both a linked list and array list.

Enqueue is basically the same as push in a stack. But the problem is that dequeue takes place in a different location than enqueue. So let's have two pointers: one for the front of the list and one for the back of the list. A pointer to the front of the list serves as a shortcut when we need to do something there because I don't want to have to traverse the entire list just to get from the back to the front. Let's see what happens if we have the "back" be the start of the list and the "front" be the end of the list:

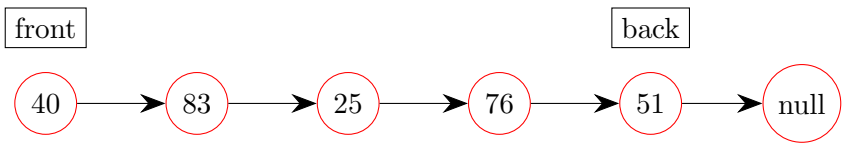


Well, it turns out if we try to implement the queue this way, we run into an issue with trying to implement dequeue. We want to move the front pointer to the left and then destroy the node behind... except we can't do that because the nodes don't have pointers to the left! Maybe let's switch where the front and back of the queue is, instead?



Seems weird that the queue arrows are pointing from the front to back, but bear with me.

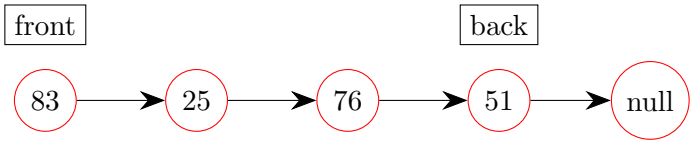
So how will enqueue work? Add a new node whose next is null. For the current back, change its next to that new node and move the back pointer to the right. So let's enqueue 51:



This is $O(1)$.

As for dequeue, we implement it similarly to pop from a stack (even though popping takes place at the end of the stack and dequeue takes place at the start of a queue).

So dequeuing means remove 40 and the new front is 83:



Also $O(1)$. As for peek, that's easy. The front pointer takes us to the node with the value we want. Simple $O(1)$.

As it turns out, an array list will be quite tricky. There will be a problem that pops up that we need to address, related to size. Okay let's have an array list of size 8 and enqueue 5, 7, 14, 23, then 2. We can keep track of what position we're at as we add in new elements.

1	2	3	4	5	6	7	8
5	7	14	23	2			

Eventually, we need to dequeue, so suppose we dequeue the first 3 elements. Well, if we dequeue 5, I don't want to have to shift everything over. So let's not shift stuff over as we dequeue.

1	2	3	4	5	6	7	8
			23	2			

Now, let's enqueue 37, 21, and then 15:

1	2	3	4	5	6	7	8
			23	2	37	21	15

Now let's try to enqueue 10. Well, we're at the end of the array, so the computer may think we're at full capacity and we need to resize... except that we're not actually at full capacity. We're just wasting a bunch of space near the front because we decided not to shift everything over. But shifting everything over takes time and we don't want to have to do it for every single enqueue.

I've got it! What if we say the next position in the array after the last one is the first one, so we can put the elements we need to enqueue in the empty space?

Definition 14.3 (Circular Array)**.** An array where we say the position after the last one is the first one.

Okay, so now if we want to enqueue 10, we put it in position 1. And if we want to enqueue 17 after that, we put it in position 2:

	B		F				
10	17		23	2	37	21	15

The F and B represent the front and back of the queue, respectively.

But how do we go about implementing this in code? After all when we want to enqueue 10, we don't put it in a non-existent 9th position. So we have to use modulus for m, where m is the size of the actual array. So if we're at index b (assuming the array is 0-indexed) for back and we want to add something, the new back is index $(b+1)\%m$ (the modulus is what allows the wrap around to happen). So if we exceed m, we just go back to the start of the array because of remainder.

Also, notice 37 is technically the 3rd element in the queue. But it's not at the 3rd position in the actual array. We know though that position in the queue is measured in relation to the front. So if we want the kth position in the queue, that's $k-1$ positions to the right of the front. But if we want the 7th position in the queue, 6 positions to the right of the front goes off the array... again, modulus saves the day! We

go to index $(f + (k-1))\%m$ (where f is the index of the front), so if we go off the array, we go back to the start instead because of remainder.

Well, that's enough on the intuition of a circular array. I should actually tell you how to implement the queue operations. From the diagram above, it becomes obvious that we know the actual array is full when B is one position to the left of F (so that gives us an indicator of when we need to resize).

If we want to enqueue in the circular array, we just add the element to the spot right of the B pointer (where the spot right of the last spot is the first spot) and then move B one spot to the right. For dequeue, we remove whatever is currently at the F pointer and move the F pointer one spot to the right. For peek, we just access whatever is at the F pointer.

So enqueue and dequeue are both amortized $O(1)$ because we may need to resize, but other than that, the stuff we do is constant time. Peek is also clearly $O(1)$.

Like with stacks, for both implementations of a queue, we can have an isEmpty and size method done in a similar way with a counter. Both are $O(1)$ time.

§14.4 Summary

> **Warning 14.4.** When using stacks and queues, they'll most likely make you assume it's an array list, so you have to state **amortized** $O(1)$ for push, pop, enqueue, and dequeue.

So when using a linked list as a stack or queue:

- Push in a Stack: $O(1)$
- Pop in a Stack: $O(1)$
- Enqueue in a Queue: $O(1)$
- Dequeue in a Queue: $O(1)$
- Peek for both: $O(1)$
- isEmpty for both: $O(1)$
- Size for both: $O(1)$

but when using an array list:

- Push in a Stack: amortized $O(1)$
- Pop in a Stack: amortized $O(1)$
- Enqueue in a Queue: amortized $O(1)$
- Dequeue in a Queue: amortized $O(1)$
- Peek for both: $O(1)$
- isEmpty for both: $O(1)$
- Size for both: $O(1)$

As for the conceptual knowledge:

- Linked list was relatively easy for both a stack and a queue. However, we did have to orient the list in such a way that we could conveniently add and remove elements at the correct spots when necessary.
- Most of the action in a stack or queue takes place at one particular location (front or back), so establishing pointers helped us quickly get to those key locations.
- An array list as a stack ran into the resizing problems we learned about previously, but is otherwise nothing new.

- As a queue, however, an array list needed a solution. Shifting elements over every time we dequeue wastes time, but leaving unused slots at the start wastes space. So we solve both in one swoop by using a circular array. This allows us to cover the front that would otherwise be wasted and not have to shift everything over.

§15 Heaps

§15.1 Introduction

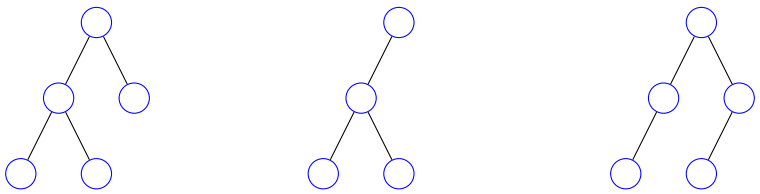
So let's say we have some video game tournament and we want to have a real-time leaderboard. But we have 1000 players and only want to display the top 10. It would be a real hassle to have to sort scores of 1000 players, regardless of what sorting algorithm we use. Not to mention if a player's score updates, we would have to update the ordering quickly because it's a real-time leaderboard. But remember, we only care about the ordering of the top 10. The bottom 990? We don't care about the ordering from one another, just that none of them have a high enough score to make it to the top 10. Introducing heaps! Which are useful for getting the max (or min) at any given point but don't care too much about the relative order of the other stuff.

First off, some binary tree vocabulary:

Definition 15.1 (Full binary tree)**.** A binary tree where all internal nodes (non-leaf nodes) have exactly 2 children.

Definition 15.2 (Complete binary tree)**.** A binary tree where all levels except the lowest level have to be full (the lowest level does not necessarily have to be full). Furthermore, all nodes on the lowest level are as far to the left as possible. Note that a complete binary tree with exactly k nodes (for some randomly fixed positive integer k) has exactly 1 possible shape.

Here's are three binary trees:



The first one fits the definition of a complete binary tree. The second one doesn't fit the definition because one of the levels besides the last level (the second to last one here) isn't full. The third one also doesn't fit the definition because the nodes on the last level are not as far left as possible.

Also, we're going to make a distinction between a "min heap" and a "max heap."

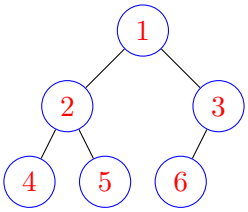
§15.2 Properties of a Heap

First of all, here are two properties a min heap has to maintain:

- Must be a complete binary tree
- Any internal node has a smaller number than its two children. It doesn't matter if the smaller of the two children are on the left or right.

A max heap is the same, except every internal node has a bigger (instead of "smaller") number than its two children.

The binary tree stuff is nice to visualize the heap. But when we implement it in Java, we're going to be using an array. The complete binary tree properties make implementation via an array easy. Let's take a complete binary tree of size 6 and number the positions in a left to right, top to bottom fashion:

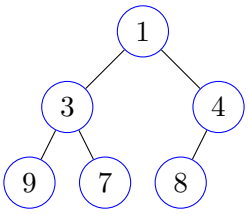


The complete binary tree properties make this numbering easy. There's no gaps in any of the levels because all levels before the last one are guaranteed to be full and the last level has all nodes as far left as possible. In fact given a node at position i:

- Its parent is at position $\lfloor \frac{i}{2} \rfloor$ ($\lfloor x \rfloor$ is the floor function, which means if x isn't an integer, round it down to the next integer less than x; otherwise, the floor of any integer is equal to that integer)
- Its left child is at $2i + 1$ (provided i isn't a leaf node)
- Its left child is at $2i + 2$ (provided i isn't a leaf node)
- Its left sibling is $i - 1$ for odd i (it doesn't have one if i is even)
- Its right sibling is $i + 1$ for even i (it doesn't have one if i is odd)

This works for any complete binary tree of any size because of the whole "no gaps" intuition. Try a few examples for yourself if you're not convinced. All this position stuff makes the array implementation even more practical because we have a surefire way of knowing where to find the parent, child, sibling, etc. of whatever node we want.

Anyway, let's take that complete binary tree and fill it with numbers in each node (instead of labeling positions). In fact, I chose numbers so that it would actually be a min heap:



Well, with numbers given to each node, as well as the corresponding position system we have, we can easily make an array representation:

1	2	3	4	5	6
1	3	4	9	7	8

And again, our position system allows us to get the parents, children, and whatever of any particular node in $O(1)$ time.

Also, let's consider how many levels a complete binary tree of n nodes has. 1 node? Just 1 level. 2-3 nodes? 2 levels. 4-7 nodes? 3 levels. 8-15 nodes? 4 levels. Not hard to see by drawing pictures. You can catch the pattern that if there are n nodes, we have $\lfloor \log_2(n) \rfloor + 1$ levels. Point is, the number of levels will be $\log(n)$, give or take. This will be needed for analyzing runtimes later.

§15.3 Overview of Operations

I'm going to be using max heaps to demonstrate all the operations and intuition. Min heaps have a very similar set of operations that follows the same logic. So here are some operations that we need to implement:

- Insert (add a given number into the heap)
- Remove the maximum (we don't care about removing anything other than the maximum)

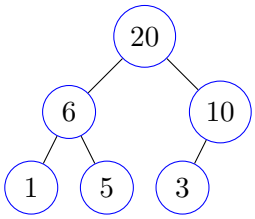
- Max Heapify (given a complete binary tree that may not satisfy the heap property at its root, move nodes around until it does)
- Build Max Heap (given an array, create a complete binary tree out of it and move nodes around until it satisfies the heap property)
- Peek (return the maximum)

With insert and remove, where we're necessarily changing the collection of elements in the heap, we have to make sure we preserve the heap property (any internal node being greater than its two children).

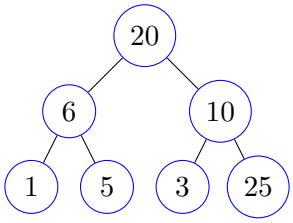
§15.3.1 Insert

- Insert the new element in the last slot of the complete binary tree. If the last level is full, that means creating a new level and inserting the new element in the first slot available there.
- Compare the new element to its parent. If the new element is larger than its parent, swap the two elements. Keep doing these swaps between the new element and its parent, until the parent is greater than the new element, or it reaches the root.

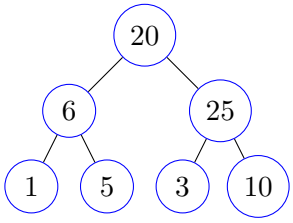
Allow me to demonstrate with this max heap:



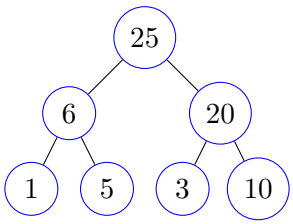
Now let's insert 25:



Well, 10 isn't happy because it's a parent less than one of its two children. That's why we have the swapping steps. So $10 < 25$, meaning we swap the two.



And then we check if 25's new parent is satisfied. $20 < 25$, so we need to make another swap:



The swapping business is easy to implement in the array because again, we have formula to find the position of a node's parent.

Okay, we can tell this particular example of insert is a max heap. But can we be sure this algorithm always gives us a max heap at the end?

Yes. We can consider induction where, given a max heap of size k, this insertion leads to a max heap of size $k+1$. One key observation is, a parent is greater than its two children, so its greater than the children of its children, and the children of its children of its children. And so on. This means any node is greater than anything in either of its two subtrees.

So consider when we did the swap for 20 and 25. 20 having the 3 and 10 subtrees as its ancestors in the original heap meant it was greater than both of them. So there was no problem in 20 becoming the new parent to either of them instead of 25. Also, we saw 25 become the new parent to the 6 subtree. 20 was already big enough and 25 is even bigger, so it doesn't cause problems becoming the new parent of that subtree.

Basically, as we make swaps for 25 upward, the new numbers that are displaced along the way do not cause problems. And 25 itself does not cause a problem when it reaches its final position. That's the intuition, the formal proof is a pain to write up.

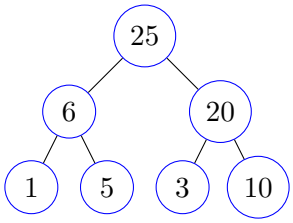
As for runtime, the worst case is we make swaps all the way up. One swap is constant time, but we can only make as many swaps as there are levels because we stop once we reach the root (the top). So this is $O(\log n)$ because there are that many levels.

Of course, we could stop prematurely. If we had inserted 4 originally instead of 25, then we don't do any swapping at all because it's already a max heap without problems.

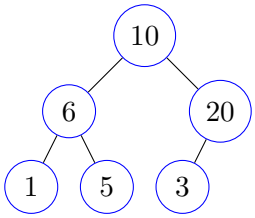
§15.3.2 Remove

- Swap the numbers in the root node and last node.
- Delete the new last node.
- Track the number currently in the root. If it's greater than its two children, don't do anything. Otherwise, swap it with the larger of its two children. Keep making swaps until that number being tracked is greater than its two children or it reaches the bottom.

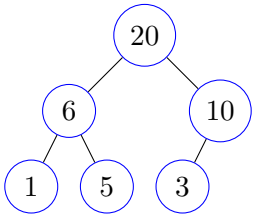
Okay, so we have that tree from the insert example:



Let's call remove, which in essence, should get rid of 25 (the maximum) and adjust the tree so it is still a heap. So we swap 25 with 10 (the last node) and then delete 25:



Obviously, the 10 causes problems, which is why we have our swapping procedure. Okay, between 6 and 20, 20 is bigger, so we make the swap:



Okay, 10 no longer causes problems, so we're good.

Proof of correctness that this operation always yields a max heap at the end is similar in spirit to insert, where we make sure any numbers displaced along the swaps do not cause problems and the number being repeatedly swapped doesn't cause problems in its final position. Take note though of why we always chose the bigger of the two children to swap with. Had we swapped 10 and 6 originally instead of 10 and 20, we would have a problem where $6 < 20$, meaning the smaller of the two children becomes the parent for the bigger child, which is bad.

So with delete, we have this intuition where we have a complete binary tree and all nodes are happy and satisfy the heap property, all nodes except possibly the root. But with a series of swaps with the root, we can always make it into a heap. This was seen when the last node became the root, and we had to swap it down the tree, but we knew that was the only node that was causing problems. This intuition will be necessary to understand max heapify.

Runtime is also $O(\log n)$, similar to insert. Worst case, we have to swap all the way down and the single swap and delete we make at the beginning is constant work.

§15.3.3 Max Heapify

When I say a node is "happy" that means, it's greater than its two children. Anyway, take a rooted complete binary tree, where the root may or may not be happy but we know everything else in the tree is. We can turn it into a tree where all the nodes are happy through this procedure (and thus a heap):

- Track the number in the root. Do the series of swaps like in the last step of remove. If it's greater than its two children, don't do anything. Otherwise, swap it with the larger of its two children. Keep making swaps until that number being tracked is greater than its two children or it reaches the bottom.

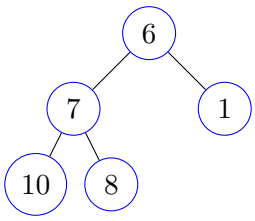
It basically uses the intuition in remove for why the swaps do not cause any problems. You can reuse the example for when 10 is the root and 25 is deleted and trace the work from there. Clearly, this is $O(\log n)$ in the worst case where we have to swap all the way down.

§15.3.4 Build Max Heap

Take any complete binary tree that may or may not be a max heap because of nodes that are not happy. We can turn it into a max heap through this algorithm. Going in reverse order by node position number (from largest position number to smallest in the tree):

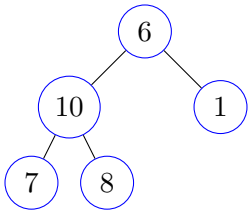
- Check the rooted tree that has the node at position i as the root.
- Use max heapify on the rooted tree

So I'll walk you through an example, while giving commentary on why this works so that it doubles as a proof of correctness.

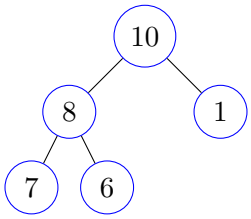


Okay, so we check the rooted tree at position 5, which is just 8. Already a heap. Same with the trees at positions 4 and 3. Now, check the rooted tree at position 2, which is in a situation similar to remove. We have the root at position 2, which may or may not be happy. But the nodes in both of its subtrees are already happy because we already managed those beforehand.

Like we went over in max heapify, whenever we have a rooted tree where the root may or may not be happy, but all the nodes in its subtrees are, the swapping procedure always gives us a heap, making the whole rooted tree happy. So we do the swaps appropriately and now, the rooted tree at position 2 is happy.



So the rooted trees at positions 2-5 are all happy and we worked through them. Now for the rooted tree at position 1. This is once again a situation where the root may or may not be happy but the two subtrees are. So we do our swapping procedure to get:



Did you catch the inductive logic here and why we went in reverse order by position? We know max heapify works any time we have a situation where the root may or may not be happy but everything in the subtrees are. So we can always fix the rooted tree so that the whole rooted tree is happy. We apply that here where we work backwards specifically because it ensured lower rooted trees are happy before we get to upper ones. The base case are the leaves on the lowest level, which are happy by default.

Now, you would be forgiven for thinking this is $O(n \log n)$ time, where we have n nodes and worst case scenario, max heapify is $O(\log n)$. You're not wrong, but we can get a better, more informative upper bound.

Not all nodes have to swap the height of the entire tree down in the worst case. If a node is in the 2nd to last level, worst case, it's only going to be swapped once. If a node is in the 5th to last level, worst case, it's only going to be swapped 4 times. Only the nodes closer to the top will have to go the height of the entire tree down in their worst case.

Label the levels from top to bottom as level 0, then level 1, then level 2, and so on, until level $\log(n)$. It can be seen through drawing some pictures that level i has 2^i nodes (except possibly the last level, which may have less than that if it's not full). Also, in level i, exactly $\log(n) - i$ swaps are required to reach the bottom. Worst case, all 2^i nodes in level i swap all the way down, accounting for $2^i(\log(n) - i)$ swaps. Summing across all levels, our worst case for the number of swaps is

$$T(n) = \sum_{i=0}^{\log(n)} 2^i(\log(n) - i) = 2^0(\log(n)) + 2^1(\log(n) - 1) + 2^2(\log(n) - 2) + \cdots + 2^{\log(n)-1}(1).$$

Some unmotivated math here. Multiply this equation by 2:

$$2T(n) = 2^1(\log(n)) + 2^2(\log(n) - 1) + 2^3(\log(n) - 2) + 2^{\log(n)}(1).$$

Subtracting this with the equation for $T(n)$ above cancels stuff out neatly. For example, we have $2^1(\log(n)) - 2^1(\log(n) - 1)$, which becomes just 2^1. So

$$2T(n) - T(n) = -2^0(\log(n)) + (2^1 + 2^2 + \cdots + 2^{\log(n)-1}) + 2^{\log(n)}(1)$$

Some further simplifications:

$$T(n) = -\log(n) + 2(2^0 + 2^1 + 2^2 + \cdots + 2^{\log(n)-1}) = -\log(n) + 2(2^{\log(n)} - 1) = -\log(n) + 2(n - 1).$$

This is $O(n)$, a more informative bound.

§15.3.5 Peek

Just look at the first element in the array. We know its the largest because it's greater than anything in the two subtrees by the heap property. So this is $O(1)$.

§15.4 Heap Sort

So we have min and max heaps. Suppose we have n elements in a min heap. For a min heap, we can get extract the minimum by removing it and the heap makes adjustments so that it still remains a heap. By then, we have a new minimum, which if we remove again, we should get the new minimum. This new minimum should be the second smallest element from the original n elements. If we do this enough times, we can get the elements in increasing order.

Okay, maybe we can make a sorting algorithm out of this? We know we can build the heap in $O(n)$ time. Then, we keep removing n times, which are each $O(\log n)$ worst case, so that's $O(n \log n)$ total.

So the whole sorting algorithm is $O(n \log n)$ and that's worst case. Better than quick sort being $O(n^2)$ worst case.

§15.5 Summary

- We learned about the operations of a heap. We can turn a collection of numbers into a min or max heap.
- The heap is a complete binary tree, which makes it easy to represent as an array: given a node's position, we then know the positions of its parent, children, and sibling.
- The maximum is conveniently stored at the top of a max heap (and minimum for min heap).
- Then, we can add numbers all we want to the heap to maintain the key properties of a heap.
- We can also remove the maximum from a max heap at any time (or minimum from a min heap) and still be able to fix the heap.
- We can also create a sorting algorithm out of a heap.
- The heap is again, useful for keeping track of the top 10 or so elements, when you don't care about the order of anything not in the top 10. This is because you can have anything not big enough to be in the top 10 excluded from the heap and sort the top 10 in $O(1)$ time.
- We will see algorithms later that necessitate the relative quickness of a heap when we care about the minimum but not the order of anything else.

§16 Recap on All Our Sorting Algorithms

We know 4 sorting algorithms now. Let's think about each of them.

I'll say this: it would be foolish to think that merge sort obviates quick sort just because merge sort is reliably $O(n \log n)$ compared to quick sort only attaining that time in expectation.

We need to introduce some vocabulary. The first one was alluded to before. The second one isn't.

Definition 16.1 (In-place Algorithm)**.** All the action happens within the input itself

Definition 16.2 (Stable Sorting Algorithm)**.** A sorting algorithm that maintains relative order on elements with the same attribute being sorted. Say we had a directory of names, where we had "John Brown" and "John Smith," and we could either sort by first name or sort by last name. Suppose we sort by last name first so we have "John Brown" before "John Smith." When we sort by first name (with the two Johns) now, will "John Brown" always come before "John Smith?" If so, then it's a stable algorithm. Otherwise, if it doesn't always happen, it's not stable.

In particular, stability is super important for searching when we want to have more than one different attribute we use as the sorting basis. The example above with search by first name order or search by last name order demonstrates that. A real world application would be a UPenn directory, where we would want to sort students in different ways: by name in alphabetical order, by graduation year, by major, and so on.

Anyway, let's talk about in-place. Insertion sort, quick sort, and heap sort are all just moving stuff around in the array. So they're all in-place. Is merge sort in-place? No because we have to allocate all

that memory for all those recursive calls until we reach the base case. That's going to use up a lot of memory and most of the action happens in the merging of all the arrays obtained recursively, not the original array.

And let's talk about stability. Insertion sort? Yes. We don't move an element any further than we need to. So John Brown gets inserted somewhere and then John Smith stops before John Brown because we notice the two Johns being the same in order by first name. Merge sort? Yes. When we merge two arrays, if we have a tie, we could prioritize the thing from the left array over the thing from the right array. So that preserves relative order of John Brown and John Smith.

Quick sort? Not stable. The pivots are random and when we do the partition, quick sort essentially just shoves elements to either side of the pivot without regards to their original relative order.

Heap sort? Not stable either. Especially when elements in consecutive positions could either be siblings sometimes but not all the time. So relative order won't be accounted for.

And runtimes in terms of best, average, and worst. We didn't do average for insertion. It's going to be $O(n^2)$, just like worst case because in expectation, we are going to insert halfway through the possible positions for each element. But halve the sum of the first n positive integers and you still get $O(n^2)$.

Merge sort doesn't ever stop prematurely. It always breaks it down to base cases and merges. It's not binary search where we stop prematurely if we accomplish the objective early. So its best and average are the same as the worst case.

Quick sort, best runtime is when the pivots are always in the center, so we have the recursion $T(n) = 2T(n/2) + \Theta(n)$, which is merge sort's time.

Heap sort? Well, we always have to build the heap, which is $O(n)$. Best case, when we extract the minimum, we don't do any swapping at all, so it is $O(1)$ per getting the minimum. So $O(n)$ best case. On average, a swap chain will probably be halfway down the tree, so $\frac{\log(n)}{2}$, but that's still $O(\log n)$. So average is $O(n \log n)$.

Sorting Algorithm	Best Runtime	Average Runtime	Worst Runtime	Stability	In-Place
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Yes	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	No
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	Yes
Heap Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	No	Yes

An informed programmer is going to look at the situation where they need a sorting algorithm, weigh the importance of the characteristics, and make their choice on the sorting algorithm based on that. For example, we might not have enough memory to implement merge sort, but we also need an $O(n \log n)$ time to do it fast enough. So we might look at heap sort. Maybe we don't care about runtime too much, but we need stability and we need to not use up too much memory. Then, we'd do insertion sort.

Advice 16.3 — This will most likely appear on the final where they may ask you on the attributes of different sorting algorithms.

§17 Huffman Trees

§17.1 Introduction

So we may need to send a large text file over the internet. Problem is, it may be so big that it could be faster to just FedEx a flash drive with the file instead. Maybe we can compress the file into something smaller to make it faster to send and then decompress it once it reaches its destination?

You may have heard of ASCII or American Standard Code for Information Interchange. This is a way of assigning a sequence of 7 bits to each of the various characters used by a computer. For example, 'H' is represented as 1001000, while '7' is represented as 0110111. There are 95 characters in the ASCII system. With 7 bits, there are $2^7 = 128$ possible different binary strings. So we can assign each of those 95 characters a unique 7 digit sequence of bits.

However, there's a good chance we're not going to need EVERY single character in ASCII. So for this large file, why not create our conversion system to change the characters to binary using only the relevant

characters? Also, there's no reason why each character's corresponding binary sequence all have to be the same length.

Suppose our file consisted of a ton of 'a's and a couple 'b's and 'c's that do not appear anywhere near as often as 'a'. One conversion system is to let 'a' be 0, 'b' be 10, and 'c' be 11. While b and c have longer strings compared to a, it won't be needed as often, since as said, a appears much more. So in this system, we have efficiency in needing less bits. But we want a generalize this.

§17.2 The Algorithm

To set things up, we need the alphabet we want to convert into an encoding system with their corresponding frequencies. The frequencies can appear as a probability, or as numbers. For example, suppose the file we want to convert has 100 characters, 90 of which are 'a's, 6 'b's, and 4 'c's. The numbers for each character are the frequencies in of itself. Or we can say 'a' for example has frequency of 0.9 because that's how many 'a's we have relative to the total number of characters ($\frac{90}{100} = 0.9$).

Suppose we have n total characters in our alphabet. Here's how we construct the tree:

1. Create n nodes, one for each character that shows up. Put the corresponding frequency of each character on their respective node.
2. Find the two nodes available with the two smallest frequencies.
3. Have those two nodes as children to a parent node. Have the frequency of this parent node be the sum of the frequencies of its two children.
4. Place this new parent node back in the collection with the remaining nodes.
5. Repeat steps 2-4, decreasing the number of remaining nodes available by 1 each time, until we have 1 node left. This 1 node is the root of the whole tree.

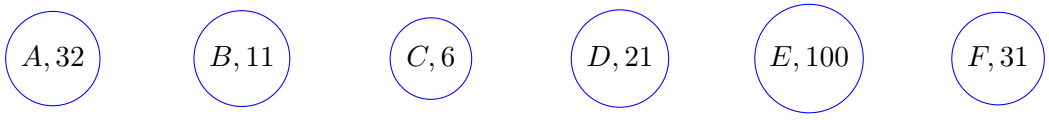
Suppose you're doing this on paper and you have, let's say, 5-6 characters in your alphabet. It's pretty easy to eyeball which are the two smallest frequencies for each time you do step 2. However, for larger alphabets and in code, you want to use a binary min heap to help speed up step 2.

Example 17.1 (Made Up Example)

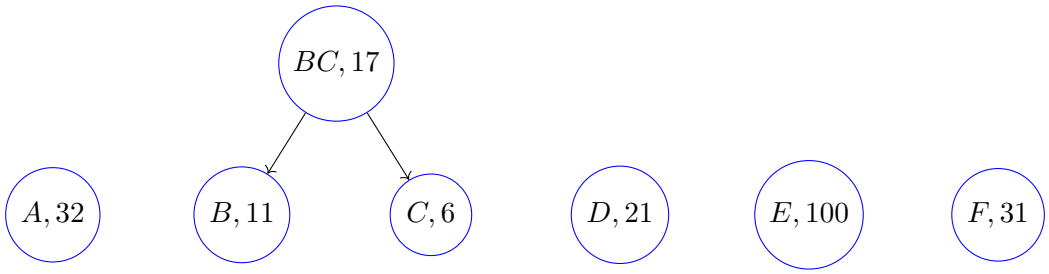
Build a Huffman Tree of an alphabet with the following frequencies:

Letter	A	B	C	D	E	F
Frequency	32	11	6	21	100	31

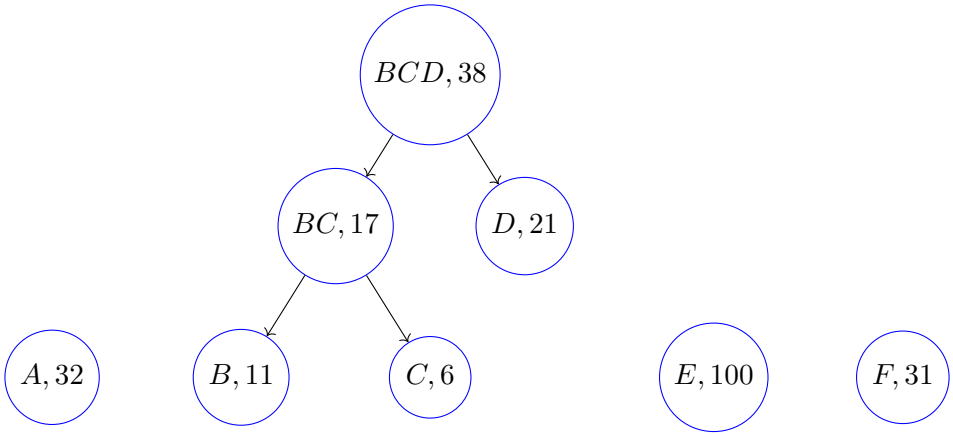
First, we create the 6 nodes as said in step 1.



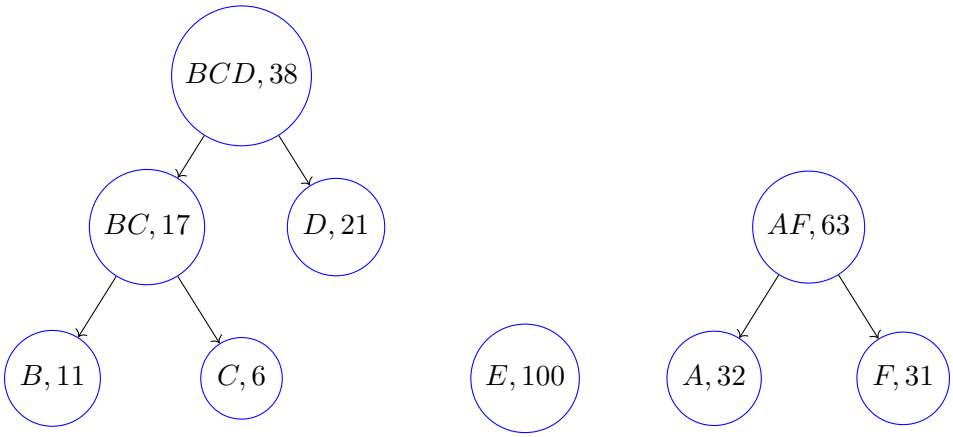
Now, we look at which 2 have the lowest frequency, which are B and C. So we create a parent node (I'm labelling it BC for convenience) with a frequency being the sum of the frequencies of B and C:



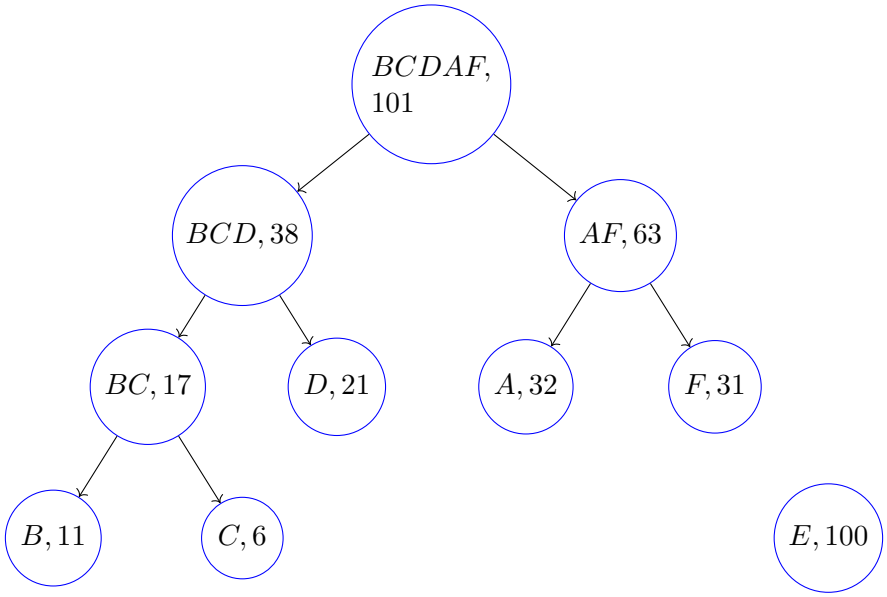
At this point, the nodes for B and C are no longer in the collection and are treated as if they are "merged" into the node BC. But we can't delete the B and C nodes outright because they'll be important for the full tree. Anyway, just pretend BC is a single node that secretly has 2 children. So we have 5 nodes left: A, BC, D, E, F. Now, we take the two lowest frequencies, being BC and D and do steps 3 and 4 again:



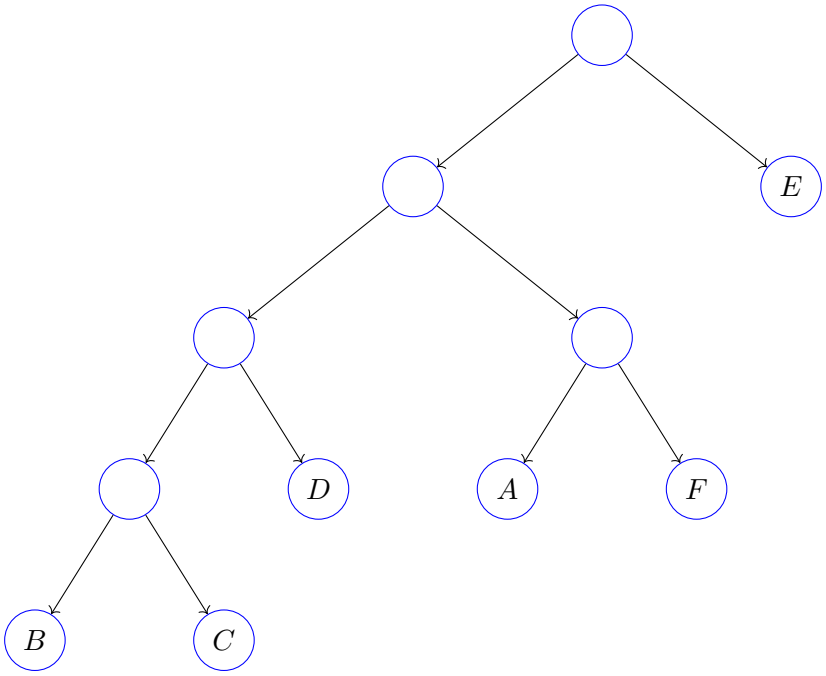
I think you get the hang of it, but I'll show the remaining merges we need to do. First, merge A and F:



Now, merge BCD and AF:



Now, merge $BCDAF$ and E for our final tree (with labels removed on the internal nodes and removing frequencies):



Now, to generate the set of binary codes for each letter. We do this by considering the path from the root of the tree to said letter and the sequence of left and right movements necessary to get there. Once you write out the sequence, change each "left" movement to a 0 and every "right" movement to a 1. That gives you the code.

For example, to get to E, we start at the root and make one movement right. So E's code is 1. To get to A, the sequence is left, right, left. So A's code is 010. Here is a table of codes:

Letter	A	B	C	D	E	F
Code	010	0000	0001	001	1	011

You use the above logic for any Huffman Tree to generate codes, not just this one example.

So when encoding, what we do is we just use the table to see how we convert a character to bits.

When decoding, we traverse the tree starting at the root, using the 0s and 1s as a guide until we hit a node with a character. That character is the next character in the original message. Then, we go back to the root.

For example, take 00011011. So the three 0s say go left three times. Still no character yet. Then, the 1 says go right. We hit C. So first character in the message is C and we go back to the root. We have a 1, which says go right. We hit E, meaning it's the next character in the message. Return to the root. Now, we have 011, which says go left once and then right twice. This takes us to F. So the original message is CEF.

It's easy to see the length of a character's encoding is simply the number of edges in the unique path from the character to the tree's root, or the level number that the leaf node is located in. (This intuition is very important for proofs)

§17.3 Proof of Correctness

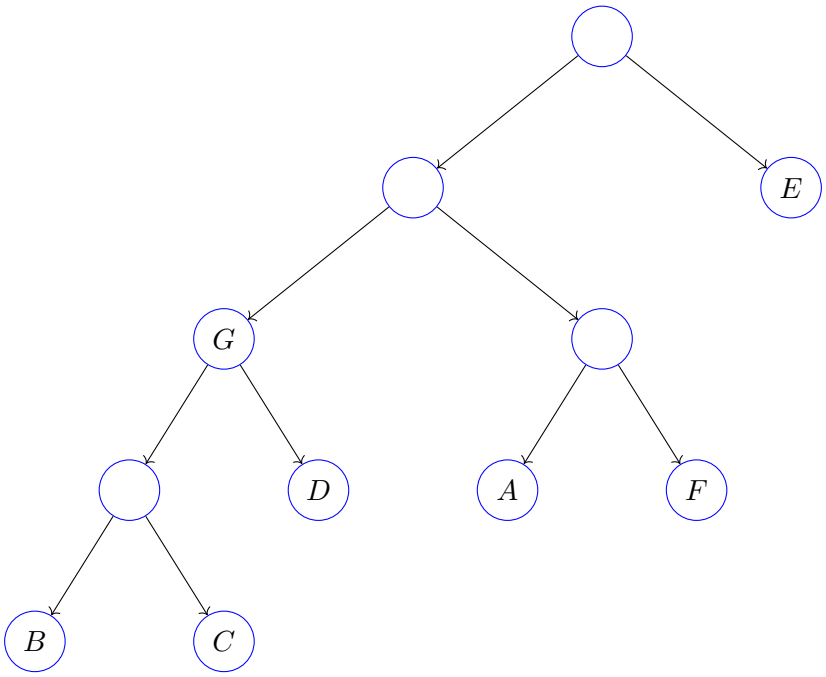
So there are two important things we need to prove:

- That this algorithm gives us an encoding that can be used to both encode and decode (as in, get the original message back from the 0s and 1s)
- That this specifically does it in the most optimal manner

Let's worry about the first one for now. Clearly, from the algorithm, every node will be added to the full tree at some point, so each letter will have a code. So it's clear we can convert a sequence of these letters into a sequence of bits.

But the thing to think about is, can we always get the original message back? Hypothetically speaking, what if we had $A - 011$, $B - 01$, and $C - 1$ and we needed to convert the bits '011' back to letters? Well, this could be either 'A' or 'BC.' Who knows which one it is! So we never want a situation like this, where there could be two or more possible original messages.

For demonstration purposes, I'm taking the Huffman tree we got at the last example and adding a G at an internal node just because it will teach us something.



Let's suppose we had a message that started with 'D,' so that when it becomes bits, the bit sequence starts with 001. But when we convert back to characters, what will happen is, we start at the root and go left twice, as the first two 0s imply. We hit a character node G, so the first character in the message back is 'G,' except we know that's not true because our original message started with 'D'! So what went wrong here?

Well, we always stop when we hit a character node and as you can see, D is "locked behind" G. So we can never actually reach D because G is at an internal node, blocking the path to D. Aha! So we never want character nodes to be at internal nodes because they block paths to other characters, so we may end prematurely and get the wrong character.

Also, we can clearly see that leaves function as dead ends, so if we always have character nodes at leaves, we never create this situation where a character node blocks the path to another. And by how Huffman works, we always have character nodes at leaves because we merge character nodes with some other node as two children of a parent, and that parent is never a character node. So we're safe here. In fact, for this reason, we call Huffman a prefix-free tree.

Definition 17.2 (Prefix-free tree)**.** A tree with codes where no two codes are prefixes of one another. For example, '011' and '01100' aren't allowed together in the same tree because the former is a prefix of the latter. If one code is a prefix of another, it will be visible in the tree as some character node being an internal node, blocking the path to another character node.

So this prefix-free aspect is what allows to decode bits without ambiguity because again, we don't reach some other character node prematurely before we can reach the node that was intended in the original message.

Now, we consider the second part: that this is the best way we can accomplish the encoding. To measure how well we can accomplish it, we introduce:

Definition 17.3 (ABL (Average bits per letter))**.** The expected length of the binary code of some random letter chosen from the alphabet sample (with frequencies taken into account). The smaller this is, the more optimal our encoding is.

The formula for ABL (which I will denote as $E(L)$ for some alphabet set L with frequencies) is reminiscent of the expected value formula. Let $f(x)$ be the frequency of some letter x in the alphabet, and $P(x)$ denote the length of x's corresponding binary string. Then,

$$E(L) = \sum_{x \in L} (f(x) \cdot P(x))$$

Back to that example at the beginning where we have 100 characters, 90 of which are 'a's, 6 'b's, and 4 'c's. I'll just tell you the corresponding binary strings from the Huffman tree is $a - 0$, $b - 10$, and $c - 11$. Then, the ABL is

$$E(L) = \sum_{x \in L} (f(x) \cdot P(x)) = f(a) \cdot P(a) + f(b) \cdot P(b) + f(c) \cdot P(c)$$

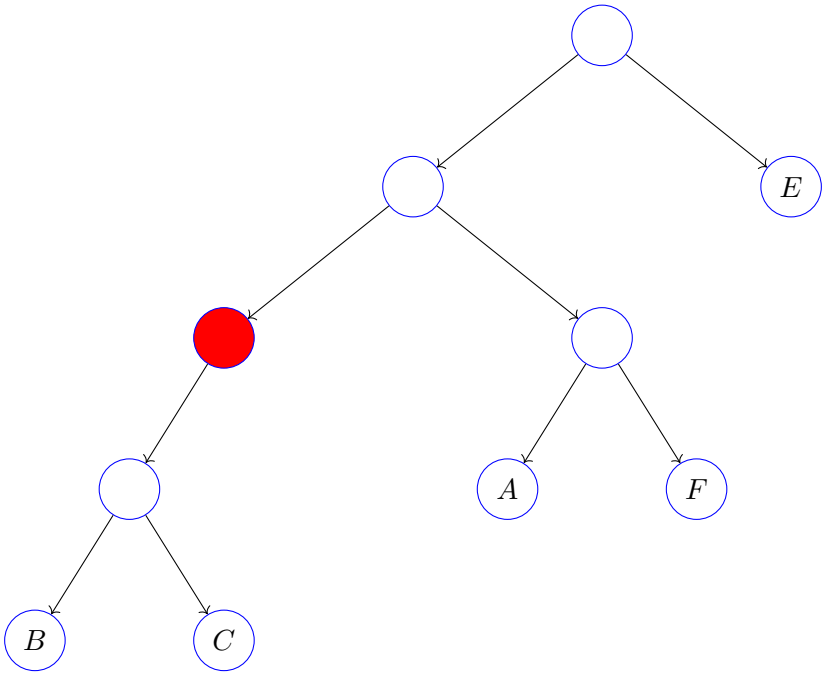
$$= 0.90 \cdot 1 + 0.06 \cdot 2 + 0.04 \cdot 2 = 1.10.$$

As you can see, it's a weighted average just like expected value is, where more frequent letters have bigger weights.

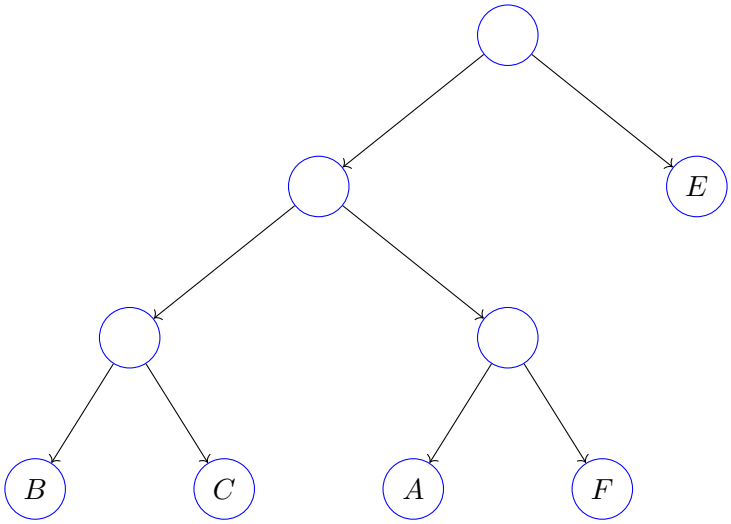
> **Advice 17.4 —** You may get a midterm question where you are asked to compute the ABL for a Huffman tree, so just remember it's like the expected value formula. Don't want to lose points on something relatively intuitive and easy to remember.

Okay, so the formal proof of Huffman's optimization is more trouble than it is worth, so I'll just give you the intuition.

First off, all internal nodes should have exactly 2 children to help with optimization. Why? Let's again, steal the tree from the example, but delete the D node for demonstration purposes. So here, we have a Huffman tree where as you can see, an internal node has only 1 child (which I colored in red).



Well, since the red node only has 1 child, what I can do is move its left subtree up and not change the fact that this tree is prefix-free.



Moving that subtree up reduced the distance from B and C to the root, which means less lengths for those codes and thus, lower ABL. So it's no good if we have an internal node with 1 child because it protracts the tree for no reason. In this regard, every internal node in a Huffman tree has two children, meaning it is a full binary tree.

Also, we can technically switch the characters around within the leaf nodes without changing the fact that the tree is prefix free. Like it makes no difference if we swapped the codes for E and B when it comes to being prefix free. But it may affect ABL. Keep this in mind.

Advice 17.5 — This technique of switching around stuff to get something that is valid but may be more optimal is called an "exchange argument." There will be a topic later on where we use this.

Lemma 17.6 (Exchange Argument)

If $f(x) > f(y)$ in an optimal tree, then $P(x) \leq P(y)$ necessarily follows.

Proof. Here, we use the exchange argument. Suppose for the sake of contradiction that $f(x) > f(y)$ in some optimal Huffman tree, but $P(x) > P(y)$. So somewhere in the ABL sum, we have $f(x) \cdot P(x) + f(y) \cdot P(y)$.

Suppose we switched the nodes for x and y in the Huffman tree. The frequencies don't change, but the path lengths of x and y do. $P(x)$ becomes $P(y)$ and vice versa. So somewhere in the new ABL sum, we have $f(x) \cdot P(y) + f(y) \cdot P(x)$. What is the net change from the original ABL to the new ABL? Well, that is

$$(f(x) \cdot P(y) + f(y) \cdot P(x)) - (f(x) \cdot P(x) + f(y) \cdot P(y)) = (f(x) - f(y))(P(y) - P(x)).$$

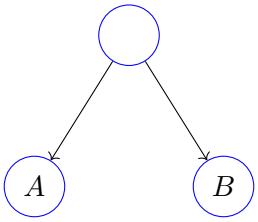
With $f(x) > f(y)$ and $P(x) > P(y)$, this quantity is negative. So the net change in making this switch is a reduction to the ABL! Remember, switching character nodes doesn't change the fact that we have a prefix free tree. But we showed we can make a simple switch that creates a better ABL if we ever have $f(x) > f(y)$ and $P(x) > P(y)$. So we never want that to happen because we can always do better when that occurs. The unoptimality of the tree before the switch was made serves as the contradiction. □

Now for the main part of the proof.

Lemma 17.7

Huffman produces the optimal tree for any alphabet on size $n \geq 2$.

Proof. Consider induction on n. The base case of $n = 2$ involves the Huffman making a tree looking like this:



I mean, the algorithm aside, we literally can't do any better than this for $n = 2$, right? 1 digit codes for both of the two letters, regardless of their frequencies. No prefixes. So the base case works for $n = 2$.

Now for the inductive step. So we assume Huffman always produces the optimal tree for an alphabet of size $n = k - 1$ and want to show that implies the IH holds for $n = k$.

Remember we can move the characters in the leaf nodes around without changing the fact that it's prefix free. We can use this later to argue the nodes with the two smallest frequencies should be at the bottom.

What Huffman will do is merge the two lowest frequencies among the k nodes to make a new node and place it back in the collection. Now, we have $k - 1$ nodes.

By the IH, Huffman will sort everything out for the $k - 1$ nodes in the optimal manner and create a tree T'. Let L' be the alphabet set involving the $k - 1$ nodes (the merged node can just be some character of its own). So we know $E(L')$ is minimal by IH.

If N is the first merged node created in the Huffman process for T, we know N is a leaf node somewhere in T'. So to get from T' to T, we just add N's two children. Obviously, the frequency of N is the sum of its two children.

Let's figure out how to get from $E(L')$ to $E(L)$. We have $f(N) \cdot P(N)$ somewhere in the ABL sum for L'. Nothing else will change except N is no longer and internal node and has two character nodes. These two new character nodes in T obviously have path length $P(N) + 1$ (one longer than its parent, N) and the two character nodes have a frequency summing to $f(N)$ because that's how the Huffman merging works. So basically, the $f(N) \cdot P(N)$ becomes $f(N) \cdot (P(N) + 1)$ in the ABL sum when we transition from L' to L, resulting in a net change of $f(N)$, so $E(L') + f(N) = E(L)$.

We could have the Huffman process merge the two largest frequencies each step or the two middle frequencies for all we care and still get prefix free codes. But it always merges the two smallest frequencies which means $f(N)$ is as small as it can be. We also don't want only 1 node in the deepest level of the tree because that would imply its parent has only 1 child (which we saw earlier is not optimal), which is why we make N have 2 children and not 1.

With $E(L') + f(N) = E(L)$ and $f(N)$ as small as it can be (as well as $E(L')$ by IH), $E(L)$ is as small as it can be. This completes the induction. □

§17.4 Runtime

Note that we should use a min heap to store all the nodes in the collection, as they will save us time.

Building the min heap is $O(n)$. Then, we extract the minimum twice ($O(\log n)$), do some constant work to make a new merged node with the two chosen nodes as children, and then put the new merged node back into the heap ($O(\log n)$). We do the extracting, merging, and insert back a total of $n - 1$ times because we have 1 less node each time, until we have only 1 node in the collection and that root serves as the tree. So this is

$$O(n) + (n - 1)(O(\log n)) + O(\log n) = O(n \log n).$$

§17.5 Problem Solving

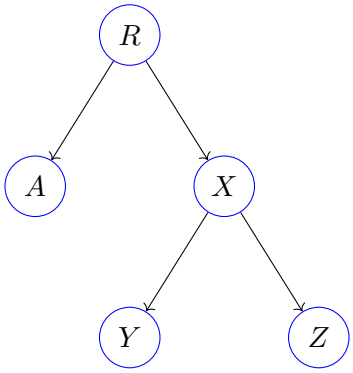
Huffman problems will often involve proof by contradiction and probability bounding. A key trick is that in every point in the process, the sum of the frequencies of all available nodes will always be 1 (shouldn't be too hard to see because the new node after merged is the sum of two previous ones, so the overall sum never changes). It's also important to remember Huffman always chooses to merge the two smallest nodes, which can help lead to a contradiction.

Example 17.8 (Recitation)

You have an alphabet with $n > 2$ letters and frequencies. You perform Huffman encoding on this alphabet, and notice that the character with the largest frequency is encoded by just a 0. In this alphabet, symbol i occurs with probability p_i; $p_1 \geq p_2 \geq p_3 \geq ... \geq p_n$.

Given this alphabet and encoding, does there exist an assignment of probabilities to p_1 through p_n such that $p_1 < \frac{1}{3}$? Justify your answer.

I claim there does not exist an assignment of probabilities. Assume for the sake of contradiction that there did. Let's draw out the Huffman tree to get a better understanding of what's going on here.



R is defined to be the root of the tree. Since we have a character with an encoding of 0 (which is length 1), we have node A to represent that, which will be a leaf. X will be the sibling node of A, which has to have two children (or else we will only have 2 letters in the alphabet, contradicting $n > 2$). So X has children Y and Z. These could have children of their own, but they are irrelevant to the solution.

Let $f(n)$ be the frequency of a node n, whether that node is a leaf or internal node.

By the tree's construction, only one node is of distance 1 to the root. Intuitively, the character with the highest frequency is closest to the root. So $f(A) = p_1 < \frac{1}{3}$.

Anyway, let's work backwards. The last merge is obviously A and X to get R. Remember that the sum of the frequencies of the available nodes at any given point is 1. So $f(A) + f(X) = 1$. With $f(A) < \frac{1}{3}$, that means $f(X) > \frac{2}{3}$.

Since A is a leaf node, it cannot have been the result of a merge during the second to last step. So the second to last step must have been merging Y and Z to get X.

By how Huffman constructs a parent node's frequency, we have $f(X) = f(Y) + f(Z)$. WLOG, suppose $f(Y) \geq f(Z)$. We must have $f(Y) > \frac{1}{3}$ (because if $f(Y) \leq \frac{1}{3}$, then that with $f(Y) \geq f(Z)$ would mean $f(X) = f(Y) + f(Z)$ is not greater than $\frac{2}{3}$, contradicting prior info).

Irrespective, $f(A) < f(Y)$ and $f(Y) \geq f(Z)$. So in the second to last step, where we have nodes A, Y, and Z, the ones with the two lowest frequencies are A and Z. But here's the contradiction: Huffman is supposed to merge the nodes with the two lowest frequencies each time, so it should have merged A and Z in the second to last step. However, information earlier says it merged Y and Z in the second to last step. So this completes the solution.

§17.6 Summary

We saw how Huffman can be used to create a coding system for any given alphabet set with different frequencies. It allows us to both convert the letters into bits and obtain the original message back without conflicts. Here are the main things from this section:

- We start out with one node for each letter. The algorithm involves repeatedly taking the two nodes with the smallest frequencies and merging them in a tree. Repeat until we have a full tree built.
- Codes for a character are generated by looking at the path from the root to the character. Decoding involves using the 0s and 1s to move along the tree until we get to a character.
- Character nodes being only at leaf nodes prevents a situation where they could be two possible original messages from the same sequence of bits.

- Intuitively, optimization is shown via ABL. We objectively want to save the longer codes for less frequent letters and more frequent ones should have shorter codes.
- The formal proof by induction involves recursively thinking about the Huffman process where we merge one node and then it's an alphabet size of one less character.
- Huffman proof problems involve using probabilities, the Huffman process, and proof by contradiction.

§18 Graphs

§18.1 The Basics

> **Review 18.1.** Obviously, CIS 1600 review would be helpful here for graphs, but I will be more thorough on review in this section. This is because CIS 1600 content differs in the fall versus spring, so it's best everyone is on the same page.

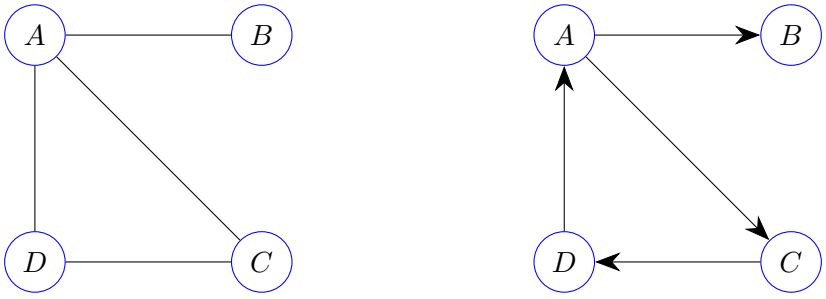
Definition 18.2 (Graph)**.** A set of nodes (usually denoted as V) where some pairs of nodes may be connected by edges (the set of edges is usually denoted as E). In this course, we will never have the same edge twice.

Now, we need to clear up some distinctions on graphs.

Definition 18.3 (Undirected graph)**.** A graph where the edges don't have directions, meaning they are all 2-way.

Definition 18.4 (Directed graph)**.** A graph where the edges have directions, meaning they are all 1-way. We can have "anti-parallel edges," which are two edges between the same pair of vertices but running in opposite directions. However, we aren't ever allowed to have two edges between the same pair of vertices in the same direction.

In the two graphs below, the left graph is undirected, while the right graph is directed.



Definition 18.5 (Unweighted Graph)**.** Graphs where there are no weights assigned to the edges

Definition 18.6 (Weighted Graph)**.** Graphs where there is a real number (representing its weight) assigned to each edge

You can also think of an unweighted graph as a weighted graph with all the edges having weight 1.

§18.2 Representations of a Graph

Let there be n vertices in the graph. There are 2 ways to represent a graph in Java that will be covered:

Definition 18.7 (Adjacency Matrix)**.** An $n \times n$ array, where $A[i][j]$ is 1 if you can get from vertex i to j and 0 otherwise.

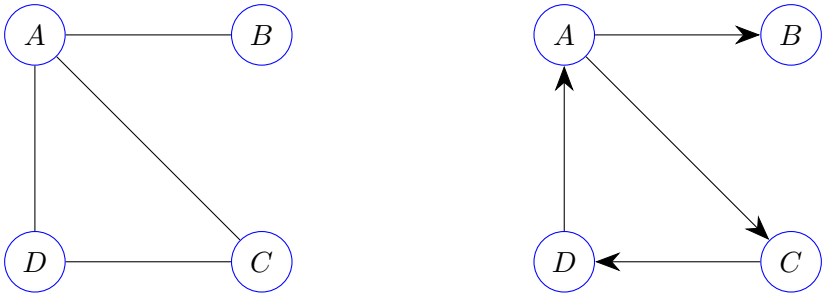
For a weighted graph, you can instead store the weight of the edge from i to j at $A[i][j]$ and have $A[i][j]$ to be null whenever we want to signify the absence of an edge from i to j.

Definition 18.8 (Adjacency List)**.** An array of size n. At $A[i]$, we have a linked list of integers containing all vertices we can get to from i (or, all edges going out of i).

For a weighted graph, the linked list can instead consist of entries that store both the neighbor j we can get to from i, as well as the edge weight from i to j.

For the most part, this class will use an adjacency list because it will help more with runtimes on graph algorithms. But some of the conceptual knowledge on when one could be better than the other is important.

But let's reuse those graphs from before and see what an adjacency matrix and adjacency list would look like.



So both graphs are unweighted and have 4 edges. Here's the matrix for the **undirected** graph on the left:

$$\begin{array}{cc} & \text{To} \\ \text{From} & \begin{array}{c} \end{array} \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \begin{array}{l} A \\ B \\ C \\ D \end{array}$$

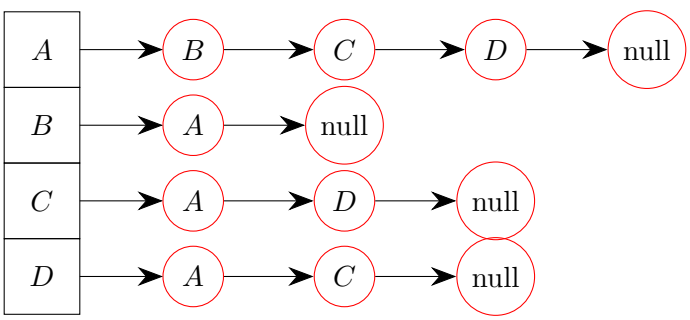
Hold on, didn't the graph have 4 edges, but there's 8 cells in the matrix equal to 1? Well, we can think of any undirected graph as a directed one, but with each edge replaced by two anti-parallel edges. So there may be an edge $i - j$, but that means both $i \to j$ and $j \to i$. That's why the number of entries equal to 1 is double the number of edges in an **undirected** graph. Also, because of that, notice when you draw a diagonal line from the top-left corner to the bottom-right corner, the matrix is symmetric about that line. That's exactly because if $i - j$ exists, you have both $i \to j$ and $j \to i$, but if that edge doesn't exist, you have neither.

For the **directed** graph, you don't have to worry about having to dissect undirected edges in to 2 directed edges:

$$\begin{array}{cc} & \text{To} \\ \text{From} & \begin{array}{c} \end{array} \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} A \\ B \\ C \\ D \end{array}$$

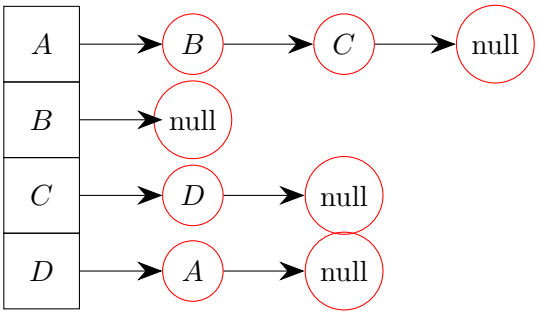
Much simpler and more intuitive.

As for adjacency lists, here's what it looks like for the **undirected** graph:



Like with the matrix, for the undirected graph, the sum of the nodes (8 of them, the null ones don't count) across all the linked lists is double that of the number of edges to signify two way traversal.

And **directed**:



Some simple distinctions between the two, if n is the number of vertices of the graph and m is the number of edges:

	Adjacency Matrix	Adjacency List
Time to lookup if a particular edge exists	$O(1)$	$O(\deg(i))$
Time to get all neighbors of a vertex i	$\Theta(n)$	$O(\deg(i))$
Space	$\Theta(n^2)$	$O(n+m)$

§18.3 Space

Let's do some space analysis. So remember that 4×4 adjacency matrix? If we want to use an adjacency matrix, any graph with 4 vertices is going to need that entire matrix to store all possibilities where they COULD be an edge and whether or not we actually HAVE said edge.

So if we have 9 vertices, we have 81 spaces in the matrix. But regardless if we have 1 edge, 15 edges, 40 edges, or 75 edges, it's going to take up the same amount of space.

On the other hand, for the adjacency list, notice we only have nodes for whenever there's an edge; if there is a lack of an edge, we just don't include it as data at all in the list.

That means if we only have 1 edge out of 81 possible, an adjacency matrix is a massive waste of space when the adjacency list would just have 1 node across all its linked lists, symbolizing the 1 edge. However, if we were to add extra edges to the graph, the adjacency matrix doesn't need more space, while the adjacency list will for extra nodes in the linked lists. So there may be a point where the adjacency list requires more space because of all the addition and so the matrix would be better. As you can see, this is a similar situation to array list versus linked list, where the former is a waste of space at first but the latter can become worse in space as we add more stuff.

Definition 18.9 (Sparse graph)**.** A graph without many edges relative to $|V|^2$ (where V is the set of vertices in a graph), meaning an adjacency list is more space efficient.

Definition 18.10 (Dense graph)**.** A graph with many edges relative to $|V|^2$ (where V is the set of vertices in a graph), meaning an adjacency matrix is more space efficient.

Example 18.11 (Recitation)

Suppose a matrix entry requires 1 byte, a vertex index requires 2 bytes, and a pointer requires 5 bytes. A directed graph G has 7 vertices and 19 edges. How many bytes is required to store it as an adjacency matrix and as an adjacency list? What if G were undirected instead?

Let's do directed first. So we have $|V| = 7$ and $|E| = 19$. Remember that the adjacency matrix is always going to have $|V|^2$ cells, no matter how many edges we have. A matrix entry is 1 byte, so the adjacency matrix is $1 \cdot |V|^2 = 49$ bytes.

As for the adjacency list, there's space from the array and space from the linked lists. The array is just $|V|$ pointers with each pointer leading to the head of the respective linked lists. A pointer is 5 bytes, so that accounts for $5 \cdot |V|$ bytes there. Remember the number of nodes we have across all linked lists

in the adjacency list is equal to the number of edges we have. A node in a linked list requires both a vertex index (to tell us the neighbor in question) and a pointer to the next node in the list (even nodes at the end of their linked list, where we just have null). So a node in total requires $2 + 5 = 7$ bytes. And we have as many nodes as edges, so that accounts for $7 \cdot |E|$ bytes. The total for the adjacency list is $5 \cdot |V| + 7 \cdot |E| = 168$ bytes.

Now for undirected. Remember we have to essentially double all the edges to turn the directed edge into 2 one-direction transmissions. So we would say $|E| = 2 \cdot 19 = 38$ in this context. The adjacency matrix is unaffected, still having 49 bytes, but the adjacency list now requires a hefty $5 \cdot |V| + 7 \cdot |E| = 301$ bytes in comparison.

§18.4 More Definitions

Lemma 18.12 (Handshaking Lemma)

For any graph with V as its set of vertices and m being the number of edges,

$$\sum_{a \in V} \deg(a) = 2m.$$

In plain English, the left hand side is saying that we take every a in the set V and sum the degrees up. So this lemma is saying the sum of the degrees of all the vertices in the graph is twice the number of edges in the graph. We can also use this to say the sum of the degrees of all vertices in the graph is $O(m)$.

Definitions applying for UNDIRECTED graphs:

Definition 18.13 (Connected vertices)**.** Two vertices a and b are connected in an **undirected** graph if there exists a path from a to b.

Definition 18.14 (Connected component)**.** A set of vertices in an **undirected** graph such that any two vertices in the set are connected, and the component is maximal (meaning you can't add another vertex to the component so that all vertices in the component are pairwise connected).

Definitions applying for DIRECTED graphs:

Definition 18.15 (Reachability)**.** We say v is reachable from u in a **directed** graph if there exists a directed path from u to v. We note reachability from u to v as $u \twoheadrightarrow v$.

Definition 18.16 (Strong connectivity)**.** An extension of reachability is strong connectivity. We say u and v are strongly connected in a **directed** graph when u and v are both reachable from each other.

Definition 18.17 (Strongly connected component)**.** Similar to a connected component in an undirected graph. In a directed graph, any two vertices in a strongly connected component are strongly connected, and the component is maximal (meaning you can't add another vertex to the component so that all vertices in the component are pairwise strongly connected).

Definition 18.18 (Directed cycle)**.** Similar to a cycle in an undirected graph: a directed walk with no repeating vertices except the first and last ones. Self-loops count as directed cycles. Two edges between the same two vertices in opposite directions (anti-parallel edges) also count as directed cycles.

Definition 18.19 (Directed Acyclic Graph (DAG))**.** A directed graph without any cycles

Definition 18.20 (Outdegree)**.** As noted by $\text{out}(u)$, the outdegree of a vertex u is the number of directed edges coming out of u

Definition 18.21 (Indegree)**.** As noted by $\text{in}(u)$, the indegree of a vertex u is the number of directed edges coming in to u

Definition 18.22 (Sink)**.** A vertex with outdegree 0. Meaning all edges connected to it point in to it.

Definition 18.23 (Source)**.** A vertex with indegree 0. Meaning all edges connected to it point out of it.

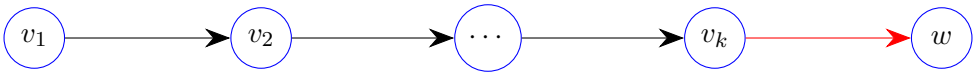
Lemma 18.24

All DAGs have at least one source and one sink.

Proof. Let's use the Maximal Path technique back from CIS 1600. Let $v_1 \to v_2 \to \cdots \to v_k$ be a maximal path in the DAG.

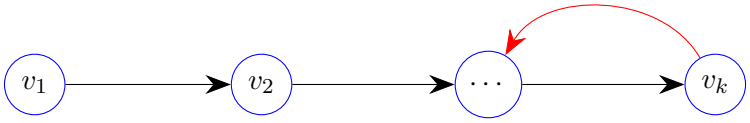
I claim that v_k is a sink. Suppose for the sake of contradiction it wasn't. Then, there would need to be some directed edge going out of v_k and to some other vertex. Two cases, either this vertex is outside the maximal path or inside the maximal path.

In the first case, $v_k \to w$, where w isn't already in the path.



Since w isn't already in the path, we can extend the maximal path to have $v_1 \to v_2 \to \cdots \to v_k \to w$. But this contradicts the definition of a maximal path (a path that cannot be extended at all). So this case leads to a contradiction.

Let's try the second case. So v_k has an edge pointing out of it that goes to some vertex already in the path.



This creates a cycle though, contradicting the definition of a DAG. So v_k cannot have edges going out of it, so it is a sink, meaning the graph has a sink.

A similar proof pattern holds for source, where if there were an edge going in to v_1, similar contradictions occur in the two cases. □

§18.5 Summary

- We did review on graphs. The Rajiv students who might not have had as much exposure to directed graphs should be caught up to pace on the vocabulary, but might need some examples later for it to sink in.
- We saw how we can either represent the graph as an adjacency matrix or adjacency list.
- There are different situations where you would want to use one or the other in terms of space. Matrices waste a lot of space if they don't have many edges, but the pointers required in an adjacency list can add up the more edges we have.
- However, for the most part, we're not going to care too much about space in the graph algorithms we do, so adjacency lists will be better because the things we do in the graph algorithms will be more useful in runtime.
- In particular, the time advantage of the adjacency list in accessing all neighbors of a particular vertex (instead of having to go through the entire row of an adjacency matrix, wasting time to check which are 0 and which aren't) is the key benefactor.

§19 Breadth First Search

§19.1 Introduction

So we have a graph. Why don't we explore it? Check every nook and cranny and see what we can find.

We'll get our first taste of a graph algorithm. It turns out that these graph algorithms can have other applications beyond the obvious.

Here, we'll do Breadth First Search, or BFS for short. The objective is to pick some starting node S and find all nodes that can be reached from the starting node.

§19.2 The Algorithm

First, establish a starting node S.

Suppose the graph has n nodes. Create two arrays of size n, one being a boolean array called "discovered" (with each entry initially set to false), and a node array called "parent" (with each entry set to null).

Have a queue and add S to it. In "discovered" for S, set it to be true.

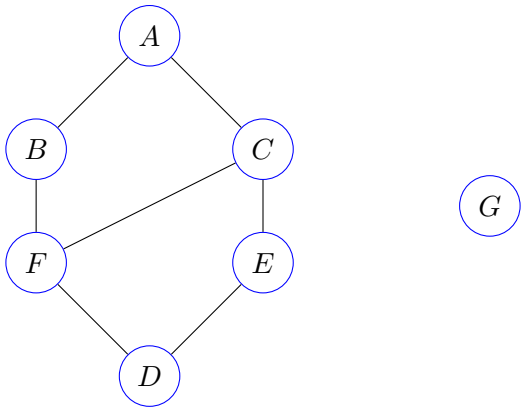
Run the following while the queue is not empty:

- Dequeue the queue and suppose v is the vertex we get from doing that
- Check all neighbors of v in the adjacency list. Suppose u is some neighbor of v. If we check the discovered array and see u is discovered already, ignore it. Otherwise, do the following:
  - Set the discovered status of u to be true
  - Add u to the queue
  - Set the parent of u to be v in the parent array

In plain English, basically we search the starting node's neighbors and add them all to the queue. Then, one by one, we look at each of those neighbors in the queue and add their neighbors (only ones we didn't discover yet). We keep adding neighbors upon neighbors upon neighbors of nodes to the queue until we discover everything we can. The parent array is optional; it is more of a way to keep track of the node from which we first discovered some node, which has applications I'll get to later. But the discovered array is not optional. We'll see when we do the algorithm in action.

§19.3 In Action

Let's do BFS on the following graph with A as the starting node: (notice that G is a different connected component than A)



Let's do the initial set up:

Node	A	B	C	D	E	F	G
Discovered	True	False	False	False	False	False	False
Parent	null	null	null	null	null	null	null

And the queue so far is just A. Now, the loop begins, where the action unfolds. We take A out of the queue. So now, we check all neighbors of A, which are B and C. Neither are discovered yet, so we set them to be discovered, add them to the queue, and their parent node to be A. Here's the new table:

Node	A	B	C	D	E	F	G
Discovered	True	True	True	False	False	False	False
Parent	null	A	A	null	null	null	null

The queue is B, C. Again, dequeue, which takes B out of the queue. We check all neighbors of B, which are A and F. We ignore A as we discovered it already. But we add F to the queue, since we didn't discover it yet and set its parent to be B. Here's the new table:

Node	A	B	C	D	E	F	G
Discovered	True	True	True	False	False	True	False
Parent	null	A	A	null	null	B	null

The queue is C, F.

Warning 19.1. Do you now see why the discovered array is essential? Suppose we didn't have it. We would check A's neighbors and add B to the queue. But upon eventually getting to B in the queue checking B's neighbors, we would add A back. Then, we eventually get to A in the queue, check its neighbors, and add B back. So we get stuck in an infinite loop. Also, there's literally no point in processing the same vertex more than once anyway.

So we dequeue C and check its neighbors. Only E is undiscovered among its neighbors. New table:

Node	A	B	C	D	E	F	G
Discovered	True	True	True	False	True	True	False
Parent	null	A	A	null	C	B	null

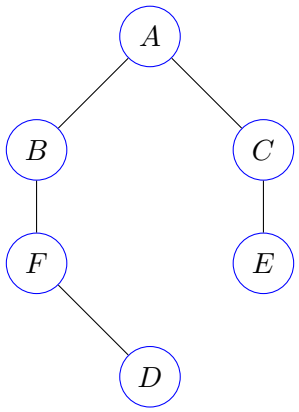
The queue is F, E. Now, we dequeue F and check its neighbors. Only new neighbor is D. New table:

Node	A	B	C	D	E	F	G
Discovered	True	True	True	True	True	True	False
Parent	null	A	A	F	C	B	null

The queue is E, D. Checking both nodes, we discovered all their neighbors already. So we remove both from the queue. Queue is empty, so we stop, and the above table is the final result. Notice we never discovered G, and it was in a different connected component than A. Maybe there is causation between those two things? We'll see in the proof of correctness.

Also, the parent of A is null because that's where the source of the BFS is and wasn't discovered by some predecessor node.

But more important data with the parent pointers is, we can use this to create what is known as a "BFS tree." So when BFS finishes, we look at the parent array. We see B has parent A, so we create a tree with $A - B$. We see C has parent A, too, so we have $A - C$. We do this for every parent-node relationship in the parent array but do not draw any other edges. That forms our BFS tree:



Note that G isn't in the tree because it wasn't reached.

As for the reason why it forms a tree along the connected component, recall the definition of a tree being connected and acyclic. As we'll prove, we get everything in the connected component from the BFS, so we retain a sequence of edges that allows us to get from any vertex in the same connected component to the source. As for acyclic, the intuition is we don't add an edge to the BFS tree when we go across an edge only to see we previously discovered the vertex on the other end, so that prevents cycles from forming.

§19.4 Proof of Correctness

So if we run BFS on a graph $\mathcal{G}$ with starting node S, we want to prove that, whether the graph is directed or undirected, the BFS tree contains all vertices u such that u is reachable from S (and no other vertices). I guess treat undirected graphs as directed ones for now, with anti-parallel edges.

So the idea is, if u is reachable from S, then we know there exists a path from S to u, or $S \to v_1 \to v_2 \to \cdots \to v_k \to u$ for some intermediate vertices $v_1, v_2, \cdots, v_k$ to get to u (or we might just be able to get straight from S to u. But clearly, all those intermediate vertices along the way are reachable from S, too. With how BFS works, by searching the neighbors, and then the neighbors of those neighbors, and so on, we will eventually get to u by some path from S to u. We always check every edge of each vertex in the queue, so that ensures we don't miss out on a vertex that is reachable from S. That's the intuition behind BFS getting everything reachable from S.

And if some arbitrary vertex w is NOT reachable from S, then there is no path from S to w. Because BFS only uses the edges to traverse and doesn't jump off connected components, it will never catch w. So we don't accidentally catch something that isn't reachable from S.

In particular, for an undirected graph, we get exactly the connected component containing S.

One more thing, to show that the queue will eventually become empty and we don't run in to infinite loop troubles. Once we discover a vertex, we mark it as discovered. In our check to see whether we add a vertex to the queue, we don't add it if it was already discovered. So once a vertex leaves the queue, it never comes back. There is a finite number of vertices (which is n), so the queue will always empty out eventually.

Warning 19.2. Do NOT use BFS when considering strongly connected components in a DIRECTED graph. The reason it works for undirected graphs is because if you can get from u to v, that automatically means you can get from v to u. But with directed graphs, just because you can get from u to v, that doesn't necessarily mean you can get from v to u. Don't worry, we'll have an algorithm later that will get you connected components in a directed graph.

Advice 19.3 — This result shows a helpful applications of BFS in getting all vertices that are reachable from node S, whether the graph is directed or undirected.

§19.5 Runtime

Just to clear something out of the way: we did use a queue for the vertices and it could delve into amortized analysis. However, we know that there are exactly n vertices, so we know an upper bound on the size of an array list. This avoids having to resize past size n, so we can pretend we never have to resize at all and just use an array list of size n. This makes enqueue and dequeue regular $O(1)$ for BFS.

The work for BFS comes from two things:

- Enqueuing and dequeuing vertices
- Check all the neighbors of some vertex

Keep in mind, we might not necessarily get to all vertices (like if they're in a different connected component). So worst case, we assume that we do get to all of them.

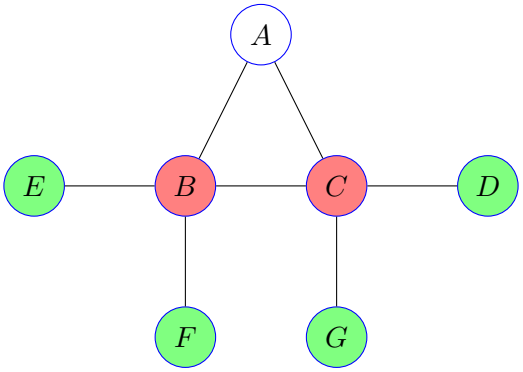
The first bullet point is clearly $O(n)$. For a vertex i, checking all its neighbors is $O(\deg(i))$ work. But summing across all vertices in the graph, this is

$$\sum_{i \in V} \deg(i) = O(m),$$

by Handshaking Lemma. So we say BFS is $O(n+m)$. Can we simplify this further, like trying to get m in terms of n? We could, but we usually don't and leave it as $O(n+m)$. Remember the number of edges can vary wildly relative to the total number of possible edges given by the number of vertices (remember sparse vs dense graphs).

§19.6 Properties of BFS

First, we need to prove a general statement about graphs. Define the "distance" from u to v to be the length of the shortest path in the graph from u to v (provided v is reachable from u). Consider the graph below and distances measured from A:



The graph is simple enough to work with that the distances from A to each node should be easy to see. So I highlighted the vertices of distance 1 from A in a light red and the vertices of distance 2 from A in a light green.

Let $D(u,v)$ be the distance from u to v. I claim that if x and y share an edge in the graph, then $|D(u,x) - D(u,y)| \leq 1$ in any graph. This will be key intuition for an application of BFS, but here's the proof, arguing through contradiction. Assume we did have vertices x and y sharing an edge with $|D(u,x) - D(u,y)| > 1$. For brevity, we can without loss of generality, assume that $D(u,y) > D(u,x)$ and say $D(u,x) = d$. Then, $D(u,y) = d+k$ for some positive integer $k \geq 2$. The definition of distance being the minimum means there should not be a path from u to y with length less than $d+k$. Using $D(u,x) = d$, there exists a path of length d from u to x. Since x and y share an edge, we can add this edge to path to get a path of length $d+1$ going from u to y. But this contradicts that there should not be a path from u to y with length less than $d+k$ (since $k \geq 2$). This completes the proof.

So for any two vertices that share an edge, their distances are either the same (like $B - C$ above), or they differ by exactly 1 (like $B - F$ above).

Now, for the magic. Define the kth layer of the graph to be the set of vertices that are a distance k from the source vertex. So if we let L_k be the kth layer, in the graph above, we would say L_0 has A, L_1 has B and C, and L_2 has E, F, G, and D.

Consider the order in which vertices were inserted into the queue. I'm going to show that first, we have all vertices in layer L_0, then all vertices in layer L_1, then all vertices in L_2, and so on. The idea is through induction.

The base case is the starting node S being inserted into the queue first, which obviously, S has a distance of 0 from itself and no other vertices have a distance of 0 from S. So we have everything in L_0 first in the queue. When we explore all neighbors of S, those are all the vertices that are of distance 1 from S (to be distance 1 from S, you clearly have to be a neighbor of S), or the vertices in L_1. So all of L_1 is inserted in to the queue in some order.

Now, let's consider what will happen when we take all vertices in L_1 and check all their neighbors. Well, because the neighbors share an edge with some vertex in L_1 (which has distance 1), these neighbors of vertices in L_1 have either distance 0, 1, or 2 by that claim we proved earlier. We already got everything of distance 0 and 1 previously, so we're not adding them back to the queue. We are adding the stuff we find that is in L_2 because we haven't discovered it yet. Also, we get everything in L_2, not just a select few from L_2. Why? Because u being in L_2 means there is a path of length 2 from the source. So some vertex in L_1 comes before u in the path. Since we check every vertex in L_1 and BFS checks all neighbors, u will get caught no matter what.

Also, remember the properties of a queue. The stuff in L_1 got enqueued before anything in L_2 got enqueued. So we always dequeue and process the stuff in L_1 before the stuff in L_2.

So now, we've caught everything in L_0, L_1, and L_2. Now to process the vertices in L_2. Neighbors of vertices in L_2 must be of distance 1, 2, or 3 (again by the claim). L_1 and L_2 have been completely covered so far, so the only new stuff we're adding is the stuff in L_3. Again, we get everything in L_3 because we check everything in L_2 and all its neighbors.

Again, the properties of a queue are relevant where stuff in L_2 gets priority in being processed over stuff in L_3.

You can see the induction. The claim from earlier is what holds this induction in place and ensures that, if we check the neighbors of some vertex in L_3, we won't end up with stuff in L_5 or L_6 or beyond. Also, the queue properties ensure we don't process vertices in a further layer than vertices in a closer layer. Because if we processed something in L_4 before something in L_3, that would break the induction with the order.

§19.7 Applications of BFS

So we went over getting all reachable vertices from some vertex S back in the proof of correctness. But I'd like to turn attention to the layer intuition. Notice how whenever we check $u \to v$ and we see v is undiscovered, v is in the next layer after u. This is a general claim you can make. Because we found that we always discover stuff in L_0 first, then stuff in L_1, then stuff in L_2. So if we check $u \to v$ and see v is discovered already, it's because v is in the same layer as u or the one before it. So that means if it's undiscovered, v comes in the next layer after u.

This means we can modify BFS to accurately keep track of the layer of each node. Essentially, we want to initially set the source vertex's layer to be 0. Then, whenever we check $u \to v$ and see v is undiscovered and u is in layer k, we can conclude v is in layer $k+1$. This layer system has an application and finding the shortest distance from S to all points reachable from S.

This layer intuition also extends to another application in checking whether a graph is bipartite (or if you recall, whether it is 2-colorable). If you think about it, say our two colors are red and blue and without loss of generality, we paint all of L_0 red. Well, everything in L_1 is connected to some vertex in L_0, so we need to have all of L_1 be blue. And everything in L_2 is connected to some vertex in L_1, so we need to have all L_2 be red. So we are forced to alternate colors in the layers, but there is the issue this might not work because two vertices in the same layer share a vertex. However, if that doesn't happen, the coloring works. Whether or not this happens can be checked by keeping track of layers. When we check $u \to v$ and we see that v is not only discovered, but both are in the same layer, it's not bipartite. That's the intuition, but you should do some examples for yourself if you are not convinced.
So here's our list of applications:

- Getting everything that is reachable from some vertex S
- Finding shortest distance between two points in an unweighted graph
- Checking if a graph is bipartite

> **Warning 19.4.** The shortest distance does NOT work if the graph is weighted! We'll see an algorithm later that does do shortest distances in a weighted graph. However, if the graph is not weighted, it's more ideal to use BFS for shortest distance because it turns out this is faster than that other algorithm.

§19.8 Summary

- We learned our first graph traversal: BFS. Basically we repeatedly check neighbors of neighbors and this ensures we eventually get everything reachable from some starting node.
- We have to keep track of what is discovered so far to prevent redundancy and infinite loops.
- We defined a layer, which meant a set of vertices that are some specific distance away from the source.

- We saw that any two vertices sharing an edge do not have their layer numbers differ by more than 1. With some inductive work, we used that to show how we always enqueue L_0, then L_1, then L_2, and so on in that specific order.
- The layers also give us distances from the starting vertex to the all others we encounter.

§20 Depth First Search

§20.1 Introduction

Oh boy, buckle up because this section is a doozy. So we learned our first graph traversal, BFS, which has some nice properties in distance. But that isn't the only graph traversal, right? Let's explore a graph algorithm with other advantages and applications!

I'll admit I might be rambly throughout this section, but I'd rather all the details be there than not. This graph algorithm is super important in later ones, so I wanted to at least put all the intuition out there so those later algorithms are easier to understand.

§20.2 The Algorithm

First, we need some setup. We set an integer variable called "time" equal to 0. We'll see what this does later. We also create a stack.

We also create 4 arrays of size n, called "color," "parent," "discovery time," and "finish time." Set everything in the color array to white initially and everything in the parent array to null.

We need to define this sub-algorithm first called "DFS Visit" that takes in a graph G and a starting node v:

- First, we increment time by 1. Then, we push v to the stack, set the color of v to gray, and put the current time as the discovery time of v.
- Do this loop while the stack is non-empty:
  - Peek at vertex u at the top of the stack. Check if it has a white neighbor.
    - If u does have a white neighbor w, increment time by 1, push w to the stack, set w's color to gray, set w's parent to u, and set w's discovery time to the current time value.
    - Otherwise, if u doesn't have a white neighbor, increment time by 1, pop it from the stack, set u's color to black, and set u's finish time to the current time value.

Then, there's the actual DFS algorithm:

- Simply go through the color array.
- Whenever you encounter a vertex that is white, call DFS Visit on it.

Holy smokes, there's a lot of moving parts here. But let me explain to you what's going on in plain English. The basics of DFS visit is, we start at a node v. From v, we basically repeatedly check if there's an unexplored vertex and if we find one, we go there. We keep doing this until we reach a point where there's no undiscovered vertex we can go to. At that point, we backtrack and check if there are vertices along the way back that haven't been discovered. If we do find such an undiscovered vertex, we again, try to keep going to undiscovered vertices for as long as possible until we have to backtrack. Eventually, we'll cover every nook and cranny and discover all nodes reachable from v.

Now, the color system is basically a way of having a record of the status of some vertex v. White means we didn't discover v yet. This is equivalent to v not having been inserted in the stack yet. Gray means we discovered v, but we're currently exploring some route past v. This is equivalent to v currently being in the stack. Black means we've discovered the entirety of all paths beyond v and that we are permanently removing it from the stack.

With the color system explained, we can understand the actual DFS algorithm that calls visit multiple times. We basically go through the color array and when we discover a white vertex, we call DFS on it. This will discover some other vertices in the process, changing their status to gray and then black. And

then we keep going through the array to see if there were vertices that are still white even after that first DFS visit (presumably because they weren't reachable from that first source vertex). Rinse and repeat until we get through the array. This ensures a series of DFS visits cover all n vertices in one pass through the color array.

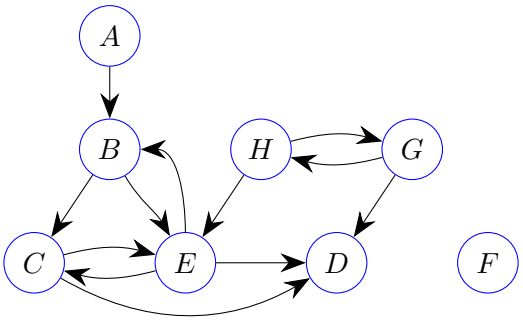
Parent system is the same idea as in BFS.

The time system is new though. Essentially, we only increment and record times when we first discover a vertex and when we're finished with it. In other words, this only happens when we add or remove something to the stack. As you will see, this time system will have some super nice properties that can be used to learn more about the graph, particular in directed graphs.

All this alphabet soup should make more sense after an example.

§20.3 In Action

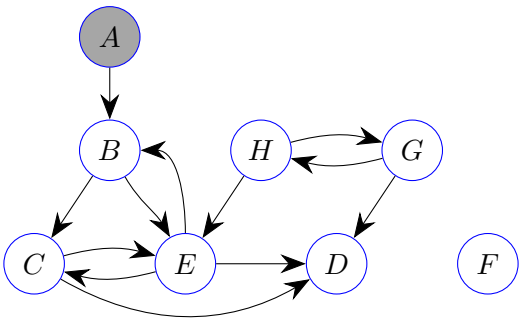
Consider the following graph, as well as the setup arrays. Time is currently 0.



Node	A	B	C	D	E	F	G	H
Color	White	White	White	White	White	White	White	White
Parent	null	null	null	null	null	null	null	null
Discovery Time								
Finish Time								

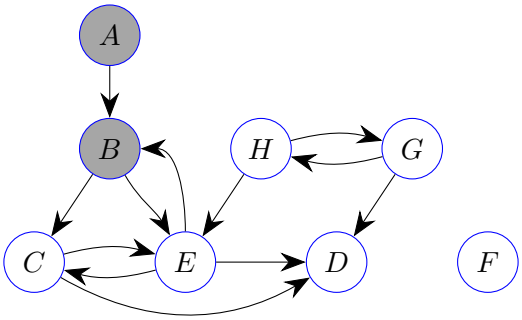
Okay, first thing's first. A is white, so we do DFS visit on that. Time is currently at 0. Keep in mind throughout the process that the stack is basically the set of vertices that are currently gray.

So we change time to 1, set A's color to gray, and log 1 as the discovery time of A. Add A to the stack, too.



Node	A	B	C	D	E	F	G	H
Color	Gray	White	White	White	White	White	White	White
Parent	null	null	null	null	null	null	null	null
Discovery Time	1							
Finish Time								

The stack is now just A. Okay, now we check for white neighbors of A. We see B. So set B's parent to A, set B to gray, and set B's discovery time to 1. Add B to the stack, too.

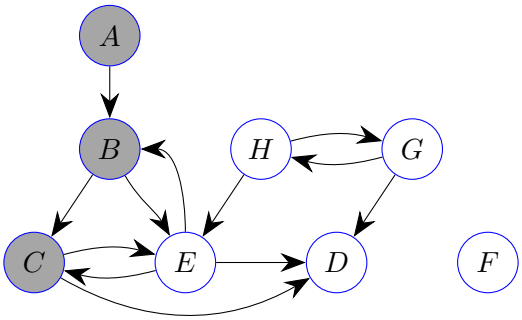


Node	A	B	C	D	E	F	G	H
Color	Gray	Gray	White	White	White	White	White	White
Parent	null	A	null	null	null	null	null	null
Discovery Time	1	2						
Finish Time								

The stack is now A, B. Any white neighbors of B? There's two: C and E.

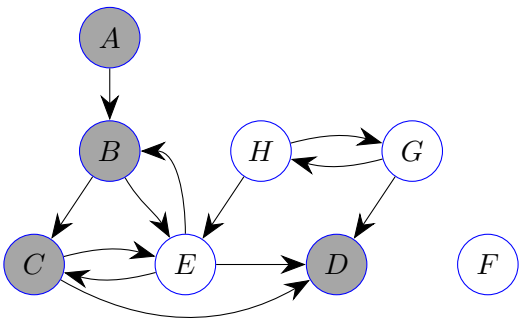
Warning 20.1. ALWAYS ask them what gets priority if there's 2 or more white neighbors. From recitation and class, we'd usually prioritize what comes first in the alphabet. So here, C gets priority over E. But you want to check because a different priority system will make your discovery and finish times different.

Warning out of the way, we visit C. So set C's parent to B, its color to gray, and discovery time to 3. Add C to the stack.



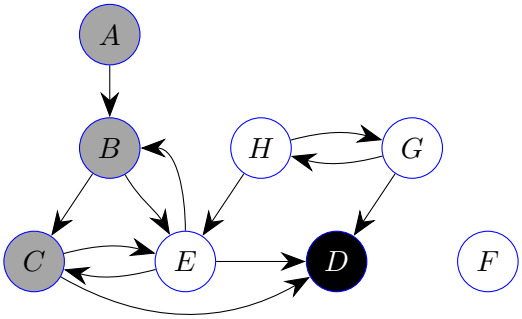
Node	A	B	C	D	E	F	G	H
Color	Gray	Gray	Gray	White	White	White	White	White
Parent	null	A	B	null	null	null	null	null
Discovery Time	1	2	3					
Finish Time								

Now the stack is A, B, C. Check neighbors of C, prioritize D over E, same old drill in editing the table:



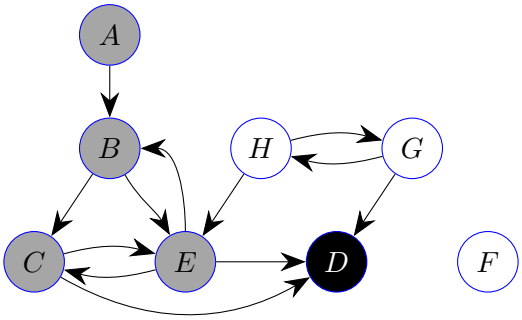
Node	A	B	C	D	E	F	G	H
Color	Gray	Gray	Gray	Gray	White	White	White	White
Parent	null	A	B	C	null	null	null	null
Discovery Time	1	2	3	4				
Finish Time								

Stack is A, B, C, D. Also, we're at a dead end now with nowhere to go and no white vertices to get to. When that happens, we finish D and remove it from the stack. We also color D black and log the finish time:



Node	A	B	C	D	E	F	G	H
Color	Gray	Gray	Gray	Black	White	White	White	White
Parent	null	A	B	C	null	null	null	null
Discovery Time	1	2	3	4				
Finish Time				5				

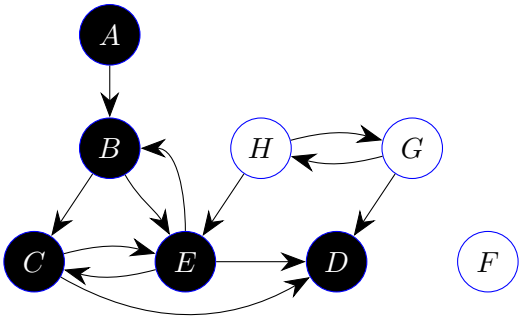
The stack now is A, B, C. Check if we have any white neighbors of C, which we do: E.



Node	A	B	C	D	E	F	G	H
Color	Gray	Gray	Gray	Black	Gray	White	White	White
Parent	null	A	B	C	C	null	null	null
Discovery Time	1	2	3	4	6			
Finish Time				5				

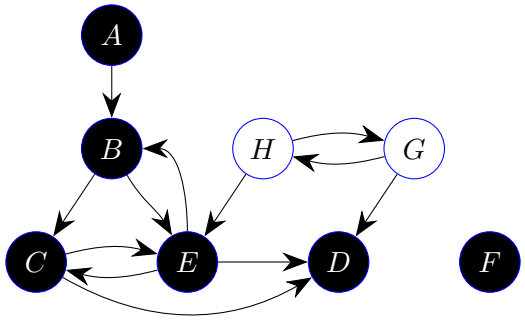
Note that we increased time when we removed D from the stack and when we found E but not when we went back to C. That doesn't count for time incrementation.

Anyway, E is a dead end, so we color it black and log the finish time. The stack becomes A, B, C. You can work it out yourself, but I'll just tell you that C completes when you revisit it because there's no more white neighbors, then B completes, then A completes:



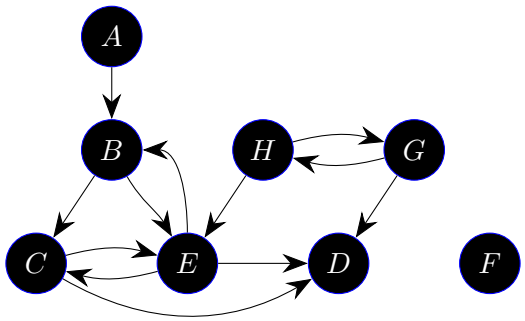
Node	A	B	C	D	E	F	G	H
Color	Black	Black	Black	Black	Black	White	White	White
Parent	null	A	B	C	C	null	null	null
Discovery Time	1	2	3	4	6			
Finish Time	10	9	8	5	7			

Remember that this is all from doing DFS Visit on A. We still need to check through the rest of the color array to see if there's any white vertices. B, C, D, E? Nope. F is the next white vertex, so we call DFS Visit on it. It'll just discover itself and then finish because there's no way out of F and thus, no white neighbors.



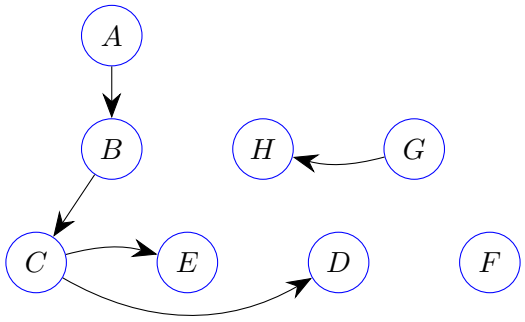
Node	A	B	C	D	E	F	G	H
Color	Black	Black	Black	Black	Black	Black	White	White
Parent	null	A	B	C	C	null	null	null
Discovery Time	1	2	3	4	6	11		
Finish Time	10	9	8	5	7	12		

And then we see G is white so we discover that, then go to H and discover that. H doesn't have any white neighbors, so we finish H. Then, we go back to G and finish G.



Node	A	B	C	D	E	F	G	H
Color	Black	Black	Black	Black	Black	Black	Black	Black
Parent	null	A	B	C	C	null	null	G
Discovery Time	1	2	3	4	6	11	13	14
Finish Time	10	9	8	5	7	12	16	15

All done, nothing is white anymore. Just like BFS, we can create a forest (not necessarily a tree because there could be multiple connected components that are each trees themselves) out of the parent array. We'll have $u \to v$ in the DFS forest if u is a parent of v.



This DFS forest is going to lead to useful properties. We'll see later.

§20.4 Proof of Correctness

It's a similar idea to BFS on how an arbitrary tree T with root r in the DFS forest will always have everything that is reachable from r. The idea with DFS is that it will always thoroughly check everything, both vertices and edges, just like BFS. It also never redundantly checks something: once something has been discovered, it doesn't get discovered again. This is done by only inserting a vertex in the stack if it's white (undiscovered).

§20.5 Runtime

This is $O(n+m)$ for the same reasons that BFS has this runtime. We don't change the amount of work we do; we merely rearrange the order in which it is done. While we use a stack instead of a queue, adding and removing the n vertices in that is still $O(n)$. We always check every edge, just not all of them from the same vertex one after another. While we often check one edge coming out of u and not check the next edge coming out of u until later, we always do it eventually. So that is still $O(m)$.

§20.6 DFS Forest and Edge Classification

First, we need to define a term relating to the DFS forest:

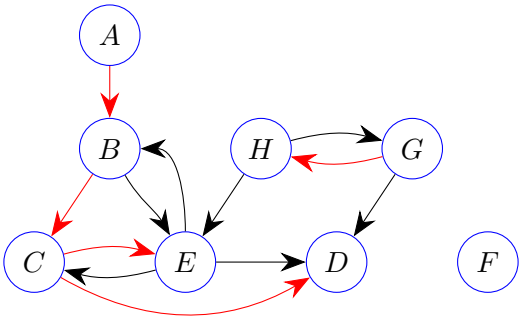
Definition 20.2 (Ancestor/Descendant)**.** Suppose in the DFS forest, you can get from u to v. Then, v is a descendant of u, and u is an ancestor of v. In the DFS forest we made above, E would be a descendant of A, B would be an ancestor of D, but D and E don't have an ancestor/descendant relationship.

Given a DFS forest along with the graph we ran DFS on, we have 4 different categories for the edges of the original graph:

- Tree edge - an edge in G that is also in the DFS forest
- Back edge - an edge from $u \to v$, where u is a descendant of v
- Forward edge - a non-tree edge from $u \to v$, where u is an ancestor of v
- Cross edge - all other edges

Warning 20.3. Depending on the priority system (like something other than alphabetical order), you may get a different DFS forest. Also, we technically don't need to scan through the array in order to find the next white vertex to do a DFS visit on; we could just keep choosing arbitrary white vertices each time we want to do a DFS visit, as long as we get all vertices in the end. That can also result in a different DFS forest. So an edge may be a tree edge in the graph for some DFS forest but the same edge may be a cross edge for some other DFS forest of the same graph.

Let's do the demonstration with the DFS we did. The edges of the graph that were also present in the DFS forest are highlighted in red. Those not present remain black:



So all the red edges are tree edges by our definition. To consider the remaining edges, we can clearly see by the DFS forest that B is an ancestor of E. So $B \to E$ would be a forward edge, while $E \to B$ is a back edge. $E \to C$ is also a back edge. $E \to D$ would be an example of a cross edge because E and D don't have any ancestor/descendant relationship (which disqualifies it from all 3 of tree, back, and forward edge). $H \to E$ is similarly a cross edge. $H \to G$ is a back edge.

Advice 20.4 — They may briefly ask you about edge classifications on an **undirected** graph instead of a directed one. Just know, an undirected graph only has tree and back edges, never any forward or cross edges.

Lemma 20.5

x is a descendant of y in the DFS forest if and only if y is gray when x is discovered

Proof. This is an if and only if, so we have to prove both directions. Proof is mostly meant to be intuitive (it may be faulty, so take with a grain of salt while reading it).

First, let's prove if y is gray when x is discovered, then x is a descendant of y in the DFS forest.

So y being gray means it's still in the stack when x is discovered. That means we're in the middle of exploring all possible routes with unexplored vertices coming out of y and backtracking if necessary. We know this is x being discovered for the first time and not "we're ignoring x because we discovered it already." So there's definitely a path leading from y to x in the graph that DFS used (otherwise, how did it even reach x in the first place). Now, DFS may sidetrack for some (maybe all) of the vertices along the way from y to x, but it will always backtrack and discover those nodes along the path for the first time, creating those parent pointers for when we make the DFS forest. So that path from y to x is in the DFS forest, meaning x is a descendant of y, by definition of descendant.

Now, we need to prove if x is a descendant of y in the DFS forest, then y is gray when x is discovered.

So if x is a descendant of y, we have a bunch of tree edges in the DFS forest leading from y to x, meaning DFS used this path in discovering new vertices. More importantly, each vertex on this path, when it was discovered for the first time, DFS must have came from the prior vertex in the path (otherwise, the vertex would have been discovered coming from some other vertex, which changes the DFS forest). Again, DFS may sidetrack along the path from y to x, but we know it makes progress along this path eventually to reach x (because of DFS's thorough nature). So we know y can't be finished before we reach x because y doesn't finish until we've covered every nook and cranny beyond y. So that's y has to be gray when x is discovered. □

§20.7 Parentheses Theorem

First, define $d[u]$ to be the discovery time of some vertex u and $f[u]$ to be its finish time.

Theorem 20.6 (Parentheses Theorem)

In any DFS, whenever we have two vertices x and y such that $d[x] < d[y]$, we have that either

- $d[x] < d[y] < f[y] < f[x]$ (occurs exactly when y is a descendant of x)
- $d[x] < f[x] < d[y] < f[y]$ (occurs otherwise)

Furthermore, $d[x] < d[y] < f[x] < f[y]$ can never happen

Proof. The easiest way to see this is through intuition of DFS and a stack. So we always define x and y so that x is discovered before y.

Let's suppose y is a descendant of x in the DFS forest. By the whole arrows in the DFS forest intuition, that means, as we were exploring undiscovered nodes beyond x, we came across y and so added it to the stack. But x must have still been in the stack when we added y because we don't remove x from the stack until we've not only return to x but we are sure we've explored everything beyond x.

Working from $d[x] < d[y]$, we obviously know any node has to finish after it's discovered, so $d[y] < f[y]$. Again, y was added to the stack while x was still in there and the LIFO intuition of a stack means you can't remove (or finish) x without removing y beforehand. So $f[y] < f[x]$. Putting it all together implies $d[x] < d[y] < f[y] < f[x]$, for the case when y is a descendant of x in the DFS forest.

As for when they don't have ancestor/descendant relationship, that means we tried to explore all we could beyond x, but didn't manage to find y. So when x finishes (the point where we're sure we

got everything beyond x), we never even added y yet. So $f[x] < d[y]$. Again, we know any node has to finish before it's discovered, so $d[x] < f[x]$ and $d[y] < f[y]$. Putting everything together means $d[x] < f[x] < d[y] < f[y]$. □

Usually, order of discovery times is not going to be very important, since the flexibility of DFS means you can just call the first visit on whatever node you want. But finish times will be super important in upcoming results because you can't so easily manipulate them.

§20.8 White Path Theorem

Theorem 20.7

y is a descendant of x if and only if at $d[x]$, there was a path consisting entirely of white vertices from x to y

Proof. Okay, this is an if and only if, so it will take some work to prove both directions. Trust me though, proving this is a reward because it's a useful theorem that will be helpful to cite whenever applicable.

First, we prove that if y is a descendant of x, then at $d[x]$, there was a path consisting entirely of white vertices from x to y. Let's argue by contradiction. Suppose that y is a descendant of x, but there is no path that consists entirely of white vertices from x to y. So we're at x for the first time. DFS wants to explore everything beyond x that is undiscovered. However, DFS only moves to a new vertex if it's undiscovered (or white). Since there are no paths consisting entirely of white vertices from x to y, DFS will never reach y from x because if DFS is at u and it finds a non-white neighbor of u, DFS ignores it.

So DFS will explore beyond x, only to find it never was able to get to y. So y isn't a descendant of x. But wait! That contradicts our initial assumption of y is a descendant of x. So we reached a contradiction, proving one direction.

Now, we prove the other: if at $d[x]$, there was a path consisting entirely of white vertices from x to y, then y is a descendant of x.

This can mostly be seen through the intuition on how DFS works. The white path is something DFS will want to traverse on so y is a reachable from x. With how DFS works, it will get to y eventually from x. It may go off the main path from x to y to explore other white vertices, but it will always backtrack and get closer and closer to y.

So that proves the other direction. □

Advice 20.8 — Parentheses Theorem and White Path Theorem often work in tandem. It will become evident in the proofs for algorithms later on.

§20.9 Cycle Detection

Lemma 20.9

A graph has cycle if and only if at some point during the DFS, we're at u and check one of its neighbors, v, and notice v is gray

Proof. Let's prove if the graph has a cycle, then DFS will detect a gray neighbor at some point.

So the graph has a cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$, where we can label the vertices in such a way that v_1 is the very first vertex that is discovered compared to all other vertices in the cycle. Consider the time when v_1 is discovered.

We know $v_2 \to v_3 \to \cdots \to v_k$ is a path (we know it's a path because it's a cycle) entirely of white vertices because v_1 is the first vertex in the cycle discovered. So by White Path Theorem, v_k is a descendant of v_1. So when we discover v_k, v_1 will still be gray by an earlier lemma. One way or another, we will eventually check the edge $v_k \to v_1$ before we finish v_k and behold that v_1 is gray. That proves one direction.

Now, we prove that if DFS detects a gray neighbor at some point, that means we have a cycle.

So we're currently at some vertex u and we check v (a neighbor of u) and we see it's gray. We know from v being gray, we're in the midst of trying to discover all we can that is unexplored beyond v, so we know there is some path the DFS used to get from v to u. Then, since v is the neighbor of u that we just so happened to notice was gray, we know $u \to v$. Concatenating the path from v to u with $u \to v$ shows the existence of a cycle. That proves the other direction. □

Advice 20.10 — This is a popular use of DFS. Maybe helpful on a written homework, especially for directed graphs.

The if and only if aspect of this statement makes it super useful. It means that we have definitive way to check if the graph has any cycles. This is because the if and only if means we will always find a cycle if there is one there AND we will never have to worry about false alarms.

§20.10 Summary

- We learned about DFS, which unlike BFS, prioritizes in trying to search deep rather than wide. In particular, DFS relies on a stack rather than how BFS relies on a queue.
- DFS will try to search as far deep as it can. Then, it will backtrack and cover paths it missed along the way.
- DFS uses the concept of colors to signal the status of a vertex and discovery/finish times. Both turn out to have some nice properties.
- DFS produces a DFS forest, in a similar analogy to BFS producing a BFS tree. The edges of a DFS forest have classifications.
- We proved Parentheses Theorem and White Path Theorem, which will be pivotal in proving later algorithms.
- DFS can detect cycles.

§21 Kahn's Algorithm

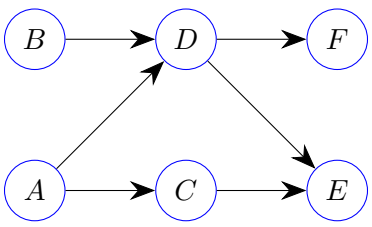
§21.1 Introduction and Topological Sorts

The topic of topological sorts was something covered in the spring for CIS 1600 but not the fall. So I should define it and give a good enough intro on the topic to equal the footing.

Definition 21.1 (Topological Sort)**.** A topological sort of a directed graph is a permutation of the vertices such that whenever $u \to v$ happens, u will appear somewhere before v in the permutation

Topological sorts do not exist in the context of directed graphs with cycles. Suppose the graph did have a cycle $v_1 \to v_2 \to v_3 \to \cdots \to v_k \to v_1$. Then, in the topological sort, we would need v_2 after v_1 by definition and using the cycle edges. Then v_3 comes after v_2, which comes after v_1. This continues until we have v_k comes after v_{k-1} which comes after everything before in the cycle, including v_1. So v_k is after v_1. But $v_k \to v_1$ mandates v_k comes before v_1, so there's a contradiction.

Anyway, a topological sort is best shown with an example.



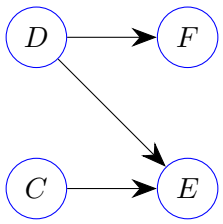
One example of a topological sort that works is A, B, D, C, E, F. You can test it out for yourself. Whenever $u \to v$ happens, u is before v in the sort. For example, $A \to D$, which forces A to appear before D.

But perhaps there is a way to actually get a topological sort instead of trial and error?

Well, let's consider the first thing we could add to the sort. We can't just add any vertex first can we? I mean what happens if we add D for example first? We know from the graph that $B \to D$, which means B needs to be before D in the topological sort. But that's impossible if D is first. So the reason D failed was because it had an arrow pointing in to it.

That lesson means, perhaps we should try something that doesn't have anything pointing in to it first, which would be a source.

So maybe let's let B go first. We might as well delete B and any edges attached to B, as it is no longer necessary. Well, A is another source in the graph now. So let's add that second. Delete A and all its edges. Here's the graph now:



Wait a minute. C and D became sources! Thanks to us removing the sources beforehand, it opened up new sources. There's no issue in adding them to the topological sort now. The sort right now is B, A, C, D. So now C and D are deleted. Then, E and F are sources and can be added. So the sort is B, A, C, D, E, F.

Essentially, the algorithm is always be adding sources to the sort. Then, delete the sources and any edges on them. This opens up new sources. Add those new sources to the sort. Rinse and repeat.

Also, similar to the first node in a topological sort always being a source, the last node in the sort is always a sink.

Anyway, the work we just did leads directly to Kahn's algorithm, which gets you a topological sort of a DAG (any topological sort as long as it works, since a graph could easily have more than one sort).

§21.2 The Algorithm

First, create a queue and an empty list T (that will soon be our topological sort).

- Create an array called "indegree" of length n, with all entries initially set to n.
- Go through all linked lists in the entire adjacency list representing the graph. Whenever you have $i \to j$ in the adjacency list, increment indegree$[j]$ by 1. At the end, this will accurately have the indegree of each vertex.
- Go through the array and check for any vertices with indegree 0. Add any such vertices to the queue.
- Repeat the following with the queue is not empty:
  - Dequeue the queue and call the vertex you get u. Add u to the end of T.
  - Check all of u's neighbors. Whenever you have $u \to v$, decrement indegree$[v]$ by 1. If this decrementation resulted in indegree$[v]$ being equal to 0, add v to the queue.
- Return T.

§21.3 Proof of Correctness

We need to show two things:

- All DAGs have a topo sort (remember, directed graphs with cycles never have a topo sort as proved earlier)

- Kahn's always gets you a full topo sort

Because it would look ridiculous if it turned out a topo sort existed for the graph, but the algorithm isn't able to get you one.

Remember that lemma that all DAGs have a source? And remember that we essentially can always add a current source of the graph at the end of T for free because there's no edges coming in to a source?

So a source S is always a reliable choice for the next thing in T. There's no issue in deleting S from the graph because all vertices u with $S \to u$ now have to appear after S in T at this point, just by inserting S in to T. Deleting S also doesn't change the fact that the graph is a DAG and reapply the lemma that there is a source. So we can always make choices at each step of the way, meaning all DAGs have a topo sort.

As for Kahn's actually giving you the topo sort, that's basically what we do whenever we discover a node has indegree 0 (meaning it's a source). So it works for all DAGs.

§21.4 Runtime

When we go through the adjacency list to compute indegrees, we essentially check each edge because it contributes something to some vertice's indegree. So that's $O(m)$ work.

We initially pass through the array once to get the ball rolling on finding sources. So that's $O(n)$ work. Also, eventually, all vertices will have made it to the queue at some point and left it because the topo sort covers all vertices. So that's $O(n)$, too.

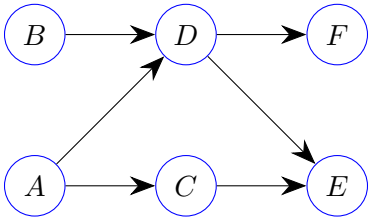
Whenever we remove a vertex u, we have to delete the edges associated with u, which is the decreasing indegrees by 1 is for. We eventually do this for all edges in the graph, so that's $O(m)$ work.

The total is $O(n+m)$.

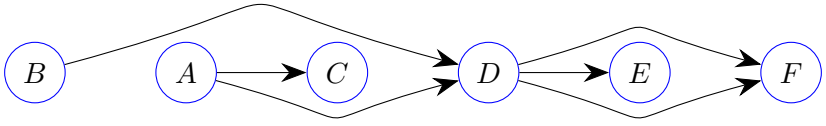
§21.5 Summary

- A topological sort is essentially a permutation of the vertices of a graph where the directed edges always point from left to right
- Only DAGs have topological sorts. A cycle will create a contradiction in trying to create a topological sort.
- We want to repeatedly remove sources and add them to the topological sort. Rinse and repeat and we will eventually get the full sort.

Last thing, take that graph we used as a demonstration:



Here's what it looks like topo sorted:



Looks a lot cleaner and organized, right?

> **Advice 21.2 —** Note that topologically sorting is $O(n+m)$ and most graph algorithms you're asked to do on homework or test will have a target runtime of $O(n+m)$ (or worse). So you don't lose anything by topologically sorting a DAG. In fact, you usually want to do it because it makes the

DAG feel more organized and easier to work with.

§22 Tarjan's Algorithm

§22.1 Introduction

So we learned what a topological sort is and how they can be better used to organize a DAG. While we learned one way, here is another way that puts good use of some intuition we learned previously.

§22.2 The Algorithm

- Run DFS on the graph (which must be a DAG). Keep track of finishing times.
- Output the vertices in order of finish time from biggest finish time to smallest.

§22.3 Proof of Correctness

So we revisit the concept of DFS, and we will get to use our theorems. First, an important lemma:

Lemma 22.1

If $a \to b$ in the graph, then $f[a] > f[b]$, regardless of how DFS is run.

Proof. Remember that we only care about topological sorts on DAGs. That will be important later. There are 2 cases.

Case 1: DFS discovers a before b

Because we discover a before b, at time $d[a]$, b is still not discovered. So there's a white path from a to b. So White Path Theorem says b is a descendant of a in the DFS forest.

Then, by Parentheses Theorem, $d[a] < d[b] < f[b] < f[a]$, so $f[a] > f[b]$ is true.

Case 2: DFS discovers b before a

We know $a \to b$ is an edge in the graph. Could there exist a white path from b to a at $d[b]$? Or better yet, could there exists ANY path (not just a white one) from b to a? If there was such a path, then we add $a \to b$ to that path to get a cycle. But this is a DAG, so that's a contradiction.

So there's no path from b to a, let alone a white path. Remember White Path Theorem is an if and only if. So by contrapositive, we can negative both sides of the definition and it's still true. Meaning, what we get is "y is not a descendant of x if and only if at $d[x]$, there was NO white path from x to y." So y and x don't have any ancestor/descendant relationship. We know b is discovered before a, so $d[b] < d[a]$. By Parentheses Theorem, $d[b] < f[b] < d[a] < f[a]$. Again, $f[a] > f[b]$ is true.

Both cases work out, so this proves the lemma. □

Our topological sort definition is saying $a \to b$ means a comes before b in the sort. The lemma means we can rephrase this to $f[a] > f[b]$ means a comes before b in the sort. It then becomes clear why putting the vertices in decreasing order by finish time creates a topological sort.

§22.4 Runtime

We run DFS once, so that's $O(n + m)$. To save time on adding the vertices in decreasing order by finish time, what we can do is add a vertex to the sort the moment it finishes, but new incoming vertices are inserted at the left in the sort, instead of the right. That's $O(n)$ time to eventually insert all vertices when they finish. So the algorithm is $O(n + m)$.

§22.5 Summary

- We showed an alternate algorithm to get us a topological sort. Both are good to know because one might be easier to modify for a problem than the other.
- We put the intuition on DFS to use. In particular, White Path Theorem and Parentheses Theorem work well in tandem.
- White Path Theorem leads to information about the presence of ancestor/descendant relationship, while Parentheses Theorem uses that information to determine relative finishing times.

§23 Kosaraju's Algorithm

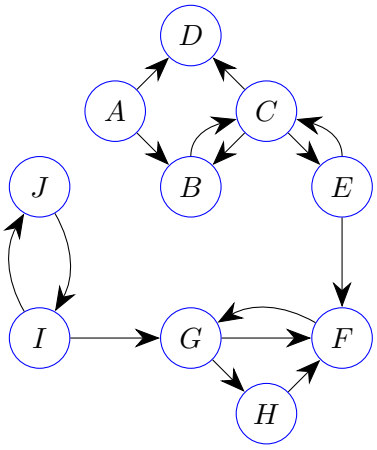
§23.1 Introduction

Let's refresh ourselves on the definition of an SCC (which functions as a connected component but for directed graphs):

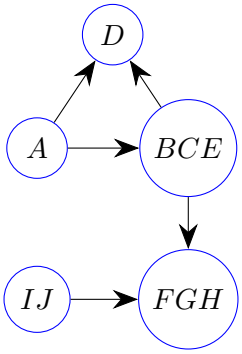
Definition 23.1 (Strongly connected component)**.** Similar to a connected component in an undirected graph. In a directed graph, any two vertices in a strongly connected component are strongly connected, and the component is maximal (meaning you can't add another vertex to the component so that all vertices in the component are pairwise strongly connected).

We know in a directed graph (unlike an undirected one) that just because we can get from u to v, that doesn't necessarily mean we can get from v to u. So it's better to think in terms of "strongly connected," which is defined as u and v being mutually reachable from one another. Furthermore, strongly connected has some nice properties with reachability.

Suppose u and v are strongly connected. If you can get from u to w, then that automatically means you can get from v to w. Similarly, if you get from w to u, then that automatically means you can get from w to v. So perhaps it's better to think in terms of strongly connected components? Let's look at this graph.



Looks pretty gnarly, right? What if we melded the vertices in the same SCC (strongly connected component) into one mega vertex? Take my word for it, but this is what the graph will look like:



Much better to look at and more organized. We call this graph of strongly connected components the G^{SCC}. It was also called the "reduced graph" in CIS 1600 during the spring.

To more formally define G^{SCC}, we say the vertices in G^{SCC} are the SCCs of G. Furthermore, there is an edge $A \to B$ if and only if there exists a vertex u in A and a vertex v in B such that $u \to v$. This edge definition should be intuitive: if $u \to v$, then that means you can get from the SCC A to the SCC B.

But of course, we need an algorithm to get this. Let's make some initial observations though that will be useful in grasping intuition.

Definition 23.2 (Transpose of a graph)**.** The transpose of a graph G is the graph obtained by keeping all vertices of G, but reversing the direction on every edge. We denote G^T as the transpose of a graph G.

Lemma 23.3

The graph $(G^T)^{SCC}$ has the exact same connected components as G^{SCC}, just the edges are reversed.

Proof. Just to clarify some notation, $u \twoheadrightarrow v$ will mean we can get from u to v and $u \not\twoheadrightarrow v$ will mean we cannot.

The key thing is, all paths from a to b in G will become paths from b to a in G^T because we reverse the direction on all edges. Thus, if $a \twoheadrightarrow b$ in G, then $b \twoheadrightarrow a$ in G^T. Likewise, if $a \not\twoheadrightarrow b$ in G, then $b \not\twoheadrightarrow a$ in G^T.

It becomes clear using those key facts that if a and b were strongly connected in G, they will still be strongly connected in G^T. In addition, if they were NOT strongly connected in G, they won't be in G^T either. This is because them not being strongly connected in G means either $a \not\twoheadrightarrow b$ or $b \not\twoheadrightarrow a$ in G. So in G^T, either $b \not\twoheadrightarrow a$ or $a \not\twoheadrightarrow b$.

This logic implies all SCCs remain the same between G and G^T. It also becomes with the "path reversing" logic why the edges are reversed between G^{SCC} and $(G^T)^{SCC}$. □

Lemma 23.4

G^{SCC} is always a DAG, for any directed graph G

Proof. We know one key part of the definition of an SCC is it is maximal: we cannot add another vertex to the component and still have pairwise strong connectivity between everything in the component. This will be necessary to make a contradiction.

So assume for the sake of contradiction that G^{SCC} had a cycle. One key property of the cycle is any two things in the cycle are strongly connected because we can just go around the cycle to get to whatever we want that's in the cycle. So all vertices in this cycle in G^{SCC} are strongly connected, we could put them all together in a set, where we know any two vertices in this set are strongly connected. But the vertices of G^{SCC} are SCCs themselves, so being able to put them together in this set violates the maximality part in the definition of an SCC (as mentioned at the beginning). So that's our contradiction. □

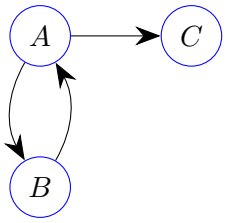
Definition 23.5 (Mega vertex)**.** A vertex in G^{SCC}, which represents some SCC in G

G^{SCC} being a DAG is useful information because we know then that it has a sink and a source (both of which have restrictive properties). It would be helpful if we could keep finding mega vertices that turn out to be sinks in G^{SCC} because the issue is an SCC could lead to another SCC, but a sink mega vertex will not lead to any other SCC by definition of a sink. Then, by removing vertices in that mega vertex from G, that could open up new sink mega vertices (sort of like Kahn's algorithm for topo sort but with sinks and not sources). Also, we learned about Tarjan's, which uses DFS finish times, so let's try that. Here's a question we want to ask:

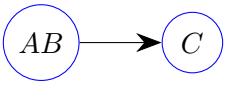
Is the vertex that finishes first in any DFS run always in a sink mega vertex?

Unfortunately, no. We will look at this graph.

Advice 23.6 — We were told this specific graph is often used as a counterexample.



Just by eyeballing it, this is G^{SCC}:



What happens if DFS starts at A and goes to B first instead of C? What will happen is A is visited then B. There are no white neighbors of B, so B finishes and is the first to finish. Clearly, G^{SCC} shows B is not in a sink mega vertex.

Well, shoot, that didn't work out like we hoped. But Tarjan's used the largest finish times first, so maybe the vertex that finishes last is always in a source mega vertex? Spoiler alert: yes. Enough backtalk, let's talk about Kosaraju's to actually get G^{SCC}.

§23.2 The Algorithm

- Run DFS on G, while keeping track of finish times.
- Compute G^T.
- Run DFS on G^T, but when choosing a vertex to call DFS Visit on, always choose the white vertex with the largest finish time. When we finish a visit call, designate all vertices caught in that visit within its own SCC.

I want to do the proof of correctness first before I show it in action. A lot of the intuition we built up in the introduction will come together in the proof.

§23.3 Proof of Correctness

Let C be some SCC. Define $d[C]$ to be the smallest discovery time among all vertices in C and $f[C]$ to be the largest discovery time among all vertices in C. Basically, this gives us a way to consider when an SCC is first discovered at all and when everything in it finishes.

Lemma 23.7

Let $A \to B$ for two SCCs in G^{SCC}. Then, $f[A] > f[B]$, no matter how DFS is run.

Proof. Remember our proof in Tarjan's? This will draw some inspiration from that.

Case 1: $d[A] < d[B]$

Let u be the first vertex discovered in A, so nothing else besides u is discovered in A yet. We know $d[A] < d[B]$, so nothing in B is discovered yet either.

Ignoring colors of vertices for now, by definition of an SCC, all vertices in A are reachable from u. Since $A \to B$, there exists a vertex v in A and vertex w in B such that $v \to w$. We can get from u to v because they're both in A, so now, we can get from u to w. But from w, we can get to everything else in B by definition of an SCC. So to summarize, from u, we can get to everything in A and everything in B.

Remember that when u is discovered, the rest of A and all of B are white. So since we can get from u to the rest of A and all of B, there are white paths from u to all those other vertices in A and B. Thus,

by White Path Theorem, if y is any vertex in A or B besides u, then y is a descendant of u in the DFS forest.

By Parentheses Theorem, $d[u] < d[y] < f[y] < f[u]$. In plain English, this means everything in A or B besides u will finish before u. So everything in B finishes before u (which is in A), implying $f[A] > f[B]$.

Case 2: $d[B] < d[A]$

Remember for Tarjan's how for this case, we exploited the fact that we were on a DAG? Well, G^{SCC} is a DAG, too, so we can play a similar trick again.

We know $A \to B$. If there were a path from B to A in G^{SCC}, that would imply the existence of a cycle. But that can't happen in G^{SCC}, which is a DAG. So for any vertex v in B, there is no white path from v to any vertex u in A. By White Path Theorem, this means v and u have no ancestor/descendant relationship.

We discover B first, so vertices in B get discovered before vertices in A. By Parentheses Theorem, we have $d[v] < f[v] < d[u] < f[u]$, which implies everything in A finishes after everything in B, so $f[A] > f[B]$. □

So a mega vertex with something pointing in to it would have to finish before the mega vertex pointing in to it. That means this lemma implies the vertex that finishes last will be a source mega vertex.

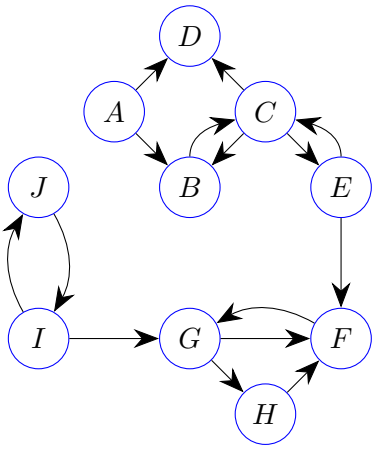
Assume we run BFS or DFS on the vertex with the last finish time in an effort to try to find everything in the source mega vertex. We will but that source mega vertex S could lead to other SCCs, so BFS/DFS could leak into that territory. And when we catch nodes in our traversal, we'd have no way of telling the difference between nodes in different SCCs.

Well, I didn't introduce the transpose of the graph for nothing. Remember how we initially wanted to get sink mega vertices because a DFS run in there would be contained completely within said mega vertex? Then we could just rinse and repeat with removing sinks and finding more? Well, taking the transpose of G keeps all the same SCCs as said before. But the edges are reversed and we can clearly see this turns sources into sinks. So we have our wish.

That's why when we have DFS run a second time, it does it in G^T. Basically, it starts at a source mega vertex in G, which becomes a sink mega vertex in G^T. The sink aspect contains DFS and prevents it from leaking into other SCCs by accident. Then, when we catch everything in the SCC, the blackness allows DFS to ignore it in future DFS visit explorations (essentially deleting those nodes from the graph). So we keep deleting nodes part of a sink mega vertex and in the process, open up new ones (like Kahn's but with sinks instead of sources).

§23.4 In Action

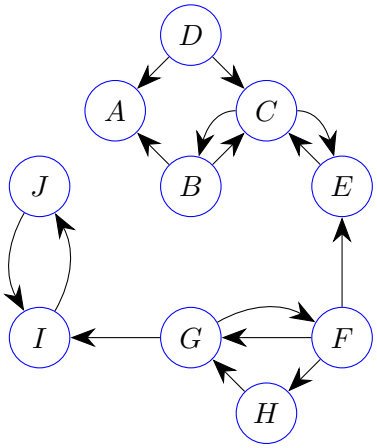
Okay, let's reuse that example from the beginning:



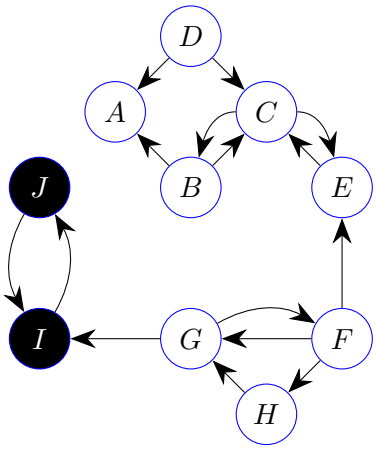
I'm not going to explain the DFS in great detail and just give you the start and finish times. If you need a refresher on how to DFS, go back to the DFS In Action subsection.

Node	A	B	C	D	E	F	G	H	I	J
Discovery Time	1	2	3	4	6	7	8	9	17	18
Finish Time	16	15	14	5	13	12	11	10	20	19

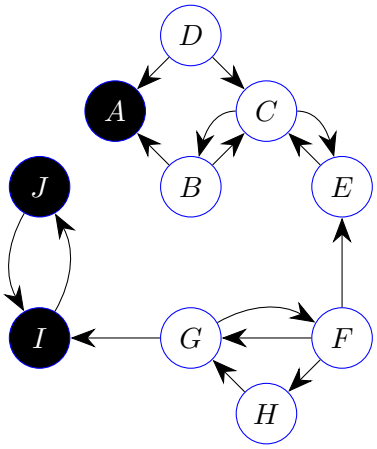
Now, let's compute G^T, where we reverse all the edges.



We start our second DFS by reverse order of finish time in our first DFS. So we start a DFS Visit with I and we get I and J. So IJ forms one SCC. We can also colors those vertices black to signify DFS has found them already.



Next highest finish time among the white vertices is A, so we do a DFS Visit on A and only get A. So A forms an SCC.



Now, we want to do DFS Visit on B. Notice that A is black and was an SCC we found previously. So when DFS tries to go from B to A but sees A is black, that's the intuition behind removing sink mega vertices as we go. Going in decreasing order by finish time helps us prioritize current sinks first and black vertices signify we put it in an SCC already. Since DFS doesn't care about black vertices, this prevents it from leaking into another SCC by accident.

Anyway, I'll run through the rest of this quick. The DFS Visit on B gets B, C, E, so that's an SCC. The next visit is on F, which gets F, G, H. The last one is on D that gets just D.

And there we have it. Our SCCs are IJ, A, BCE, FGH, and D.

Advice 23.8 — Remember G^{SCC} is a DAG too! So you can topologically sort it. In fact, there are problems where you will see Kosaraju's and then Kahn's in tandem, especially involving strong connectivity or reachability in directed graphs.

§23.5 Runtime

The first DFS run is $O(n+m)$. We have to compute the transpose though. This is luckily $O(n+m)$. Just create a new adjacency list to designate for G^T. Then, go through the entire adjacency list for G and whenever we have $u \to v$, we add $v \to u$ to the adjacency list for G^T. Then, the second DFS run is $O(n+m)$. So we have 3 $O(n+m)$ operations, meaning the runtime is $O(n+m)$.

§23.6 Problem Solving

I shall demonstrate the beauty of graph algorithms in tandem, since we've learned a good number so far.

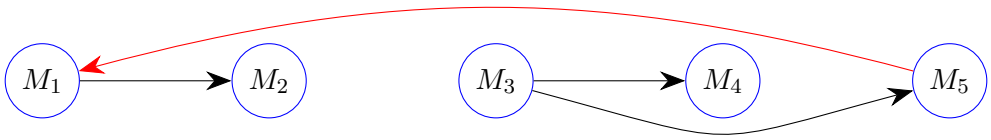
Example 23.9 (Recitation)

A directed graph is called "almost strongly connected" if adding a single edge makes the entire graph strongly connected. Design an $O(n+m)$ algorithm to determine whether a graph is almost strongly connected.

Okay, the problem is talking about strong connectivity, so maybe we should think about this in terms of strongly connected components. Let's run Kosaraju's to get G^{SCC}. Oh wait, G^{SCC} is a DAG, so let's topo sort it! Say T is the topo sort.

We know by definition of an SCC that any two vertices in the same SCC are strongly connected. So the issue now is being able to get from SCC to SCC and back.

Remember how in a topo sort, the last node L is a sink and the first node F is a source. So at the current moment, there is no way to leave L and no way to enter F. If we want any chance of the graph being strongly connected, we need a way to get from L to F with only one edge, so suppose we draw an edge from L to F. However, that doesn't necessarily make the graph strongly connected. Just look at this G^{SCC}, where adding the edge I mentioned doesn't necessarily make the graph strongly connected:



So how do we check that this is strongly connected? Run Kosaraju's on the graph with the added edge. If we get 1 SCC, the new graph is strongly connected, making the original graph almost strongly connected. If not, the original graph is not almost strongly connected.

Proof of Correctness? I alluded it as I talked through the thought process, but the idea is(there necessarily has to exist both a sink mega vertex and source mega vertex in the G^{SCC} because G^{SCC} is a DAG and all DAGs have at least one. And the topo sort properties allow us to locate an example of a sink and source (even though there could be more than one).

I also said how you can never leave the sink and never enter the source. Let u be some vertex in the sink mega vertex and v be some vertex in the source mega vertex. So at the moment, u cannot get to v, but we need that to occur to ever have a chance at the graph being strongly connected (where any two vertices are mutually reachable from one another). So we justify the necessity of drawing the edge to even have a chance. And to show whether or not drawing the edge actually works, the new graph is strongly connected if and only if it's 1 SCC, which is why we use Kosaraju's again and check the number of SCCs.

Runtime? Well, Kosaraju's is $O(n+m)$. Topo sorting is also $O(n+m)$. Kosaraju's again is $O(n+m)$. So the algorithm is 3 back-to-back $O(n+m)$ operations, which is still $O(n+m)$.

Advice 23.10 — See the beauty of turning a directed graph into G^{SCC}? Because a directed graph may not be a DAG, but G^{SCC} always is. And G^{SCC} not only makes thinking about strong connectivity easier, but the topo sort (which you can always do for a DAG) gives nice properties to get us on the right track.

§23.7 Summary

- We discovered G^{SCC} is always a DAG, which has the property of always containing a sink.
- We wanted to find sink mega vertices, but our initial plan failed. A sink mega vertex would be ideal though to make sure a DFS run doesn't leak into other SCCs by accident.
- We found we could get what we want in the end. We found highest finish times gets us source mega vertices. But switching the direction of all edges turns the sources into sinks. Then, we can repeatedly remove sink mega vertices to repeatedly get the SCCs.
- G^{SCC} being a DAG gives us the ability to topo sort it. This will be useful in thinking about reachability or strong connectivity because both topo sort and G^{SCC} organize the graph.

Advice 23.11 — You know the intermediate step of computing G^T? That can be a helpful strategy sometimes. We don't have many tools for finding all vertices that can reach v efficiently in a graph G. But by looking at G^T where all paths are reversed, it becomes finding all vertices that v can get to (or are reachable from v). So all vertices that can reach v in G are the same as all vertices that are reachable from v in G^T. And we have BFS that can accomplish this.

§24 Dijkstra's

§24.1 Introduction

We know BFS can get us the shortest path but that's only for **unweighted** graphs. If the graph is weighted, the shortest path may not necessarily be the one that uses the smallest number of edges (in unweighted, it's always the smallest number of edges). Suppose from a to b, there was an edge between the two but has weight 2023. However, there is an alternate route to get from a to b, which requires having to traverse 5 different edges, but each edge has weight 1. We can clearly see the alternate route is better despite having to traverse more edges.

So Dijkstra's is an algorithm that will get us shortest paths on a **weighted** graph. **But we cannot have any edges with negative weight.** The graph can be either directed or undirected and the algorithm will be successful in both cases.

Warning 24.1. If your graph is unweighted, Dijkstra's will be less efficient than BFS, so you'll want to use BFS instead for a better runtime.

§24.2 The Algorithm

First, some setup. Choose a source node s, where we'll be computing distances from s to everything else. Have a distance array of length n, which will eventually be updated to accurately record each node's distance from s. We'll also have a parent array of length n.

Also, create a min heap containing all the nodes of the graph. The min heap will use distances associated with each node in the distance array as the comparator. Set every node's distance to be ∞, except s's distance, which will be 0.

Now, repeat the following steps while the heap is not empty:

- Extract the minimum from the heap. Note the node v from extracting, as well as the associated distance d we got from extraction.

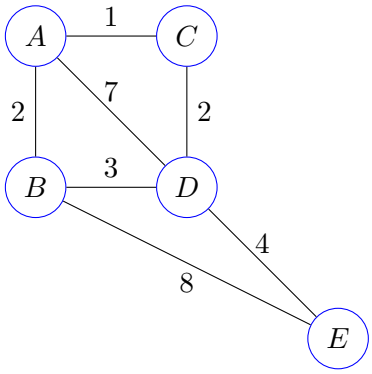
- Set v's distance to be d in the distance array.
- Check all neighbors u of v. Let $\text{wt}[v, u]$ be the weight of the edge from v to u. If we notice that distance$[v]$ + $\text{wt}[v, u]$ is less than distance$[u]$, we update distance$[u]$ with distance$[v]$ + $\text{wt}[v, u]$ and set u's parent to be v.

After the loop stops running, we can get the shortest path from s to all other vertices through the parent array. So if we want to see the shortest path from s to f, we trace parent pointers starting from f until we reach s and then we have the path in reverse order.

Warning 24.2. Do NOT use Dijkstra's if there are any negative edge weights. Dijkstra's will terminate if there are negative edge weights, but it won't always get you the shortest path. Edges with weight 0 are fine.

§24.3 In Action

For my demonstration, I'm going to give commentary that will make the proof of correctness easier to understand. Consider the following undirected weighted graph, where we will choose A to be the source.



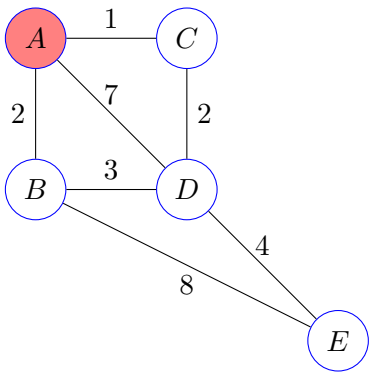
Let's do the setup. Here's our distance array, where from here on, everything I shade in gray means the associated vertex is still in the heap.

A	B	C	D	E
0	∞	∞	∞	∞

And we have our parent array, which doesn't have anything useful yet but will soon.

Node	A	B	C	D	E
Parent	null	null	null	null	null

Okay, A has the smallest distance in our heap, so we take it out. We check all neighbors of A to find B, C, and D and their associated distances from A. So we can create a path from A to C with weight 1, so we know the true minimum distance between A to C is 1 or less. Similarly, the true minimum distance from A to B is 2 or less and from A to D is 7 or less. That's why we keep track of this information in the table.



A	B	C	D	E
0	2	1	7	∞

Node	A	B	C	D	E
Parent	null	A	A	A	null

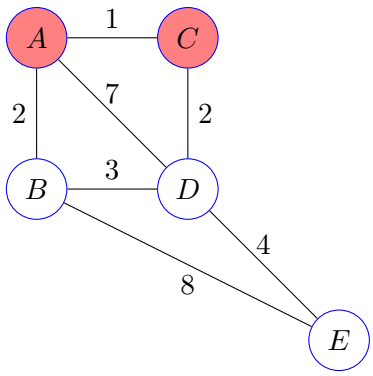
I shaded A in bright red to show it's out of the heap. We'll see the significance of that soon.

We know the only ways out of A are to B, C, or D. Of those vertices, C has the smallest edge weight with A. I claim, we already found the true shortest distance between A and C because of that fact of C having the smallest edge weight with A. Why is this? Well, if we want to get to C, we have to leave A somehow. If we don't go directly to C, we have to use B or D (the only other ways out of A) to get there. But $A-B$ and $A-D$ have heavier weights than $A-C$. So the paths to B and D already suck in comparison to $A-C$, so any alternate route using B or D to get to C automatically sucks. So we know we found the shortest path from A to C because all other routes out of A suck immediately.

However, we cannot conclude that we have found the shortest path from A to D just by the edge weight $A-D$ because it's not the smallest out of the edges coming out of A. We know $A-C$ is smaller, so for all we know, there could be a path that goes from A to C then D that is lighter than the direct way from A to D. So we cannot say anything for sure about the shortest distance from A to D (or A to B for that matter), unlike A to C. This intuition will be necessary for the proof of correctness.

Anyway, we are told to extract the min from the heap (the gray cells in the array), so we do so (which is C). We check neighbors of C. We see distance$[C]$ + wt$[C, A]$, which is 2, is greater than distance$[A] = 0$, so we don't do anything. However, distance$[D]$, currently at 7, is greater than distance$[C]$ + wt$[C, D] = 3$, so we change distance$[D]$ to 3. In plain English, what we're saying is "we found a new path from A to D that's lighter than the lightest path we found so far going from A to D, so we update the data with a new lightest path found thus far from A to D."

Let's update all our data:



A	B	C	D	E
0	2	1	3	∞

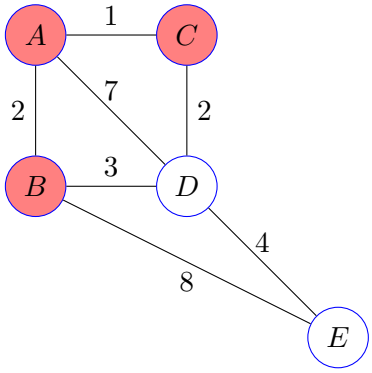
Node	A	B	C	D	E
Parent	null	A	A	C	null

Note that at this point, we checked all neighbors of all red nodes thus far and kept a record of all the lightest paths we've gotten thus far from that.

So we notice B has the smallest distance in our min heap (the gray cells in the array) now, and let's think about the significance of that. We want to get from A to B, which will require us to exit the blob of red nodes at some point. We know we found the direct path from A to B, but we have to consider if we can get from A to C then to B. Well, we checked all neighbors of C and logged the appropriate distances on the table. Because B is the smallest currently in the heap, if there was a smaller distance in

the heap, we would have that minimum be extracted instead of B. So all routes out of C to get to B automatically suck because they're automatically bigger than the path from A to B with weight 2. So we can be sure we found the true smallest distance to B.

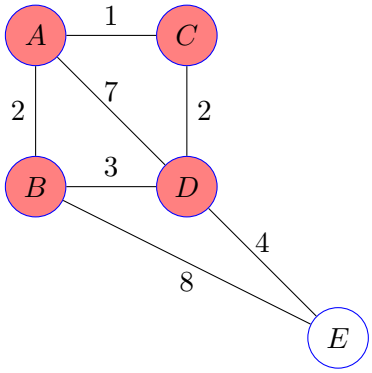
Anyway, we remove B from the min heap and check all its neighbors and do the data update. Here is what the data looks like now:



A	B	C	D	E
0	2	1	3	10

Node	A	B	C	D	E
Parent	null	A	A	C	B

Same drill, get the min in the heap, which is 3 and check all neighbors of D. Update data accordingly:



A	B	C	D	E
0	2	1	3	7

Node	A	B	C	D	E
Parent	null	A	A	C	D

Checking E, we find there's no updates to our data that we need to do. So here's the final data:

A	B	C	D	E
0	2	1	3	7

Node	A	B	C	D	E
Parent	null	A	A	C	D

§24.4 Proof of Correctness

Warning 24.3. This is one difficult proof of correctness to understand. Don't feel ashamed if you need to ask me on the Reddit thread to clear up some stuff. I will say it may be good though to apply this proof to the example we just did to better understand it.

Remember how we colored those nodes red to signify they were taken out of the min heap? I'm going to show via induction that those red nodes are exactly the vertices where we know for sure we found the true minimum distance to them.

The base case is when we have the source node s. Because we have nonnegative edge weights, it turns out s's smallest distance from itself is always 0. Because even if we venture out of s and come back, we're going to be using some edges along the way. Those edges are nonnegative, so that risks increasing the distance as we go along them. Even if we have a path of edges of weight 0 going in and back to s, there's no point in that tomfoolery when we can just stay still because it doesn't change the fact that the smallest distance from s to itself is 0. Bottom line, we know the shortest distance from s to itself, so we can color it red.

So here's the inductive step. We're looking to expand our collection of red nodes (nodes where we know we found the shortest distance to). But to recruit a new white node w to the red node club, a path from s (which we know is red) to w will eventually have to leave the red node club, since w is not red right now. Part of the algorithm involves checking neighbors of a node as they're recruited to the red club and updating new smallest distances. So for all white vertices v that have a red node pointing in to them, we know the shortest path to them that explicitly involves everything in the path being red up until v (but this might not be the actual shortest path).

Among all such v, the one u with the shortest distance can be recruited into the red club. Why? Because we explored all other ways out of the red club. Them having higher distances means those paths are greater in weight (and thus suck) in comparison to the shorter one we found. So we can be sure we found the true shortest path in the graph between s and u and add it to the red club. That's the inductive step done.

§24.5 Runtime

Note the graph doesn't necessarily have to be 1 connected component. But we will never get the distance between s and those vertices that aren't reachable from the source and that will mean less work.

Assume worst case though that everything is reachable from the source, so all nodes and edges will be processed.

We create a min heap of all the nodes at first and setup the distance and parent arrays, which is $O(n)$ total work.

We eventually remove all nodes from the heap, removing when we want to recruit a new red node. Removing one element is $O(\log n)$, so eventually removing them all is $O(n \log n)$.

Whenever we traverse an edge, we have the possibility of updating a distance because we found a better one. Worst case, assume we do that for every edge. Updating a distance is basically decreasing a key in the heap, which is $O(\log n)$ work through sifting in the heap. So if we assume we do this for every edge, worst case, this is $O(m \log n)$.

So the total is $O((n + m) \log n)$. As you can see, this is less efficient than BFS, but BFS won't work on weighted graphs. So you wanted to use BFS for distances in unweighted graphs and Dijkstra's on weighted graphs. Note that if all edge weights are equal, it might as well be an unweighted graph, so don't let them trick you with that.

§24.6 Summary

- We learned about Dijkstra's, which helps us get the smallest distances from some source node to every other node in the graph. We can retrace the relevant path using the parent array.
- Essentially, the proof involves taking the vertices we know we found the shortest distance to and slowly expanding that set. The set can be expanded by looking at the set currently and checking all neighbors, particularly those outside the set. The one with the smallest distance is the best because all other routes out of the set suck in comparison.

- The runtime is $O((n+m)\log n)$, which is our first instance of a graph algorithm having a different runtime than $O(n+m)$.

§25 Kruskal's Algorithm

§25.1 Introduction

Suppose we're given an undirected weighted graph that is connected. Our goal is to get a subgraph such that all the original vertices are present in the subgraph, any two vertices in the subgraph are connected, and the sum of the weights of all edges used in the subgraph is as small as possible. We call this the "minimum spanning subgraph."

Basically, this could have applications in wiring where we want to connect everything up in a network together but with as little cost as possible.

Lemma 25.1

When all edge weights are positive, the minimum spanning subgraph is a tree.

Proof. A tree is connected and acyclic. We know the minimum spanning subgraph is connected by definition. So we now have to show it's free of cycles. Suppose for the sake of contradiction that we did have a cycle.

Remember that deleting an edge from a cycle does not compromise connectivity. Because if a path from two vertices used that deleted edge, it can take a detour the other way around the cycle instead. Deleting an edge (which is positive) from the cycle reduces the overall weight of the spanning subgraph. So there is no negative consequence to deleting this edge and by doing so, we get a spanning subgraph with overall less weight. So the "minimum spanning subgraph" with a cycle isn't actually minimal, which is our contradiction. □

Warning 25.2. This lemma does NOT work when we have negative edge weights or weights equal to 0. In fact, we lose nothing by including such edges in the subgraph, even if it ends up forming a cycle. But for the most part, we only care about graphs with only positive edge weights, so the minimum spanning subgraph will be a tree.

§25.2 Minimum Spanning Trees

So as alluded to previously, with a graph G having positive edge weights, its minimum spanning tree (or MST) is defined as a subgraph of G with the following properties:

- It is connected and includes all the original vertices
- It is acyclic
- It uses as little total weight as possible

Warning 25.3. Note that the MST is not necessarily unique, as the minimum could be achieved by multiple MSTs.

We need to prove some stuff first to better understand some algorithms for finding MSTs:

Theorem 25.4 (Cut Property)

Partition the vertices of G in to two disjoint groups P and Q such that every vertex of G is in one of the two groups. As long as the groups are disjoint and every vertex is in one of the two groups, create the groups however you want. Now, consider the set of edges where one endpoint is in P and the other is in Q. Among those edges, if there is a unique edge with minimum weight, that edge must be in every MST.

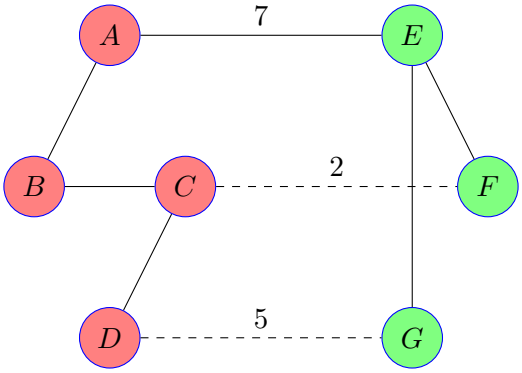
Proof. Remember the exchange argument back from Huffman? We'll use that here. So suppose we have our groups P and Q. Let S be the set of edges in the graph where one endpoint is in P and the other is in Q.

In an MST, we're only going to use only one edge from S. Why? We know by a tree being connected that all vertices in P can reach other and same with Q and the two groups can have their vertices reach each other through an edge in S. If we didn't use any edge in S, well, then vertices in P could never reach anything in Q, contradicting connectivity of a tree. Also, we know a tree is minimally connected (meaning deleting any edge always disconnects the graph). But if we had more than one edge from S used in an MST, we could delete one such edge from the MST and still have a way to get between P and Q, so the graph is still connected, meaning the original "tree" is not minimally connected. That's a contradiction there. So in an MST, we only use 1 edge from S because anything else creates a contradiction.

We know if we delete this edge e from S, it will divide the MST in to exactly two connected components. However, if we choose some other edge going from P to Q and use that instead, it will connect the graph again.

So now suppose, e is the edge we use from S in our MST, but it isn't the lightest edge from S. Then, we can "exchange" e for the lighest edge from S. As reasoned before, this still gets the job done in connecting the graph, but reduces the overall weight by choosing an edge that gets the same job done but with lower cost.

We can see in this picture below with groups $ABCD$ and EFG:



How, $ABCD$ and EFG are connected up is not relevant for this demonstration: just the edges going between the two groups. Suppose only $A-E$, $C-F$, and $D-G$ go between the two groups, and we're using $A-E$ for this "MST." But why bother using $A-E$ when we can use $C-F$ to connect the two groups instead? And this has lower cost without changing anything else, so this is necessarily a better option. □

Theorem 25.5 (Cycle Property)

Consider a cycle in a graph G, where the cycle contains a unique heaviest edge. That edge is never in any MST of G.

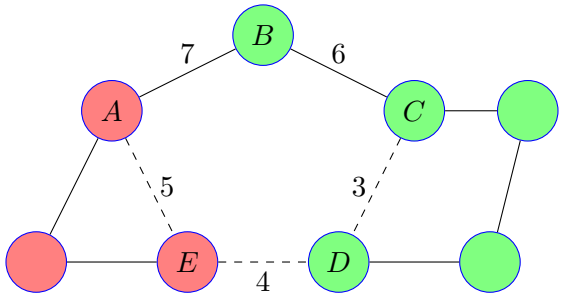
Proof. So let $v_1 - v_2 - v_3 - \cdots - v_k - v_1$ be the cycle and label $v_k - v_1$ as the unique heaviest edge. Suppose an MST had this edge. Let's partition the vertices into two disjoint sets P and Q. Suppose P has v_k and Q has v_1.

Now, let's suppose we walk our way around the cycle, regardless if the edges along the way are actually in the MST or not. From v_k to v_1, we left P and entered Q. Since P and Q are disjoint sets, if we want to go around the cycle and eventually return back to v_k (in P), that will have to involve going from Q into P. So there's at least one other edge in the cycle besides $v_1 - v_k$ that has one end in P and one end in Q. Call this edge $v_i - v_{i+1}$, where i is some integer with $1 \leq i \leq k-1$.

Now, this isn't the exact same thing as the cut property because we don't know all edges with one endpoint in P and another in Q. However, we can use the exchange argument here. Whatever edge $v_i - v_{i+1}$ is, it is lighter than $v_k - v_1$ because $v_k - v_1$ is defined as the unique heaviest edge in the cycle.

Just like the cut property logic, we can exchange $v_k - v_1$ for $v_i - v_{i+1}$ and get the job done with less weight. So that automatically disqualifies our original tree as optimal containing the unique heaviest edge in a cycle because we showed we can improve it.

Once again, here's a demonstration:



The cycle in the graph is $A - B - C - D - E - A$. The dashed edges are those in the cycle of the original graph but not included in the MST. Everything not in the cycle is not really relevant for the demonstration.

As you can see, we made a partition with the red and green vertices. As we go around the cycle, we have a second edge $D - E$ that goes from red to green. We can see $A - B$ also goes from red to green but is the heaviest edge in the cycle. And so, we could exchange $A - B$ with $D - E$ and still have the tree be connected but at lower cost. So that means our original MST wasn't actually minimal. □

§25.3 The Algorithm

I've showed enough background. Here's an algorithm to get us an MST:

- Sort the edges in increasing order by weight
- Go through the edges in order one by one. Add them to the graph one at a time except if adding the edge would create a cycle in which we ignore the edge and move on to the next.

§25.4 Proof of Correctness

This essentially utilizes both the cut property and cycle property. Suppose we want to add the next edge $u - v$ in to what we have of our MST so far and we know adding this edge doesn't create a cycle in the MST. Consider the connected component containing u in our current progress of the MST. This will be one group in the partition. Everything else will be in another group.

Now, u and v cannot be in the same group. If they were, then it can be seen that u and v are in the same connected component, so in our MST so far, there is a path from u to v. So adding $u - v$ creates a cycle, but we're defining $u - v$ so that adding it doesn't create a cycle. So we can rest assured u and v are in different groups. Also, we haven't drawn any edges in our MST so far that goes between the two groups, since one of the groups is entirely its own connected component, disconnected from the rest of the graph.

Since we define $u - v$ to be the smallest edge we haven't drawn so far, we can rest assured this is the smallest edge between to two groups in the partition. The cut property ensures we are justified in adding this edge to the MST.

One more thing, when we come across $u - v$ and it creates a cycle, we have to be justified in discarding it. The reason for this is as follows: we process edges in increasing order. So if we know $u - v$ creates a cycle we have already come across all smaller edges in the cycle, so $u - v$ is some largest edge in its cycle. The cycle property justifies why we discard this edge.

So that justifies our algorithm and then the correctness follows inductively by adding the edges.

§25.5 Runtime

Okay, sorting the edges is $O(m \log m)$ by merge sort (the fastest sorting algorithm we've learned in the worst case, tied with heap sort).

Whenever we add each of the m edges, we have to check if it will create a cycle. We know DFS can do cycle detection. If we do that m times, that's already $O(m(n+m)) = O(mn+m^2)$. Shoot. Is there a faster way? We'll be learning union-find next section for that purpose. But I will tell you that union-find can determine whether or not the added edge creates a cycle in $O(\log n)$ time. So the total time spent on cycle detection is $O(m \log n)$.

So the total runtime is $O(m(\log m + \log n))$. We can make some additional simplifications to do this. We know to create an edge, we have to choose a starting point (n possible choices) and an ending point (n possible choices). So there are n^2 possible edges. So an upper bound for m is when all these edges are present, so $m \leq n^2$. Taking the log of both sides means $\log(m) \leq 2\log(n)$. So we can simplify the O time by bounding $\log(m)$ above by $2\log(n)$ so that it is $O(m(3\log n))$ or $O(m \log n)$.

§25.6 Summary

- The minimum spanning subgraph of a graph with positive edge weights must be a tree.
- We proved the cut and cycle properties for MSTs so that they can be used to judge optimality or unoptimality of our choices.
- Kruskal's involves adding the lightest edges to the graph one by one, unless adding the edge creates a cycle.
- We justify the optimality of our choices inductively through the cycle and cut properties.

> **Advice 25.6 —** Note that Kruskal's will get you the minimum spanning **tree** for graphs even with edge weights that are negative or 0. However, the minimum spanning tree isn't necessarily the minimum spanning **subgraph** when we have non-positive edge weights.

§26 Union-Find

§26.1 Introduction

So we may have a bunch of elements that are in disjoint sets. Union-find is a data structure that helps keep track of which elements are in the same/different sets. The disjoint property means every element is in exactly ONE of these sets.

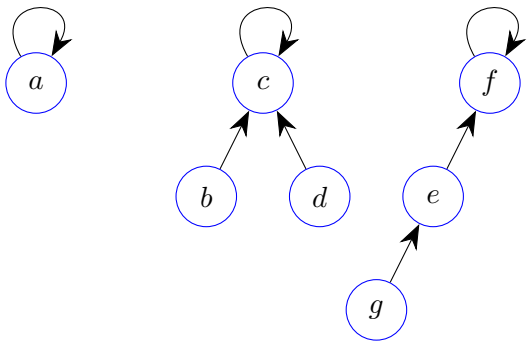
For example, we can a collection of elements a, b, c, d, e. One example of how they can be in disjoint sets is $S_1 = \{a\}$, $S_2 = \{b, e\}$, and $S_3 = \{c, d\}$.
There are 3 operations for Union-Find:

- makeset(x) - creates a singleton set $\{x\}$
- union(x, y) - merges the set containing x with the set containing y
- find(x) - returns the set that x belongs to

Note that we can't break sets up with Union-Find. But we won't need to for what we're going to use it for.

As for implementation, we're going to be using nodes and make a forest-like structure. Basically, each set will be its own distinct tree in the forest and each set will be represented by 1 element in said set (called the "head"). The head will be located at the root of its tree and all other elements in the set will have a series of pointers eventually leading to the root.

Here is an example of a forest representation of the collection of sets $S_1 = \{a\}$, $S_2 = \{b, c, d\}$, and $S_3 = \{e, f, g\}$ (with a, c, and f being the heads of their respective sets):

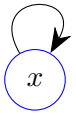


It doesn't really matter which element of a set is designated as the head, as long as each set has exactly 1 head and all other elements in the set eventually lead to the head.

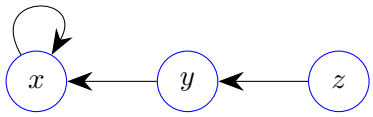
Also, every head node has a pointer to itself, so that will let us know when we've reached the head.

§26.2 Implementation of Operations

For makeset on an element x, we simply create a node for x and have it be the head of the singleton set. So x will point to itself.

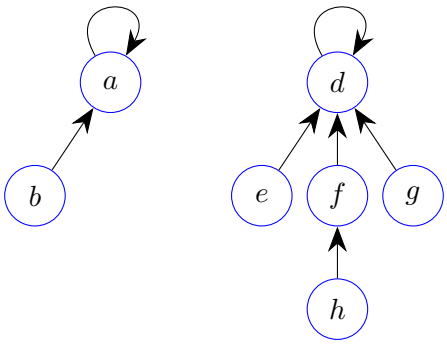


For find, we want to return the head of the set containing the input vertex. So we just follow the arrows until we reach the head. We know we reach the head when we get something pointing to itself. Suppose we want to find(z) in the example below.



So we go from z to y to x by the arrows. We check the arrow of x to see it points to itself, so we return x (the head of the set containing z).

Now for union. Here's the general idea. Consider these two sets:



If we want to merge these two sets, we should ideally have $a \to d$ or $d \to a$. This is because all elements eventually point to the head of the set and changing the pointer redirects everything to the head of the new set. So elements from both sets all lead to a head of the combined set.

But notice that the tree with root a has smaller height than the tree with root d. So if we do $a \to d$, we don't change the height of the overall tree (but $d \to a$ would result in a taller tree). We want shorter trees objectively because it speeds up the find operation.

This introduces a new concept called "rank," where the rank of a tree in our forest is its height (or essentially, the largest number of nodes we may need to travel in the tree to get to the head). We'll use this concept in order to do cleverly execute union so that it doesn't make the new merged tree any taller

than it needs to be. The example above shows the tree with root a has rank 1, while the tree with root d has rank 2.

How to do Union using Rank

What we'll be doing is storing the rank of a tree at its head for convenient access to it when we need it. And whenever we create a new singleton set, we say its rank is 0 and store that information at the head of the singleton set. So to do union(x, y):

- First, say h_x and h_y are the heads of the sets containing x and y, respectively. Also, let R_x and R_y be the ranks of the sets containing x and y, respectively. We get this information at h_x and h_y, since we store ranks at the head.
- If $R_x > R_y$, change h_y's pointer to h_x
- If $R_x < R_y$, change h_x's pointer to h_y
- If $R_x = R_y$, change h_x's pointer to h_y AND increment R_y by 1

So this takes the intuition of wanting to change one head pointer of the two sets and direct it to other head in order to merge the two sets. However, we do it cleverly to make the rank of the merged set as small as possible. I'll explain the intuition behind the strategy.

So suppose one set S_1 has rank a with head h_1 and another set S_2 has rank b and head h_2. Also, suppose $a < b$. So the worst node in S_1 is a nodes away from h_1. If we do $h_1 \to h_2$, then the worst node originally from S_1 has $a + 1$ nodes to get to h_2: it follows the path to h_1 and then goes to h_2. But since a and b are integers, $a < b$ implies $a + 1 \leq b$. So the rank of this merged set is b because adding one extra edge to traverse for the nodes in S_1 doesn't increase the overall height of the merged tree. So we don't need to increase the rank any more than we need to when the two sets are of different rank.

When they are of the same rank, however, we don't have a choice. Suppose both sets have rank a, so the worst node in S_1 is a nodes away from h_1, and the worst node is S_2 is a nodes away from h_2. When we do $h_1 \to h_2$, that worst node in S_1 becomes $a + 1$ nodes away from the new head. Similar problem when we try $h_2 \to h_1$, where the worst node in S_2 becomes $a + 1$ nodes away. So in this case, we just have to accept we're going to increase the rank.

But let's examine more about the rank aspect because it can't increase TOO much right? Let $f(r)$ be the minimum number of nodes to build a tree with rank r, assuming we use the union with rank strategy we worked out. Obviously, a base case would be $f(0) = 1$, since a set in union-find has at least one element and the singleton has rank 0.

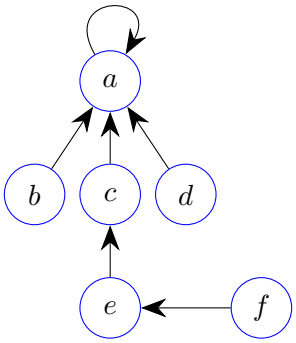
If we want to craft a set with rank r, we need to merge two sets both with rank $r - 1$. Remember if the two sets are of different sizes, the rank of the taller set won't increase (merging a set of rank $r - 1$ with one of rank $r - 2$ for example gives you another set of rank $r - 1$, per our algorithm). If we assume the two sets of rank $r - 1$ that we're merging have as few nodes as possible, they both have $f(r - 1)$ nodes. They're two of these sets to merge to create the set of rank r, so $f(r) = 2f(r - 1)$. With the base case of $f(0) = 1$, it's not hard to see inductively that $f(r) = 2^r$. So we require at least 2^r nodes to have a tree with rank r, per definition of $f(r)$.

So a set of n nodes when put in a union-find tree is going to have a rank $O(\log n)$. Rank is essentially how many nodes it takes at worst to get to the root, so now, find and merge are $O(\log n)$ operations.

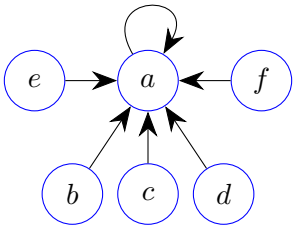
§26.3 Path Compression

Our rank system gives us good enough times for what we're doing. But if we really wanted to save time with find (which is also used in union), there's a neat trick called "path compression."

Here's a tree in our union set. Note that this is not a possible tree that can be created from the rules of union by rank, but that isn't necessary for this demonstration.



Suppose we called find on f. It will take us $O(\log n)$ time worst case to get to the head. However, we know everything along the path to get to f will also eventually lead to the head too. So what we can do is take all nodes en route to the path and change their pointers all to the head when we find the head:



So for f and anything that was in its way to a, it now has immediate access to the head, so find is now $O(1)$ for these nodes. This makes find operations on these nodes faster in the future or when we may need to do union (which requires find) on anything involving the moved nodes.

§26.4 How to apply to Kruskal's

It turns out that while path compression is a nice feature, the rank stuff is enough to speed up time for Kruskal's. So assume no path compression for simplicity sake.

Now, we show how to use union-find with Kruskal's. The basic idea is each set in our union-find will represent 1 connected component in our graph so far. To begin, no edges are drawn yet, so all vertices are their own connected component. So we make singletons for each vertex.

Now, suppose we wanted to add edge $a - b$ as Kruskal's algorithm instructs, but we need to check if it will create a cycle first.

Essentially, we can figure that out by calling find on a and b and seeing if we get the same output or not (meaning, we can determine if a and b are in the same set or same connected component). Why is this?

Suppose a and b were not in the same connected component, meaning they are in different sets. Then, the "island" aspect of a connected component means there's no current way to get from a to b before $a - b$ is added. So adding the edge $a - b$ isn't going to create a cycle.

However, if a and b are in the same connected component, then they are in the same island, so there's a path from a to b before $a - b$ is added. Then, using that path from a to b along with edge $a - b$ will create a cycle,

So we have a definitive method of checking whether adding $a - b$ creates a cycle: through checking if a and b are in the same set.

Anyway, if we add edge $a - b$, we want to call union(a, b) because we connect the CCs containing a and b, so merging the sets reflects just that. Otherwise, if adding $a - b$ would create a cycle, we don't care and move on.

So to handle one edge, we call find twice to check if the two endpoints are in the same set and worst case, that includes a union, too. So that's 3 $O(\log n)$ operations, so worst case for checking if a single edge creates a cycle is $O(\log n)$. This is definitely better than having to run DFS every time we want to add an edge!

§26.5 Summary

- The Union-Find data structure organizes a group of elements into disjoint sets. Each set is represented by a head element with all other members of the set eventually leading to the head of

its set.

- We can merge two sets by changing where the head of one set points to.
- Rank essentially measures how tall a tree is representing a set. We can then do our merges as to not increase the rank any more than we need to, which speeds up the operations of Union-Find.
- Union-Find is used in Kruskal's by having each connected component be its own set. This makes cycle detection $O(\log n)$ instead of having to run DFS each time.

§27 Prim's Algorithm

§27.1 Introduction

We learned about Kruskal's algorithm, but it requires Union-Find to be time-efficient. Here's another algorithm for finding an MST that is easier to implement in Java because you aren't required to implement Union-Find as a prerequisite. This algorithm shares many parallels to Dijkstra's.

§27.2 The Algorithm

Again, setup. Choose a source node s, which can be chosen arbitrary. Have an array of length n called "LBE" (standing for lightest bridge edge). We'll also have a parent array of length n with all parents set to null initially.

Also, create a min heap containing all the nodes of the graph. The min heap will use the LBE associated with each node in the LBE array as the comparator. Set every node's LBE to be ∞, except s's, which will be 0.

Now, repeat the following steps while the heap is not empty:

- Extract the minimum from the heap. Note the node v from extracting, as well as the associated LBE w we got from extraction.
- Say p is v's parent in the parent array. Add $p - v$ to the MST.
- Check all neighbors u of v. Let $\text{wt}[v, u]$ be the weight of the edge from v to u. If we notice that $\text{wt}[v, u] < \text{LBE}[u]$, we update $\text{LBE}[u]$ with $\text{wt}[v, u]$ and set u's parent to be v.

§27.3 Proof of Correctness

We use the whole "red node club" intuition back from Dijkstra's. But here, red nodes are nodes that we know are connected up correctly for our MST so far. Essentially, our inductive step relies on the Cut Property with one group being our red nodes and the other group being the white nodes (everything other than a red node).

The LBE (or lightest bridge edge, as we called it) is a system to keep track of the edges that cross the two groups and thus create a "bridge" between them. Obviously, the Cut Property says the lightest edge crossing the cut between the two groups is in our MST, so we want to find that.

Our LBE system accurately keeps track of that: the min heap at every step will have every white vertex and when a vertex leaves the heap, that marks the vertex becoming red. As for how we know our LBE system is accurate, at every step of adding a red node, we check all its edges and update LBE. So at $\text{LBE}[u]$ in the array, we are essentially keeping track of the lightest edge crossing the cut as we add more vertices (sometimes updating it when new vertices become red). The parent pointer allows us to retrieve the actual edge should we want to add u to the MST and its associated smallest edge crossing the cut.

So that's the inductive intuition behind now Prim's works: we start with the source node in our red club and then keep expanding it by using the cut property repeatedly in finding the lightest edge crossing the cut and adding to the MST. Then, adding this new node to the red club forces us to check if we need to update our data and the process repeats until everything is in the red club.

§27.4 Runtime

This is exactly the same as Dijkstra's runtime, which is $O((m+n)\log n)$ because the workload at each step hasn't dramatically changed. However, we want the graph to be connected (or we wouldn't be able to get an MST in the first place). So $n \leq m-1$, meaning n is $O(m)$.

That allows us to simplify the runtime to $O(m \log n)$, just like Kruskal's.

§27.5 Summary

- Prim's algorithm is an alternative way to get an MST of a connected weighted graph that doesn't require Union-Find.
- It is very similar to Dijkstra's where we repeatedly use the Cut Property.
- We have a set of red vertices that we know are connected up properly for an MST. So we use the red vertices as one group and the other vertices as another and examine the edges cutting the cut. We get the smallest such edge crossing the cut to add to our MST.
- When adding a new vertex, we need to update some data because the red vertex group is slightly different. So we check edges of this new vertex to see if there is a better one crossing the new would-be cut.

§28 Hashing

§28.1 Introduction

Okay, imagine we're in some massive archive storage room of the records of every person in town. We have important documents pertaining to each person stored in boxes. If we didn't have boxes with labels, we'd just have a pile of random documents everywhere, and it could take a long time to find documents associated with a particular person.

We have a number of data structures related to this data storage and retrieval. We have a "key" and a "value." The key is like the name of the folder/box as an identifier for a set of data. Then, the value is the data associated with said name.

What we could do is sort all the keys by alphabetical order (or numerical order, since we use integer keys instead of string keys). Then, use good old binary search to see if we have a specific key in $O(\log n)$ time. But can we search in $O(1)$ time? Not quite, but we can get it in expected $O(1)$ time with hashing.

We not only have to be concerned with time but also space. Here's an example. Say like we're making a hash system for UPenn students, where the key is some student's 8 digit Penn ID and the value is important information associated with that student (like their name, major, school, year of graduation, etc).

Well, we know the 8 digit Penn ID is some number from 00000000 (eight 0s) to 99999999 (eight 9s), so we could have an array for each possible ID. Suppose my Penn ID is 12345678. Then, you could store my data at the 12345678th position at the array. Because it's an array, retrieving it would be $O(1)$ time.

However, of the 10^8 possible IDs, there are only roughly 28,000 students at UPenn in comparison, so only a small fraction of the possible IDs will be in use (I don't know how accurate that number is, but point is, the number of ID numbers actually in use is far fewer than the number of possible IDs). So we're wasting a ridiculous amount of space if we have this array for every possible ID.

We'd want an array of size maybe 30,000-35,000, give or take, to not waste as much space. But my ID number 12345678 doesn't fit in the table anymore, so we have to have some method to get it on there. However, we don't want to place my data in any old slot without thinking. Otherwise, we may forget where we put it and have to search the entire array if we want to find it again.

§28.2 Basics of Hashing

So with some of the problems we could run in to with hashing, let's actually define the basics.

Definition 28.1 (Hash table)**.** The array we use to store the data associated with a collection of keys. We define m to be the size of the array.

Definition 28.2 (Hash function)**.** A function that takes in a key and outputs a slot for it in the hash table. The output of the function must always be between 0 and $m-1$, inclusive, regardless of the input (or that would imply we place the key out of range of the array). An easy way to ensure it is within the bounds is to take modulo m (remainder when divided by m). If we were using integer keys k, one example of a hash function could be $h(k) = (k^2 + 5k + 7) \pmod{m}$.

Definition 28.3 (Collision)**.** The biggest problem with hashing. A collision when two different keys have the same slot in the hash table. For example, suppose $m = 10$ and our hash function is $h(k) = (k^2 + 1) \pmod{m}$. If we had keys 4 and 6 for example, we'd have $h(4) = 7$ and $h(6) = 7$, meaning 4 and 6 both go to slot 7 in the array. This is a collision.

Definition 28.4 (Simple Uniform Hashing Assumption (SUHA))**.** All the slots in the hash table must have the same probability of housing the next element to be added

Definition 28.5 (Load factor)**.** If m is the size of the array used for the hash table and n is the number of keys in the hash currently, the load factor (denoted as α) is given by $\alpha = \frac{n}{m}$. A higher load factor means operations like get, insert, and delete will be slower.

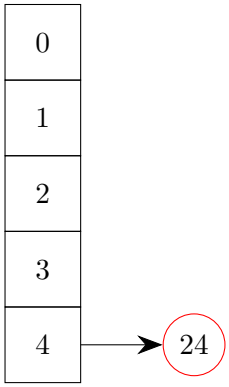
Now, we should address methods of handling collisions.

§28.3 Open Hashing

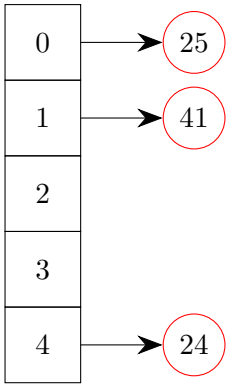
With open hashing, what will happen is the hash table will be an array of linked lists. We add elements to the start of the linked list in their respective slot in the hash table. This way, we can have multiple elements in the same slot.

Let's suppose $m = 5$, so our hash table is of size 5. Also, suppose our hash function, for simplicity sake, is $h(k) = k \pmod{5}$. Let's see what happens if we try to insert the keys 24, 25, 41, 9, and 74 in that order.

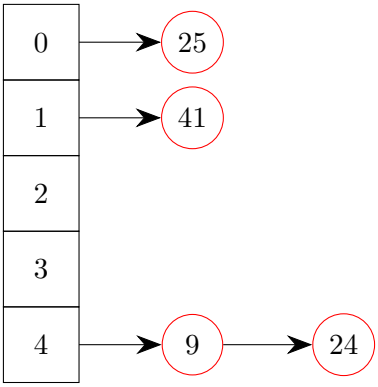
So $h(24) = 4$, so we try to insert 24 at slot 4 in table. Inserting in slot 4 is no problem.



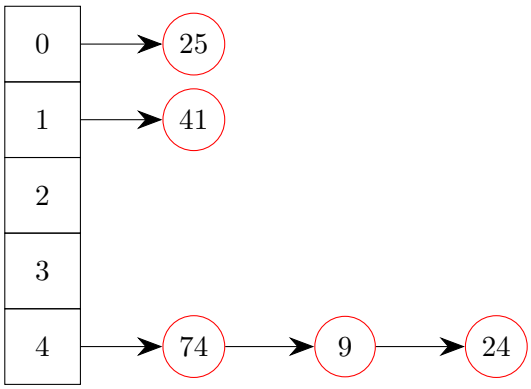
Then, $h(25) = 0$ and $h(41) = 1$ with both slots currently empty. Again, put them in no problem:



Next thing to insert is 9 and $h(9) = 4$. But slot 4 is already taken, so what we do is have a node for 9 and have its next node be the 24 we already have. Then, we change the pointer at the 4th position in the array to the node with 9:



Now, we try to insert 74 and $h(74) = 4$. Again, slot 4 is already taken, so we have the node with 74 be the new head of the linked list at slot 4 of the array:



We were able to do insert in $O(1)$, since we just make a new node and move some pointers around. But keep in mind, the worst case for retrieve/delete would be $O(n)$ here because the elements could all be in the same linked list. Using SUHA, all elements will be equally distributed throughout the hash table, so the expected time would be $O(1)$ for insert/retrieve/delete.

§28.4 Closed Hashing (or Open Addressing)

Closed hashing (also known as open addressing) is another collision resolution method. What happens is every slot in the array can store at most 1 element. So if we want to store a new element but see that the slot it would go in is already occupied, we have to find another slot for it. But we don't want to store it any random empty slot or it would be a hassle to find it again.

Definition 28.6 (Home slot)**.** The home slot for a key is the slot in the hash table where we WOULD put a key if that slot were empty.

Definition 28.7 (Probe sequence)**.** Basically, these are "plan Bs" for the home slot. This is a sequence of alternative slots where we put the key instead, should the home slot for the key already be occupied. If the first alternative is occupied too, we move on to the next alternative and so on until we find an empty alternative.

The probe sequence gives us a way to know the alternatives of where we may place a key should we need to find it again, instead of randomly sticking the key in any slot, where it will be harder to find again.

§28.4.1 Linear Probing

Linear probing is when the probe sequence goes to the next slot over on the right as the alternative and keeps going to the right until we find an empty slot. The position function for linear probing will be

noted as $h(k, i)$ (where is the key and i is essentially a counter for how many alternatives we tried so far). It will be in the form

$$h(k, i) = (h'(k) + i) \pmod{m},$$

where $h'(k)$ is a hash function that gives us the home slot for key k. Basically, what will happen is, if we want to insert key k, we first try position $h(k, 0)$ in the array. If that slot is unoccupied, we place the key there. If it's occupied, we try $h(k, 1)$ next (basically, the first alternative). Again, if it's unoccupied, we place it there, but if it's occupied, we move on to $h(k, 2)$. We keep doing this, incrementing i by 1 until we find an empty slot.

This is best done with an example. Let's say $m = 10$ and $h'(k) = k \pmod{10}$. So then, $h(k, i) = ((k \pmod{10}) + i) \pmod{10}$ Let's try to insert 35, 22, and 46. We know $h(35, 0) = 5$ and slot 5 isn't occupied. So we put in 35 without an issue. Also, $h(22, 0) = 2$ (not occupied yet), so that's no problem. Neither is $h(46, 0) = 6$.

0	1	2	3	4	5	6	7	8	9
		22			35	46			

Easy so far. But let's see what happens when we try to put in 72. We try $h(72, 0) = 2$. But slot 2 is taken, so we move on to $h(72, 1) = 3$. Slot 3 doesn't have anything so we're good here.

0	1	2	3	4	5	6	7	8	9
		22	72		35	46			

Let's try to put in 52. $h(52, 0) = 2$, fails. $h(52, 1) = 3$, that fails too. $h(52, 2) = 4$, empty. So we put it in slot 4.

0	1	2	3	4	5	6	7	8	9
		22	72	42	35	46			

Now, I want to put in 103. I'll spare you the details: we'll try slot 3, then slot 4, then slot 5, then slot 6, until we finally get to slot 7.

0	1	2	3	4	5	6	7	8	9
		22	72	42	35	46	103		

And from that insertion, you can see a clear problem: we have this blob of numbers all in a line that requires several probes to get through until we reach an empty slot. And placing the 103 at the end of the blob only made it bigger. This is called "primary clustering." It undermines the efficiency of this collision resolution method and thus, degrades performance.

§28.4.2 Quadratic Probing

So primary clustering in linear probing is an issue because it creates this contiguous blob of numbers in the array. What if we spaced out the alternative slots, so we create this blob? This is what quadratic probing is and the position function will be in the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \pmod{m},$$

where c_1 and c_2 are some constants. Same drill as linear probing to handle collisions: to insert key k, we first try $h(k, 0)$ then $h(k, 1)$ then $h(k, 2)$ and so on until we find an empty slot.

Here's an example because it turns out quadratic probing can have clustering of its own. Let's have $m = 10$, $h'(k) = k \pmod{10}$, $c_1 = 0$, and $c_2 = 1$ so that we have

$$h(k, i) = ((k \pmod{10}) + i^2) \pmod{10}.$$

Let's insert 5, 15, 25, 35, 45, 55, and 65 into the hash table. Note that all of these keys satisfy $h'(k) = 5$, which is deliberate for demonstration. So we know $h(5, 0) = 5$ and we can insert 5 no problem.

0	1	2	3	4	5	6	7	8	9
					5				

Now, 15. We have $h(15, 0) = 5$. Nope, already taken. Then, $h(15, 1) = 6$. Okay, we can insert it there.

0	1	2	3	4	5	6	7	8	9
					5	15			

Inserting 25? $h(25, 0) = 5$, nope. $h(25, 1) = 6$, nope. $h(25, 2) = 9$, that works.

0	1	2	3	4	5	6	7	8	9
					5	15			25

Okay, looks good, we don't have this annoying blob because the quadratic function helps us space stuff out. Let's try 35. We try $i = 0$, $i = 1$, and $i = 2$, all of which are taken with $k = 35$. But $h(35, 3) = 4$, which works.

0	1	2	3	4	5	6	7	8	9
				35	5	15			25

Again, I'll spare you the details and let you know that for inserting 45, it isn't successful until $h(45, 4) = 1$.

0	1	2	3	4	5	6	7	8	9
	45			35	5	15			25

And then 55 is successful at $h(55, 5) = 0$.

0	1	2	3	4	5	6	7	8	9
55	45			35	5	15			25

Now, we try 65. Take my word for it: no matter how big i is (could be 1 million), we will never be able to place 65. The explanation requires some number theory knowledge that isn't at all required for this course. But in Layman's terms, quadratic functions in modulo m might not always output every remainder from 0 to $m - 1$. So as we can see, the quadratic function will miss some values in the table, which means it won't be able to place a key when there are empty available slots in the table as you can see. This is called "secondary clustering." So quadratic probing isn't perfect either.

§28.4.3 Double Hashing

This is when we try to space things out but not have secondary clustering. We'll have two hash functions $h_1(k)$ and $h_2(k)$, so our position function for a key k is in the form

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \pmod{m}.$$

Not much to say here, except it's the same drill. To place key k, we try $h(k, 0)$ then $h(k, 1)$ then $h(k, 2)$ and so on until we find an empty slot.

§28.5 Summary

- Hashing is a data structure for data storage and retrieval. It maps keys to slots in an array where we can find them later.
- We don't have to want to create a massive array accounting for every single possible key if only a small fraction of the array will actually be used for data, since that would be a massive waste of space.
- However, by using hashing (where the array size is more appropriate in relation to the number of keys), operations only become expected $O(1)$ as opposed to the reliable $O(1)$ from one massive array. Operations could be $O(n)$ worst case.
- Different keys that go to the same slot cause a collision.
- Open hashing is where we have a linked list at each slot in the array to store more than one key that maps to the same slot.
- Closed hashing is where we have at most one key per slot. So if we want to insert a key at a slot that's already taken, we have to find an alternative.
- A probe sequence helps determine a list of alternatives to go through. We have linear probing, quadratic probing, and double hashing.
- Linear probing has the issue of primary clustering, where we have a blob of keys stored in consecutive slots in the array. Quadratic probing has the issue of secondary clustering, where we may not be able to place elements in the hash table despite there being empty slots in the hash table.

Warning 28.8. DO NOT use hashing if the target runtime isn't "expected" on a homework or exam question. Because if aren't measuring it in terms of expected runtime, they assume worst case, which hashing won't fare very well. Also, if you do you want to use hashing where the target runtime is expected, be sure to clarify in your runtime analysis that operations like add or find are **expected** $O(1)$.

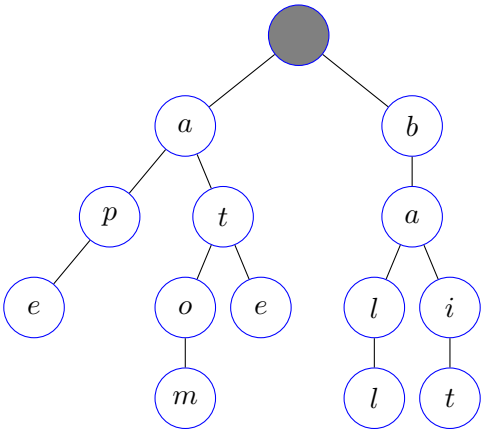
§29 Tries

§29.1 Introduction

So hashing can have its issues in the worst case scenario. But when our keys are strings, we can use a data structure called a "trie" that is more reliable in time with data retrieval and has other uses besides data storage/retrieval.

§29.2 Basics of a Trie

So let's suppose we have a set of strings: $\{\text{ape}, \text{ball}, \text{atom}, \text{ate}, \text{bait}\}$. Here's what the trie would look like:

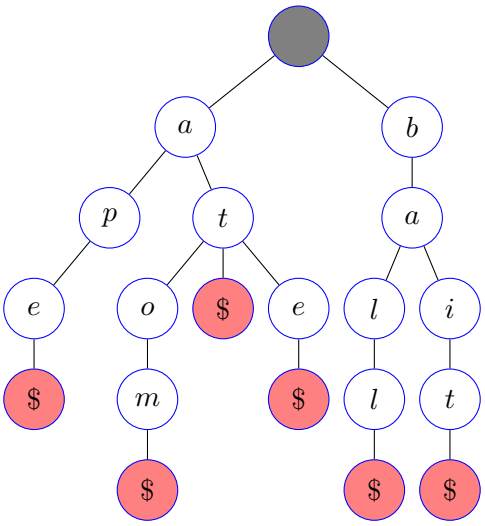


Basically, for every string in our set, you can spell out that string using a path that start at the root and traverses across nodes with characters. For example, "atom" is in our set and we can spell out atom by starting at the root and going to 'a' then 't' then 'o' then 'm.'

Also, as you can see for "ball" and "bait," we reuse the "ba" they share at the beginning because the two strings eventually become different and can branch off in their own paths. But there's no need to create "ba" twice, so we just have them share the same path until they branch off.

For worst case in space, suppose we have n strings, each of which are at most k characters. Then, space is $O(nk)$, since the worst case is no sharing of nodes. It also takes $O(nk)$ time to build the trie because we take each string and create nodes along the way for that string or check if the node necessary is already there.

As you can see, the ends of each word are at leaves...wait, not always. What if "at" was in our set? Then, the last character 't' would be at an internal node instead of a leaf. Well, remember Huffman? Everything was leaves. Why? Because it was prefix-free. So to make this prefix-free, let's add some special character we know we're not going to be using in the strings like \$ at the end of each word. So our set (with "at") is $\{\text{ape\$}, \text{ball\$}, \text{atom\$}, \text{ate\$}, \text{bait\$}, \text{at\$}\}$. Now, at\$ is no longer a prefix of atom\$. So our trie looks something like:



So the \$ (shaded in red) are always at leaves and tell us precisely when a word ends. Problem solved. Clearly, the number of leaves in the tree now is exactly equal to the number of strings we have. So if we have n strings in the tree, we have n leaves in the trie (when it's prefix free).

Note that adding these \$ isn't entirely necessary. We can also have an indicator of where a word ends by simply having a value (or data) in a node if it's the end of word and having said value be null otherwise.

§29.3 Operations

So we have 3 operations to implement:

- Get

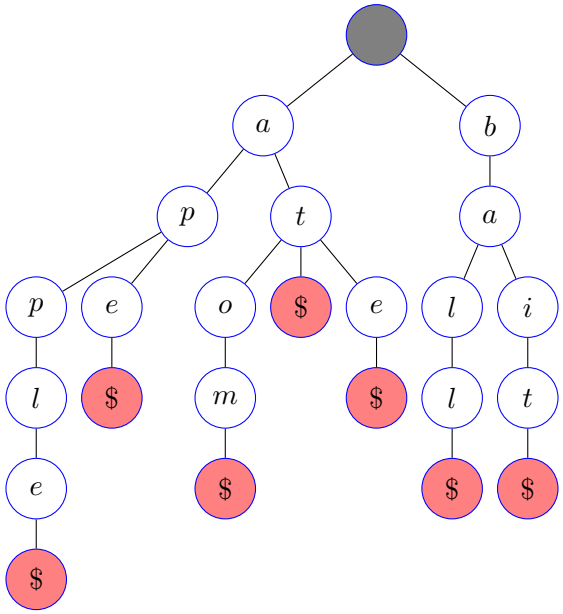
- Insert
- Delete

All three operations should be self-explanatory in what they're supposed to do. So it's a matter of how we would carry them out.

First, consider get. Pretty clearly, if we want to check if a word is in our trie, we use the characters in the word to traverse the nodes corresponding to the characters until we reach the end of the word (where the data is for that word) or we can't find the next character. So if we wanted to find "apple" in our trie, we see that we can start at the root and go to 'a.' From there, we can go to 'p.' But there's only 'e' as the next character from there and no 'p.' So there's no "apple" in our trie. On the other hand, we know "ape" is there because we can follow a sequence of nodes to make "ape." But we don't want to make the mistake of thinking a word is in the trie when it isn't. We know "ap" isn't in our trie, but the existence of a path "ap" could make us think so. That's where the $ comes in because it always marks the end of the word.

For all nodes, we can use an array to store all the children of the node, where "null" indicates that a child for that letter isn't there. For example, consider the first 'a.' It has a child of 'p' and 't' but nothing else. So the child array for 'a' will have nodes leading to its 'p' and 't' children but everywhere else will be null, since it has no other children. So we can determine whether the next character is within the trie in $O(1)$ time. So if we want to check whether a word is in the trie and the word is k characters long, it takes $O(k)$ time. Check each character in the string and see if the next node is there. If it is ever null, we know the next character doesn't exist in the trie.

Insert isn't too hard. Say we want to add "apple" to the trie. Basically, we check the characters one by one and see if they're already there or we need to add them. First 'a?' Already there, move to it. Next up is 'p,' which is already there, so we move to it. This 'p' doesn't have a 'p' child for us to move to next, so we create it. Similarly, we create 'l' then 'e' then '$' and then store whatever data we have for "apple" at that $ node.



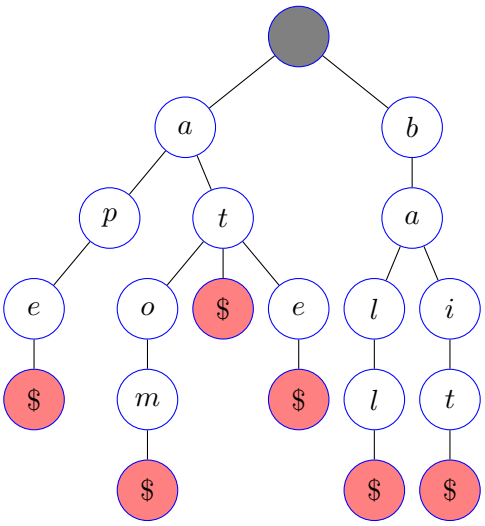
This is $O(k)$ time, just like insert.

But delete is going to be tricky. Suppose we want to delete "ball." Well, "bait" still needs the "ba." But everything after the two branch off is where we no longer needs stuff. So to delete "ball," we delete "ll$," which is the stuff after the branch point. If we wanted to delete "ape" though, there are multiple branching points to get to "ape," with other routes leading to "atom" and "apple."

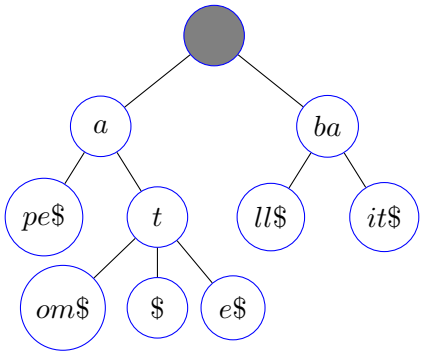
So the rule is, to delete everything after the last branching point. So for here, the last branching point for "ape" is after the 'p.' The reason why we do the last branching point is because the very last branching point means no more branching off after that. So we can rest assured no other string needs that branch for anything. But when we have a branch everything before is used by some other string like "ape" and "apple" both needing "ap." Delete is $O(k)$.

§29.4 Compressed Trie

Well, our trie is certainly going to take up space with all those nodes, so maybe we can save up on space? We can draw some inspirations from Huffman again. With Huffman, it was a full binary tree, meaning every internal node has exactly 2 children. What if we designed our trie so every internal node had at least 2 children? And we merged nodes together that only had 1 child? I mean, if you want to get to "ape," which branches off after 'a,' you have to get to p then e then \$. But there's no other branching path along the way, so we can just treat that entire path as one big node to save space. Okay, so we're going to compress the trie. Here it is again:



Now, for every stretch of consecutive nodes where there's no branching off anywhere in the stretch, we merge them into one node to represent that entire stretch. So the nodes 'b' and 'a' get merged into a 'ba' mega node. The 'i,' 't,' and '\$' get merged into 'it\$.' There are other instances of where we merge nodes, but here's what the new trie looks like in a compressed form:



Much better in terms of space. There were 20 nodes in the original trie and only 10 here in the compressed version. In fact, we can get a bound on space. Remember the full binary tree theorem, where a full binary tree with n leaves has $n-1$ internal nodes. But we occasionally do better than that because some internal nodes have more than 2 children, so the n leaves will be connected up in less internal nodes. Regardless, the upper bound for the number of nodes in a compressed trie will be $2n-1$, so a compressed trie takes $O(n)$ space, where n is the number of strings.

> **Warning 29.1.** Note that a compressed trie does NOT improve runtime, only space. Think about it, if you wanted to check if "ape" is there, when you get to the mega vertex with "pe\$," you still have to check all those characters in the mega node. So no time is saved when you still have to go through each character to see if you can continue further.

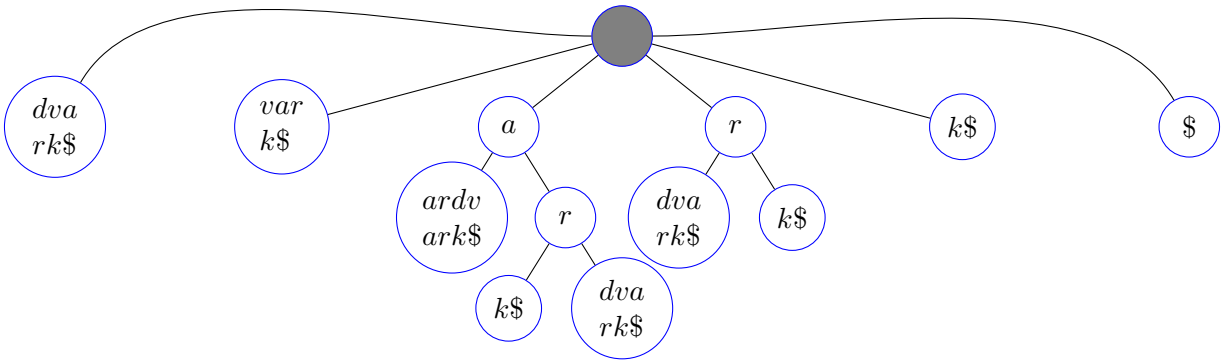
§29.5 Suffix Trie

First, we must properly define a suffix of a string.

Definition 29.2 (Suffix)**.** Suppose a string s has length L. Also, choose a starting position i, where $1 \leq i \leq L$ a suffix of s is a string that has all the characters from the ith position to the last position in s and in the same order they appeared in s.

For example, some suffixes of "computer" would include "uter," "mputer," and "er." It's okay if the suffixes are nonsensical words.

A suffix trie is a trie that contains all suffixes of a particular word. For example, let's make a suffix trie of the word "aardvark." We should put a \$ at the end for good measure. So the suffix that starts at the first character would be "aardvark\$" itself. The suffix that starts at the second character would be "ardvark\$." At the third? "rdvark\$." Regardless, all of these suffixes are going in the suffix trie (we'll include \$ to represent the empty suffix of the original word). This is what the suffix trie of "aardvark" looks like in its compressed form:



Since aardvark has 8 characters (but we're including \$), it should have 9 suffixes. Because we choose a starting position and then extend the string until we hit the end. So there's 9 leaves here. Check each path actually corresponds to a suffix of "aardvark."

If the suffix trie were compressed, it'd be $O(L)$ space (where L is the length of the string) because we have $L+1$ suffixes (which includes \$) and our analysis of the space in relation to the number of strings we have.

But if we were to not compress the tree, well, we have a suffix of length 1 (only including the last character), a suffix of length 2 (last 2 characters), suffix of length 3 (last 3 characters), and so on until $L+1$ characters (the suffix that is the whole string). So that's $\sum_{i=1}^{L+1} i = O(L^2)$ characters total and that many nodes in the worst case (when none of the nodes are reused for two or more suffixes).

However, we can create a suffix trie in $O(L)$ time using Ukkonen's algorithm. They say it's beyond the scope of the course though, but you're allowed to cite that algorithm for runtime in a homework or exam problem.

Suffix tries seem rather unintuitive in a way that makes you go "What's the purpose of them?" Well, we're going to see when we solve some problems.

§29.6 Problem Solving

Example 29.3 (Recitation)

Suppose we have N strings total, where the sum of lengths of all the strings is m. Design an $O(m)$ algorithm to find the longest prefix that all the strings share. For example, if our strings were $\{\text{shrub}, \text{shrink}, \text{shred}\}$, all the strings have a prefix of "shr" but not anything longer, so we would return "shr" here.

Here, we can see how tries can be useful for not just data storage with strings as keys but also string related problems. Let's take that example set and think about what the trie is like without having to draw it. So we all know they share "shr" at the start. So our trie is going to have "shr" from the root but without branching off, since all the strings have that. But the strings have different 4th characters, so they all branch off after that.

So I got it! Let's put all the strings in a trie (but add \$ to the end of each of the strings), which is $O(m)$ time total because we go through each string and process each character. The sum of the lengths of all the strings is the total number of characters, so we create that many nodes worst case (or check if we just reuse an existing node). Then, we go as far in to the trie as we can until it splits off into multiple branches. We know the first branching off point is the longest common prefix because the trie is saying "Okay, everything up until the first branching off point is the same for all our strings. But when we branch off, that means some strings will now have different characters."

The reason we need to add \$ at the end of each string is because we may need to stop prematurely if words end. For example, if "sh" was in our set of strings, the longest common prefix would only be "sh." If we didn't include \$, we would have no way of knowing that "sh" marked the end of some string in our collection and thus, we'd hit the first branching point at "shr" instead, which is wrong if "sh" is in there. However, with the \$, if "sh" is in there, we have a branch off at "sh" with one path leading to \$, so we'd be able to tell.

This search of the first branching point should take $O(a)$ time (where a is the length of the shortest string in our collection) because that's the worst case in how far deep we may have to check. Also, remember that all the strings can't all have anything in common once the shortest string ends for obvious reasons. Clearly, $a \leq m$ because the shortest string should be less than or equal to the total sum of the string lengths. So the $O(a)$ isn't going to change the $O(m)$.

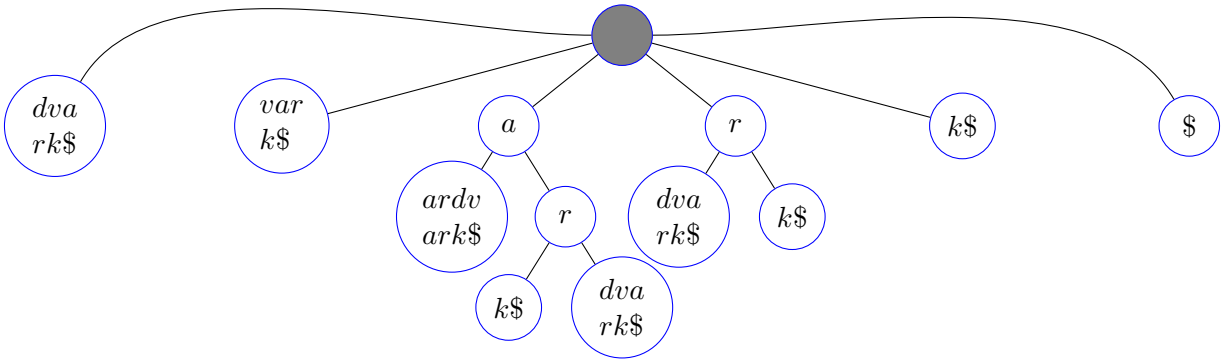
Example 29.4 (Recitation)

For a string s, design an algorithm to find the longest substring of s that appears more than once.

Let's take "aardvark" as an example. I'll just tell you that "ar" is the longest substring appearing more than once. But how do we make an algorithm that can get us the longest substring for an even bigger string? I guess we could try substrings of different sizes and count how many each time appears and take the longest that appears twice. No, that will take too long!

But we know every substring has to start somewhere in the string, right? What happens if we extend the substring all the way until it hits the end of the string? Take the first instance of the "ar" substring. We could extend that to get "ardvark." Now, it's a suffix of the string. So all substrings can be related to a suffix. Also, we can take a suffix and have its starting point be wherever. Then, we determine an endpoint within the suffix to create a substring. For example, to create the only "rdv," take the suffix "rdvark." Then, we say the 'v' is the endpoint to get "rdv."

So any substring of s is a prefix of a suffix of s. Go figure! Let's take that suffix trie of "aardvark" again:



We notice the path "ar" from the root leads to exactly 2 leaves and that "ar" as a substring appears twice. Using our earlier intuition about substrings, this makes sense. Since "ar" appears twice, there will be exactly 2 suffixes of "aardvark" beginning in "ar." Aha, so to count the number of times a specific substring appears in a string, we make a suffix trie. Then, we use that substring to navigate the trie (if we ever go off the trie because a certain character doesn't exist in the trie, then that substring doesn't appear in the trie at all). From there, we count how many leaf nodes we can get to. This is because the number of leaf nodes after the substring is equal to the number of suffixes starting with that substring, which in turn, is equal to the number of instances of that substring in the whole string.

Back to the problem at hand. What we do is search down the suffix trie (perhaps with BFS) and find the deepest branching point in terms of characters (and not nodes in compressed form). Because everything before the branching point corresponds to a substring and a branching point leads to at least 2 other leafs, which means at least 2 suffixes use that substring. So in our example, the longest repeated substring is "ar" because it leads to 2 different leaf nodes, but there's no deeper branching point.

Advice 29.5 — As you can see, suffix tries will be incredibly helpful in problems involving substrings. So their point isn't mostly for data storage but rather problem solving.

Example 29.6 (Class)

Design an algorithm to find the longest common substring between two strings. For example, the strings "aardvark" and "cards" both have "ar" as a substring, but the longest substring they have in common is "ard."

Okay, we have that intuition of a substring being the prefix of a suffix. Put \$ at the end of the first string, and put # at the end of the second string. Add all suffixes of the first string and all suffixes of the second string into a trie. Why do we add # at the end of the second string instead of \$? Because for a suffix in the trie, we want to be able to tell if the suffix came from the first string or the second string. The different special characters help make that distinction.

Once the trie is built, we'll see the deepest point we can go where from that point, we can reach both a \$ and a #. The intuition is similar to that of the prior problem. A substring can be extended into a suffix. In the example with "aardvark" and "cards," we know both have "ard." So in the trie, we'll have "ardvark\$" as a suffix and "ards#." So from the root of the trie, we have the path "ard" and then from there, we can eventually reach a \$ and a #.

Also, the \$ and # being different is extremely important. What if "ard" led to 2 leaves, but secretly, both instancs of "ard" came from the first string and none from the second string? Then, that doesn't count as a common substring because the second string doesn't have "ard." So we need the different special characters to know whether or not both strings have a substring appearing more than once.

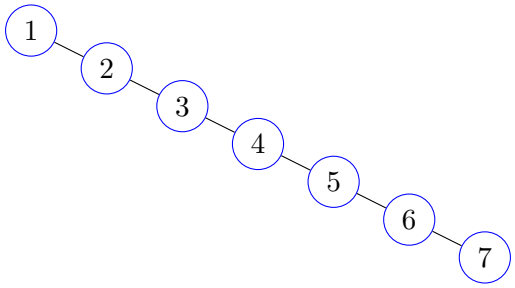
§29.7 Summary

- Tries are a way to store data with strings as keys.
- They use a tree like structure with nodes to spell out the strings in our collection of keys. Then, we can store the data relating to that key in the last node for said key.
- We can use \$ to make the trie prefix-free and more easily tell when a node corresponds to the end of some key.
- We can also compress the trie to merge a path of nodes with 1 child each into a single node to save space.
- If we had n strings with the longest string being k characters long, the trie takes up $O(nk)$ space. However, compressing it takes up $O(n)$ space.
- A suffix trie of a string is a trie containing all suffixes of that string.
- Suffix tries are great for string related problems, especially those relating to substrings.

§30 AVL Tree

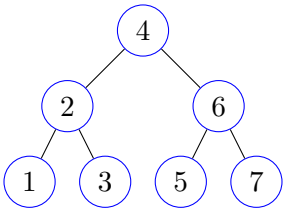
§30.1 Introduction

We know what a binary search tree is. The parent is always greater than its left child but smaller than its right child. We could implement a binary tree so it uses keys as the comparison between parent and child and then store the data (or value) associated with the key at the node for the key. Let's do some runtime analysis. How long could it take to find 7 in the given BST?



Well, this picture alone shows how inconvenient finding could be. Worst case, the tree is skewed a particular way to make us have to do $O(n)$ work. Want to insert 8? Similar problem. $O(n)$ work worst case. Deletion? Yep, $O(n)$ worst case.

But why not try to prevent the BST from being this skewed in the first place as we're trying to insert elements? After all isn't this following BST so much better than the ugly one above?



Introducing AVL trees! Which fix the balancing issues to prevent ridiculous skewing in the first place. But first, we have to get regular BST operations down.

§30.2 Review From CIS 1200

First, we have 3 different traversals of a BST:

- Preorder - parent, left subtree, right subtree
- Inorder - left subtree, parent, right subtree
- Postorder - left subtree, right subtree, parent

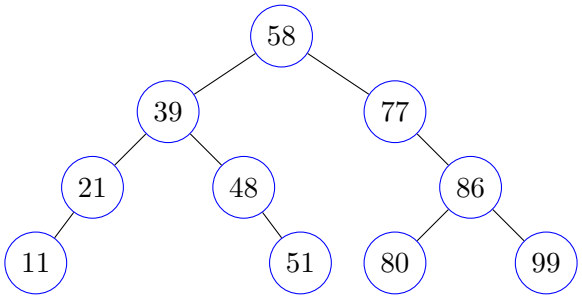
In particular, inorder returns the keys of a BST in increasing order.

Some basic operations:

- Getting the maximum in a BST - start at the root and just keep going right as much as you can until you reach a dead end trying to go right
- Getting the minimum in a BST - start at the root and just keep going left as much as you can until you reach a dead end trying to go left

Definition 30.1 (Successor)**.** The successor of a key v in a BST is the key that comes directly after v, if the keys were to be sorted in increasing order.

Take the following BST for example:

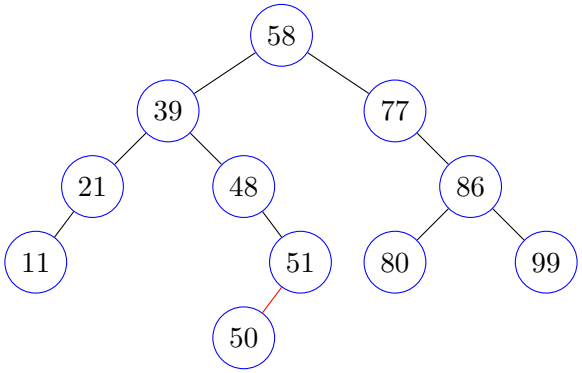


There are 3 cases to find the successor of v:

- Case 1: v has a right subtree - Then, the successor of v is the minimum of its right subtree. So the successor of 70 would be the minimum of the subtree rooted at 86, so the successor is 80.
- Case 2: v doesn't have a right subtree - Then, we consider the path from v up to the root of the BST. The first time we make a right turn, that's our successor. So to get the successor of 51, we go to 48, then 39, and then at 58, we make a right turn to get there. So the successor of 51 is 58. To get the successor of 80, we make a right turn immediately en route to the root, so the successor of 80 is 86.
- Case 3: v doesn't have a right subtree, but we don't make a right turn at all en route to the root - Then, v is the maximum of the entire BST and thus, doesn't have a successor. We can see this with 99, where we only go left to get to the root and never right.

Worst case, this is $O(n)$ when the tree is skewed.

To insert a new key K in the BST, we traverse the tree as if we were looking for K. When we come across an empty spot, we insert K there. Take the BST above and suppose we wanted to insert 50. We move around as if we were trying to search for 50: we go from 58 to 39 to 48 to 51. If 50 were there, it would be somewhere in the left subtree of 51. But the left subtree of 51 is empty space for now, so we can put 50 there:



Again, worst case, this is $O(n)$ when the tree is skewed.

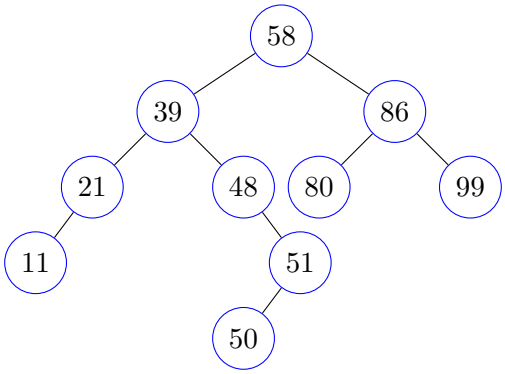
Deleting v is a bit trickier. Again, 3 cases.

- Case 1: v is a leaf - easy case, just delete v where it is
- Case 2: v has exactly 1 subtree - find the subtree of v that is still there and pull it up to where v is
- Case 3: v has 2 subtrees - swap v and its successor. For v's new location, it will always have 0 or 1 subtrees (never 2). So then, delete v in its new location in accordance with one of the first two cases.

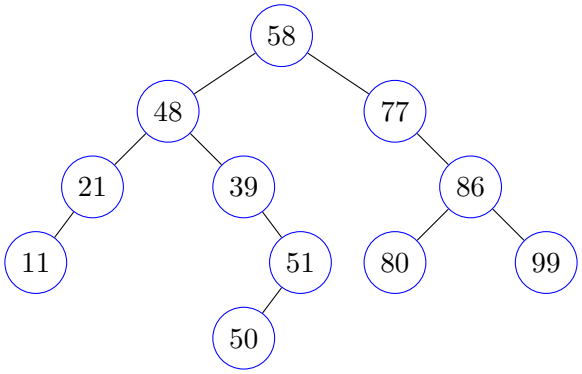
Some intuition for the third case. If v has two subtrees, its successor s is the minimum of the right subtree R. Also, clearly, s is smaller than anything else in R (easily seen through the definition of a successor), so it has no problem being the new root of R. As for s's original location, since it's the minimum of R, it cannot have a left subtree (since to get to a minimum, we going as left as possible, so s having a left subtree means we wouldn't be stopping at s to get the minimum). So s's original location has 0 or 1 subtrees, so that's why after the swap, we're in a situation described one of the first 2 cases.

Anyway, examples. If we wanted to delete 11, that's easy. Just delete it.

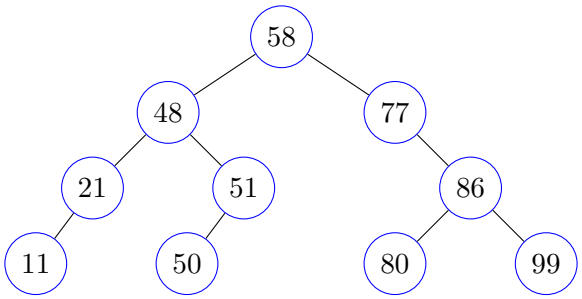
If we wanted to delete 77, we pull the subtree rooted at 86 and have it replace the spot at 77:



But what if we wanted to delete 39 instead? Again, this is the tricky case. So we see 39 has a successor of 48, so we swap those 2.



Now, 39 has 1 subtree, which is reminiscient of Case 2. So we pull the tree rooted at 51 up to where 39 is now.



Once again, $O(n)$ worst case. Okay, we did all the review. Now to the new stuff.

§30.3 Properties of AVL Trees

We're hoping that the height of tree is $O(\log n)$ worst case, instead of $O(n)$. We can get that done with AVL trees, which satisfy 2 key properties:

- It is a BST
- For any internal node, the height of its two subtrees differs by at most 1 **(Just so we are all on the same page, I will refer to this property as the "balance property")**

Let's take a look at the balance property. Define $f(h)$ to be the minimum number of nodes in an AVL of height h. So obviously, $f(1) = 1$ and $f(2) = 2$, as seen by the 2 AVL trees below:

Suppose we want to construct an AVL tree of height h. We'd need a root (contributes 1 node) and 2 subtrees. In order to correctly have the height of the entire AVL tree be h, the taller of the two subtrees at the root must have height $h-1$. So the other subtree has height at most $h-1$. But remember the balance property, so the other subtree could have height $h-2$, but not anything lower than that. For minimization purposes, we let the other subtree have height $h-2$. So the AVL tree with height h has the root (1 node), a subtree of height $h-1$ (has a minimum of $f(h-1)$ nodes), and a subtree of height $h-2$ (has a minimum of $f(h-2)$ nodes). So we have a recurrence

$$f(h) = 1 + f(h-1) + f(h-2).$$

We can prove by induction that $f(h) \leq 2^h$ for all $h \geq 1$.

Basically, this means an AVL tree with n nodes will have height $O(\log n)$. Perfect! But if we want to insert and delete stuff, we need to make sure we preserve the balance property.

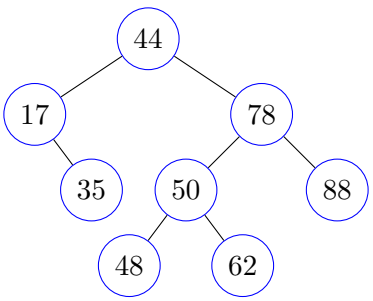
§30.4 Insertion into AVL Tree

Here's the process:

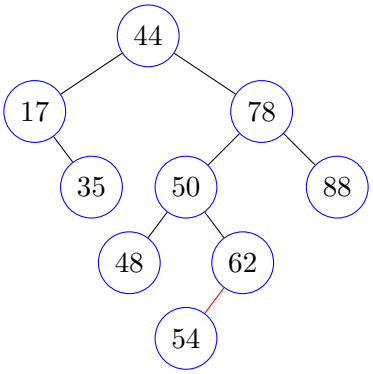
- Insert the new key K as you normally would in a BST.
- From K, follow the path to the root. The moment we encounter a node that violates the balance property, we do this tedious process to restructure the tree and restore balance:
  - Say z is the first node we encounter violating the balance property.
  - z will have 2 subtrees. Let y be the child of z in the taller subtree of z.
  - Let x be the child of y in the taller subtree of y.
  - Rearrange x, y, and z so their keys are in increasing order. Now, of those three nodes, rename them so a has the smallest key, b has the middle, and c has the largest.
  - Consider the 6 subtrees hanging from a, b, and c (the subtrees could be empty). Exactly 2 of those subtrees will have a, b, or c as a root and we ignore those, so we have 4 relevant subtrees. Label them T_1, T_2, T_3, and T_4 from left to right.
  - At z's location, replace it with b and have a as its left child and c as its right child. Then, have T_1 be the left subtree of a and T_2 be its right subtree. Finally, have T_3 be the left subtree of c and T_4 be its right subtree.

So the whole restructure process of the tree when we encounter an imbalance is quite annoying, but one restructuring is luckily all constant time. It turns out for an insert, we only need to do at most one restructuring (or sometimes, none at all). So the process is $O(\log n)$ for insert: $O(\log n)$ time to insert it normally and then $O(\log n)$ to go down the path and check for imbalances (possibly needing to restructure).

Of course, the list of instructions wouldn't be helpful without a demonstration. You can verify for yourself that this is actually an AVL tree:

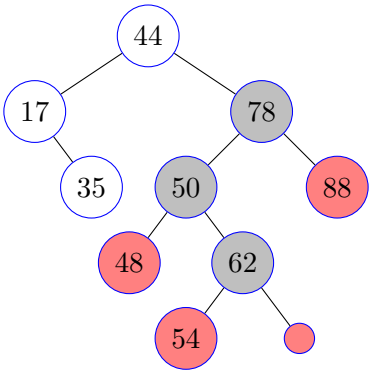


I want to insert 54, so we first do it normally as we would in a regular BST.



Okay, but now we need to check for balance, so we start at 54 and on the path to 44, we check for violations of the balance property.

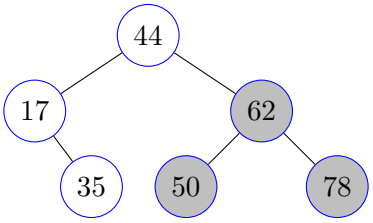
The first node in violation is 78, where its left subtree has height 3, but its right one has height 1. So we first define z, y, and x, as described in the instructions I gave. We have $z = 78$ then $y = 50$ then $x = 62$. Then, we rearrange these in increasing order, so know we have $a = 50$, then $b = 62$, then $c = 78$. Next step is to check the subtrees of each of these nodes. The three nodes that correspond to a, b, and c are shaded in gray, while the roots of the 4 relevant subtrees are shaded in red to make it easier to see:



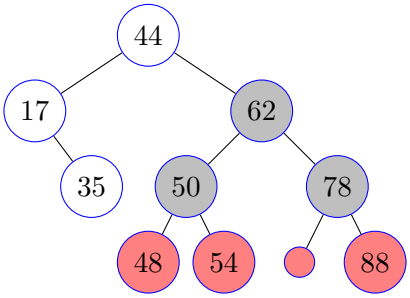
The right subtree of 62 is empty, but I drew a red node to symbolize it (empty subtrees count when we examine the subtrees of a, b, and c.

Anyway, we look at the roots of the 4 subtrees and arrange them from left to right. So T_1 would be the subtree with root 48, T_2 would be the subtree with root 54, T_3 would be the (empty) right subtree of 62, and T_4 would be the subtree with root 88.

Home stretch, we replace 78 with b (which is 62). Then, 62 will have 50 and 78 as its two children:



Now, we just attach the subtrees to 44 and 78 in the order we defined them in:



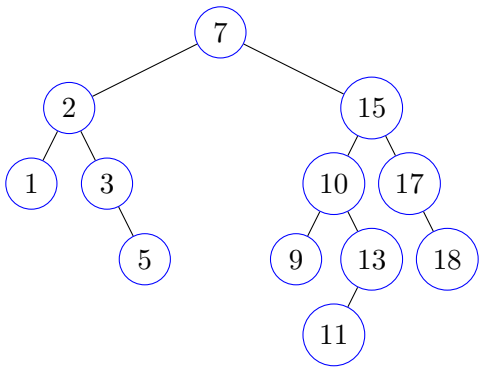
It isn't hard to check this new tree satisfies both necessary properties of an AVL tree, so we have restored balance.

§30.5 Deletion in an AVL Tree

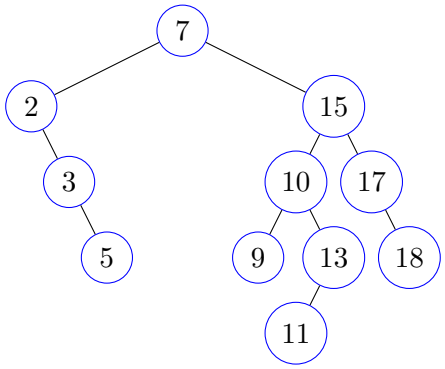
Deleting v follows a similar process to insertion:

- Delete as you would in a normal BST.
- Now the restoring balance part. In the location where you just deleted v, go up the path from that location to the root.
- When you encounter a node violating the balance property, call it z. Then, do the whole restore balance steps like we did in insert with y and x that are define the same, as well as a, b, and c and the 4 subtrees.
- However, unlike insertion, we may have to do more than one rebalancing. So even if you rebalance once, you still have to go up the path to the root and check every node along the way for balance.

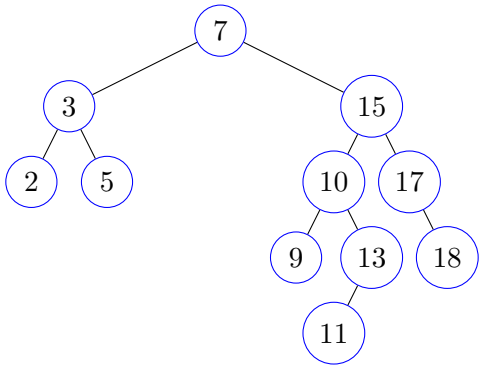
We're going to do an example that requires 2 rebalances. Let's have the following AVL tree and we want to delete 1:



So we delete 1 and then go from the path starting at 1 to the root. We go to 2, which is now imbalanced:

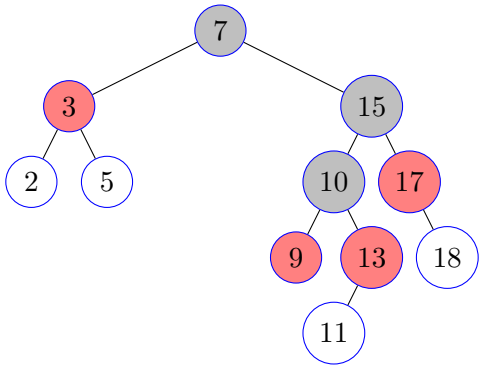


So we restore balance at 2: we see $z = 2$, then $y = 3$, then $z = 5$. So $a = 2$, $b = 3$, and $c = 5$. All 4 associated subtrees are empty, so it's just rearranging a, b, and c and we don't attach any children to a or c:

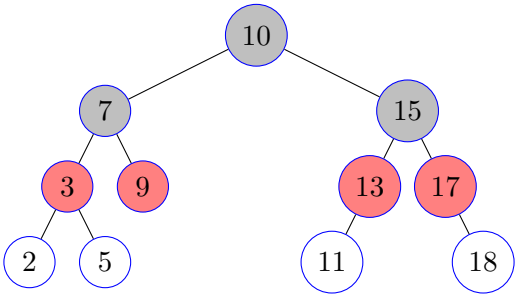


But our instructions say we're not done and we have to continue up the path to check for any more imbalanced nodes. So we go to 3 and then check 7. Yes, 7 is imbalanced because its left subtree has height 2, but its right one has height 4.

Same drill. We see $z = 7$, $y = 15$, and $x = 10$. Then, $a = 7$, $b = 10$, and $c = 15$. The 4 pertinent subtrees have its roots in red, while a, b, and c are in gray.



Then, we rearrange to get the final tree:



We made it to the root, restoring balance along the way, so now, we can rest assured there's no more rebalancing to do. We can check that the above tree is actually an AVL one.

Let's analyze runtime. Worst case of a normal delete would be $O(\log n)$ when we have to do Case 3 in not only locating the vertex to delete (which is $O(\log n)$) but also getting the successor (which is luckily also $O(\log n)$). So the first step is $O(\log n)$. Luckily, 1 rebalance is constant time. Worst case, we do rebalances all the way up, so that's $O(\log n)$ (corresponding to the height of the tree). So the deletion in an AVL tree is $O(\log n)$.

§30.6 Summary

Before I list the usual main points, I shall explain the intuition behind the restructuring. It's essentially a rotation of some section of the tree. Basically, if we have a node where one subtree has height h and the other has height $h + 2$, the rotation strives to make both have a height of $h + 1$ instead to restore balance. Now for the list of stuff:

- First, we saw how incredibly inefficient a BST could be because worst case, it's skewed to make searching, insertion, and deleting inconvenient.
- AVL trees fix this by having the balance property. For any internal node, the height of its two subtrees differ by at most 1. This leads to a height of $O(\log n)$.
- However, we need to maintain this balance property when we make insertions and deletions, so future operations like search, insert, and delete are still efficient.
- Insertions and deletions may require us to restructure (or rotate as another way of putting it) the tree. The process is annoying but can become straightforward with repeated practice.
- Search, insertion, and deletion are all $O(\log n)$ in an AVL tree. This is much better than the worst case of $O(n)$ for each operation in a regular BST.

§31 Test Section and Review Sheets

Before I get in to the details, keep in mind all these tests are closed notes. Again, remember that content may be different depending on the iteration of the course.

This is how it's going to work for each of the three tests: I'll briefly recap the main topics. Then, I'll have bullet point lists over all the content that was covered. Yes, the lists will be long, but I want to make sure I didn't miss any of the crucial points that they could test you on. Most of the bullet points should be stuff you're familiar with anyway, but it doesn't hurt to have a good refresher every couple of times in the days before the test. Go back to the appropriate sections if you don't feel confident on something. I'll also give test-specific strategies because there may be types of problems you're not used to, or non-standard ways of applying the material you learned.

§31.1 Good Sources to Review, In General

- Your notes
- Lecture notes
- Past HW problems (because intuition from such problems could help you with coming up with solutions)
- Recitation slides and problems
- Practice tests, if they post them
- Review Sessions

§31.2 Midterm 1

This is a 90 minute test, takes place just over a month in to the course.

Statistics from Fall 2023 iteration (out of 100 points):

- Mean: 53.71
- Median: 54
- Standard Deviation: 16.55

§31.2.1 Main Topics

- Asymptotic Analysis
- Recurrence Relations and Code Snippets
- Divide and Conquer
- Quicksort, Select
- Stacks, Queues, and Heaps

§31.2.2 List of Content

- Euclid's algorithm: $\gcd(a, b) = \gcd(b, c)$, where $c \equiv a \pmod{b}$ and $c < \frac{a}{2}$. Runs in $O(\log n)$.
- $f(n) \in O(g(n))$ means there exists a positive real c and positive integer n_0 such that $f(n) \leq cg(n)$ for all integers $n \geq n_0$
- $f(n) \in \Omega(g(n))$ means there exists a positive real c and positive integer n_0 such that $f(n) \geq cg(n)$ for all integers $n \geq n_0$
- $f(n) \in \Theta(g(n))$ means $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

- Use induction when they ask you explicitly for n_0 and c. Go about the inductive step first to get an n_0 that fits your needs. Then, choose a c so that your base case works.
- Polynomials of degree d are always $\Theta(n^d)$
- $\log(n)$ is faster than n, while polynomials are faster than exponentials.
- Use strong induction for POC in recursive algorithms.
- Binary search works only on sorted arrays and is $O(\log n)$: we repeatedly check the middle and use the sorted array part to smartly choose which half we should search now.
- Insertion sort takes elements 1 by 1 and puts them in the correct place among an already sorted group of elements.
  - It works due to induction in the elements we placed so far being sorted and inserting maintains the sorted aspect.
  - Runs in $O(n)$ best case (when the array is already in order)
  - $O(n^2)$ average and worst case (when you have to constantly move elements to the middle or back in order to properly insert them)
- For POC in loops, show three things: initialization, maintenance, and termination.
- Expansion Method: For a recurrence relation, keep plugging it in to itself, and it will eventually reach a base case.
- If they don't force you to expand for a recurrence relation, use Master Theorem as a shortcut.
- Code Snippets/Iterative Code:
  - For nested for loops where the loop increments by 1 each time, use superset-subset. Superset is usually simple by having each loop run to the maximum, but subset requires clever positioning of the loops in order for it to actually be a subset and get the desired Ω bound.
  - Otherwise, use the table and summation method. Check how each loop's variable increases per iteration and get a nested sum in terms of the iteration numbers. Be sure to get the upper bounds on the sum. Then, do the math.
- Merge sort divides an array in to two halves, sorts both halves recursively, and then merges the sorted parts.
  - It works because of strong induction and merge always works on two sorted arrays.
  - Has recurrence $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ without much nuance between a worst or best case scenario., which is $\Theta(n \log n)$
  - Merge sort is not in-place
- Quick sort is done by designating an element to be a pivot. Then, everything less than the pivot should be to the left of the pivot in the array and everything greater than the pivot should be to the right of the pivot in the array.
  - It works because the pivot always ends up in the right place after partitioning the array about it. Then, we can recursively solve both sides, where the numbers are on the correct side of the pivot.
  - Has expected time $O(n \log n)$, since that's the expected number of element to element comparisons
  - Has worst time $O(n^2)$ when the pivots are always on extreme ends
  - Quick sort is in-place
  - Deterministic quick sort involves a set rule for how we choose the pivot (always the first element, always the middle, etc), while randomistic quick sort chooses them at random

- Divide and conquer is a technique where we divide the problem into multiple parts (usually 2), solve each part recursively, and then combine the solved parts.
  - Some divide and conquer algorithms lack a combine step, usually when we're searching for something.
  - $O(\log n)$ is a good sign to try a recurrence along the lines of $T(n) = O(1) + T(\frac{n}{2})$, where we only care about 1 side of the array
  - $O(n \log n)$ is a good sign to try a recurrence along the lines of $T(n) = O(n) + 2T(\frac{n}{2})$, where we recurse on both sides and do $O(n)$ work to combine the solved parts
- Select/Partition: We can find the ith smallest element in $O(n)$ time, as well as determine which elements are greater and which are smaller than the ith smallest element.
  - We find the median of the medians M.
  - M always has $\frac{3n}{10}$ elements greater than it and $\frac{3n}{10}$ elements less than it. Then, we partition the array about M and recurse on the appropriate half.
- An array list is an array where we copy and paste elements into an array of different size for space issues.
- A linked list is a series of nodes, each containing data, where we have pointers to the next node.
- Amortized time for 1 operations is done by doing n operations and noting the average time. This helps offset when the worst case happens only every so often. One such example is occasional resizes with array lists.
- Array lists can waste a lot of space with unusued slots. As we fill in those unused slots and linked lists require more pointers and space for elements, the array list becomes better than a linked list, which occurs at the break-even point.
- There is a method we went over for resizing arrays: we resize arrays up when it gets full from size s to $2s$ (copying everything to the new array). When we have an array of size s, but only less than $\frac{s}{4}$ elements, we resize down from size s to size $\frac{s}{2}$, as to not waste space. But they could have a question where they introduce a different method of resizing like increasing/decreasing the size by 5 each time.
- Stack
  - Last In, First Out
  - Push inserts element to the back (amortized $O(1)$)
  - Pop removes last element and returns it (amortized $O(1)$)
  - Peek ($O(1)$)
- Queue
  - First In, First Out
  - Enqueue inserts element to the back (amortized $O(1)$)
  - Dequeue removes first element and returns it (amortized $O(1)$)
  - Peek ($O(1)$)
- A circular array helps for implementing queues in an array list because otherwise, there could be unused space in the front that we don't know about.
- Complete Binary Tree: All levels before the last level are totally full. In the last level, all nodes are as far left as possible.
- Max Heap is a CBT (complete binary tree) where each parent is greater than both its children. Min Heap is the same but each parent is less than both its children.

- Heaps are useful for when we want the maximum/minimum at any given point but don't care about the relative ordering of the rest
- Max Heapify is for when you aren't sure about the root node, but the two subtrees connected to the root are definitely max heaps. You sift the root node down appropriately and get a max heap. Runs in $O(\log n)$ time.
- Build max heap turns an unsorted array into a Max Heap. Read off the elements in the array into a heap without worrying about the heap invariant yet. Then, sift internal nodes down appropriately, starting from the rightmost one on the lowest level and then working your way left. Go to the next level up for more internal nodes when you're done with one level. Runs in $O(n)$ time.
- To insert a number n into a heap, place n in the next available spot in the lowest level. Then, sift n up appropriately. Takes $O(\log n)$ time.
- To delete the largest element in a max heap (or smallest for min heap), switch the root r with the rightmost node in the lowest level. Then, delete r, and sift the switched node down appropriately. Takes $O(\log n)$ time.
- To turn a max heap into an array that is sorted from lowest to highest, switch the root with the last element in the lowest level. Then, detach the last element from the tree, and sift the new root down appropriately. Rinse and repeat until all nodes are detached.
- Heap sort is a sorting algorithm by putting everything in a heap and repeatedly extracting the minimum. Takes $O(n \log n)$, but is not stable.

§31.2.3 Test Specific Strategies

- They'll provide Master Theorem. Use it as a timesaver for finding the runtime of a recursive algorithm. BUT THEY MAY SAY NOT TO USE IT, IN WHICH CASE, DO NOT CITE IT. In that case, use expansion, but you can use Master Theorem to check your work.
- When ask to expand, try to do 2-3 expansions before you generalize it (not including the line with the original recurrence relation).
- They'll provide log rules, but it helps to be familiar with them, so you know when to use where. It often helps to know $\log(a) + \log(b) = \log(ab)$ to combine logs, $\log(a^b) = b\log(a)$ to move exponents in and out of logs, and $a^{\log_a(b)} = b$ to remove logs in exponents.
- If they want an algorithm done in $\log(n)$ time, that's usually a sign to use Divide and Conquer, but recurse on only one side (like binary search) and not both. However, $n \log n$ usually necessitates both sides of the array
- If you have to write a recurrence relation or recursive algorithm, DO NOT FORGET TO WRITE THE BASE CASE.
- Remember that Merge Sort is done in $\Theta(n \log n)$. Sometimes the problem may be so much easier if the array were sorted, so if Merge Sort fits in the given time, go for it.
- Similar to above, remember that Select runs in $O(n)$. This may be useful for finding the median, min, or max if it fits in the given time as part of an algorithm.
- Remember common recurrences and their runtimes because it helps for designing new algorithms. You know Merge Sort is $\Theta(n \log n)$ and has the recurrence $T(n) = 2T(\frac{n}{2}) + \Theta(n)$. So if asked to design a $n \log n$ algorithm and you decide to divide and conquer on both halves of an array, make sure the combine step doesn't exceed $O(n)$ (else, the whole algorithm exceeds Merge Sort's time).
- $T(n) = 2T(\frac{n}{2}) + O(1)$ with $T(1) = O(1)$ implies $O(n)$ time
- $T(n) = T(\frac{n}{2}) + O(n)$ with $T(1) = O(1)$ also implies $O(n)$ time

§31.3 Midterm 2

This is a 90 minute test, takes place just over two months in to the course.

Statistics from Fall 2023 iteration (out of 100 points):

- Mean: 67.3
- Median: 67
- Standard Deviation: 17.9

§31.3.1 Main Topics

- Heap runtimes
- Huffman
- Graph representations and space
- BFS
- DFS
- Topological sorts
- Kosaraju
- Dijkstra

§31.3.2 List of Content

- Building a min heap is $O(n)$. Removing the min or adding an element is $O(\log n)$.
- Huffman:
  - Huffman is a "prefix-free tree," where the character nodes are at leaves, preventing a situation where two messages can create the same sequence of bits.
  - To create a Huffman tree, you repeatedly merge the two smallest nodes and traverses the edges from the root to get codes.
  - ABL is essentially an expected value for the length of the codes.
  - An exchange argument can be used to show that if $f(x) > f(y)$, then $P(x) \leq P(y)$ (where f denotes frequency and P denotes path length from root)
  - A min heap is generally used to help get the two smallest frequencies in an efficient manner. Huffman is $O(n \log n)$ with a heap
  - Try proof by contradiction for Huffman proof problems.
- A graph can be represented as an adjacency list (array of linked lists) or an adjacency matrix (2D array).
  - Adjacency matrices waste a lot of space for sparse graphs (don't have many edges compared to the number of vertices)
  - However, for dense graphs (lots of edges), the pointers required for each node in the linked lists in an adjacency list may take up more space than the matrix.
  - We generally use adjacency lists for graph algorithms because better runtime in going through all the neighbors of one particular vertex.
- For space problems, let a be the number of bytes for a vertex index, b be the number of bytes for a pointer, and c be the number of bytes for an edge weight. Then, an adjacency matrix takes up $c \cdot |V|^2$ bytes, while an adjacency list takes $b \cdot |V| + (a + b + c) \cdot |E|$. Note that for undirected graphs, you have to double E because an edge in an adjacency list has to appear twice for both directions.

- BFS:
  - Input is a graph (undirected or directed) and a source node. Output is the BFS tree of the source's connected component.
  - BFS traverses the graph one layer at a time, branching outwards. Nodes that are closer to the source get caught first.
  - It uses the FIFO properties of a queue to ensure nodes closer to the source are prioritized.
  - Nodes in layer k are a distance k away from the source.
  - Applications of BFS:
    - Reachability - finding all nodes for which there is a path from u to v
    - Shortest path (using the layers, but only in unweighted graphs)
    - Bipartitiness (using the layers)
  - BFS is $O(n+m)$, where $O(n)$ comes from processing every node and then each neighbor of each node is processed, which is $O(\deg(v))$ work for a node but $O(m)$ work overall by Handshaking lemma.
- DFS:
  - Input is any graph (undirected or directed). Output is a DFS forest.
  - DFS is another graph traversal but involves going as far as you can until you hit a dead end and then backtracking and exploring alternate routes that were missed along the way.
  - Start time in DFS is when you discover a node for the first time. Finish time occurs when there's no more undiscovered neighbors and then you backtrack.
  - Edges in the original graph fall in to one of 4 categories, using the DFS forest:
    - Tree edges: this edge is also in the DFS forest
    - Back edge: edge is $u \to v$, where u is a descendant of v
    - Forward edge: a non-tree edge in the form $u \to v$, where u is an ancestor of v
    - Cross edge: edge is $u \to v$, where u and v have no ancestor-descendant relationship. Usually between different connected components of the DFS forest, or different branches in the same connected component.
  - In an undirected graph, there are only tree and back edges.
  - Parentheses Theorem: Suppose $d[u] < d[v]$. Then, $d[u] < d[v] < f[v] < f[u]$ if v is a descendant of u. Or $d[u] < f[u] < d[v] < f[v]$ if u and v don't have an ancestor-descendant relationship.
  - White Path Theorem: In DFS forest, v is a descendant of u if and only if at $d[u]$, there is a path of white vertices from u to v
  - Above two theorems are super important for proofs on algorithms that will involve DFS (Kosaraju and Tarjan for example) and are good to cite in general when necessary.
  - Popular use of DFS: detecting cycles (which happens if and only if DFS detects a back edge)
  - DFS is $O(n+m)$ by the same intution as BFS.
- Topological Sorts:
  - Topological sort is a permutation of the vertices such that all edges point from left to right.
  - Only DAGs have topological sorts; cycles create a conflict with trying to order them.
  - Kahn's algorithm: Find all sources. Remove sources from the graph and add them to the sort. This opens up new sources, and then rinse and repeat. Runs in $O(n+m)$.
  - Proof involves that every DAG has a source and sink (by Maximal Path) and removing sources still gives a DAG.
  - Tarjan's algorithm: Run DFS on the graph. The topological sort is given by the finish times of the vertices, in decreasing order. Runs in $O(n+m)$.

  - Proof involves Parentheses and White Path, where if $u \to v$, then $f[u] > f[v]$. Cases on which of $d[u]$ or $d[v]$ is smaller.

- Kosaraju:
  - Input is a directed graph G and output is G^{SCC} (the graph of strongly connected components of G)
  - A strongly connected component is a subset of vertices in a directed graph where any two vertices in the subset are reachable from one another, and this subset is maximal.
  - Kosaraju is done by:
    * Doing DFS and noting finish times
    * Computing the transpose of G, call this G^T
    * Keep doing DFS on G^T but in order of finish times on G from highest to lowest to get each SCC.
  - Proof involves if we have $A \to B$ in G^{SCC}, then $f[A] > f[B]$. Proof is similar to Tarjan's in which we use Parentheses and White Path in tandem along with cases on which of $d[A]$ or $d[B]$ is smaller.
  - Runs in $O(n + m)$

- Dijkstra's
  - Input is a directed/undirected graph that is weighted and a source node. Output is the shortest path from source to all other vertices.
  - The idea involves creating a set S of nodes where we can be confident nodes in S have the shortest path found. S starts out by only having the source. But we slowly expand out and update possible paths as we go. The vertex v we find with the shortest distance thus far can safely go into S because essentially, all other routes into v already suck.
  - Dijkstra's is $O((V + E) \log V)$ specifically with a min heap: creating the min heap is $O(V)$, extracting min for each vertex is $O(V \log V)$, and possibly updating distances when we traverse each edge is $O(E \log V)$.
  - Doesn't work when we have negative edge weights. It will still terminate when that happens, but it won't always get you the best path.

§31.3.3 Test Specific Strategies

- Remember applications of each algorithm
- Think about how each part of the algorithm contributes to the runtime. For example, examining neighbors over all vertices is $O(m)$ by Handshaking Lemma.
- Be comfortable with the DFS process as there could be a question involving it. Also, be able to identify classify edges in the DFS based on the forest.
- Toposort is a good strategy for DAGs
- It may be helpful to use Kosaraju and then toposort G^{SCC}, since G^{SCC} is always a DAG. Especially useful in reachability in a directed graph.
- Transpose is also something good to think about when dealing with reachability in a directed graph. Because testing whether u can be reached from v in G is equivalent to whether you can get from u to v in G^T (which can be done with BFS)
- Don't merely memorize graph algorithm runtimes: understand why they are that way. Funny story: I remember saying to my neighbor before the midterm that they could ask something about different implementations of Dijkstra's because it could be done with an array, just not really used in the class. Lo and behold, they did ask such a question to troll those who merely memorized runtimes without understanding where the runtime comes from. We joked about it after the test.

§31.4 Final

This is a 2 hour test.

Statistics from Fall 2023 iteration (out of 150 points):

- Mean: 84
- Median: 87
- Maximum: 135
- Standard Deviation: 28.5

§31.4.1 Main Topics

- O, Ω, and Θ
- Recurrences
- Code Snippets
- Divide and Conquer
- Merge Sort, Quick Sort, Select
- Stacks and Queues
- Heaps
- Huffman
- BFS and DFS
- Toposort
- SCCs
- Dijkstra's
- MSTs
- Hashing
- Tries
- AVL Trees

§31.4.2 List of Content

Everything taught in the course is fair game on the final. Go back to the review sheets for midterm 1 and midterm 2 if you need a refresher on prior content. This list is mostly going to be on material after midterm 2, as well as any observations from hindsight on prior material.

- Minimum Spanning Trees:
  - Given a connected undirected weighted graph, find a connected subgraph that includes all vertices and has as little cost as possible
  - When we have positive edge weights, this subgraph is always a tree, as the presence of cycles creates redundancy.
  - When we have negative edge weights, the minimal subgraph isn't necessarily a tree, but Kruskal's and Prim's will give you the minimal subgraph that is specifically a tree.

  - Cycle Property - For any cycle in a graph, if there is a unique heaviest edge, it is never part of any MST.
  - Cut Property - Partition the set of vertices in to two subsets however you want, as long as all vertices are in one of the two subsets. Among the edges going from one subset to another, the lightest of those edges is in every MST.
  - Kruskal's Algorithm: (gets us MST)
    * Sort the edges by weight in increasing order. Go through the edges in increasing order and add them to the graph one by one, except when adding the edge creates a cycle.
    * Proof of correctness comes from the fact that we never want to create a cycle. Since the algorithm prioritizes lighter edges, we only get to the heaviest edges in the cycle later, only to see they create cycles and are not ideal but the cycle property. The lightest edge that is not used so far and won't create cycles is the most ideal by the cut property on the connected components we generated so far.
    * Union-Find datastructure prevents us from having to run DFS every time we want to add an edge to check if the added edge creates a cycle.
    * Runtime is $O(m \log n)$.

- Union-Find:
  - A collection of disjoint sets, where 1 element from each set serves to represent said set.
  - We can make singleton sets, merge sets with union, or find the set a specific element belongs to.
  - The rank of the set is the maximum number of pointers we need to follow from any element in the set to reach the root of the set (the element representing the set).
  - Elements have a series of arrows that eventually point to the root of the set. If we call find and traverse the path to get to the root, any element along that path can have its parent pointer changed to the root to speed up future find lookups. This is path compression.
  - When unionizing two sets S_1 and S_2, suppose S_1 has a smaller rank than S_2. To do the union, we change the parent pointer of S_1's root to that of S_2's root.
  - A set of rank r has at least 2^r nodes. So if we have a set with n nodes, find is $O(\log n)$ (and faster on future lookups with path compression).
  - Union-Find speeds up Kruskal's by having connected components created so far act as sets. If we try to add an edge and see the two endpoints are in the same set, that creates a cycle.

- Prim's Algorithm: (gets us MST)
  - Create a set S. Then, choose a starting vertex v to add to S. Rinse and repeat with the following steps:
    * Consider all edges emanating from a vertex in S. Choose the lightest one and add it to our MST.
    * Add the other endpoint of the edge to S.
  - Proof of correctness comes from repeatedly using the Cut Property with S as one of the subsets.
  - Runtime is $O(m \log n)$ due to the same logic as Dijkstra's.

- Hashing is a data storage and retrieval technique with expected $O(1)$ time. Expected runtime relies on SUHA. A hash table is an array, but there is the problem where two elements could end up in the same slot in the array because they have the same key.

- Open hashing:
  - Relies on using linked lists when two elements end up in the same slot of the array.
  - Worst case, we have all n elements in a linked list in the same slot, so retrieval would be $O(n)$.

- Closed hashing (or open addressing):
  - No two elements can be in the same position in the array. So if we want to add something but the position it's supposed to go in is already taken, we need to look for another one.
  - Linear probing: $h(k,i) = (h'(k)+i) \pmod m$, where m is the size of the table and $h'(k)$ is the home slot for k. Basically, we try $h(k,0)$ as the slot in the array to insert k. If that's already taken, we try $h(k,1)$. If that's already taken, we try $h(k,2)$. And so on.
  - Quadratic probing: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \pmod m$ for some constants c_1 and c_2.
  - Double hashing: $h(k,i) = (h_1(k) + i h_2(k)) \pmod m$.
- Tries are a data structure to store strings and data associated with each string.
  - We could add some sort of symbol like a \$ sign to the end of each string to make the trie prefix-free, if we wanted to. Not necessary though.
  - If we have n strings in the trie, each of which are at most k characters long, the trie takes $O(nk)$ space, uncompressed.
  - Compressing it (so that a series of consecutive nodes only have 1 child each get merged) means it takes $O(n)$ space.
  - Suffix trie is a trie where all the suffixes of some particular string are stored. Takes $O(n)$ space and Ukkonen's algorithm creates it in $O(n)$ time.
- AVL trees are binary search trees but are designed to be balanced.
  - For any internal node, the height of the left and right subtrees differ by at most 1.
  - The minimum number of nodes in an AVL of height h is $O(2^h)$. This means an AVL with n nodes is of height $O(\log n)$.
  - Search is $O(\log n)$ as a result of the height.
  - To insert m, we insert as we would normally in a binary search tree. However, we start from m and follow the path to the root and check if any nodes along the way violate the invariant of left and right subtrees differing by at most 1.
    - The moment we encounter such a node in violation, we label that node z.
    - Then, we label z's child that is in the taller subtree y.
    - Then, we label y's child that is in the taller subtree x.
    - Rearrange x, y, z in increasing order and relabel them a, b, c in that order.
    - Identify the 4 subtrees hanging from a, b, c (not included the trees rooted at b or c, themselves). Call these trees T_1, T_2, T_3, and T_4 from left to right.
    - Replace z with b and have b's two children be a then c. Have a's two children be T_1 then T_2. Have c's two children be T_3 then T_4.
  - To delete m, we delete it as we normally would in a binary search tree. Then, we follow the path from the deleted node to the root and check if we encounter any nodes along the way that violate the invariant. The moment we do, label that node z. Then, all the steps we did in insert follow.
  - Insert and delete are both $O(\log n)$.

§31.4.3 Test Specific Strategies

- During the last recitation, we did jeopardy in preparation for the final. Ask them to post the slides if they do something like that for practice.
- We also had PollEverywhere questions in the last few classes to review the intuition behind concepts. That is valuable as well.
- 2 hours is a lot less time than you think on this exam.

- There was high variation in the difficulty of the questions from my experience. You had some straightforward questions mixed in with some seriously challenging ones. Focus on securing the easy pickings before you attempt at the harder questions, where there's the risk of you wasting time because you didn't make much progress.
- But be careful with skipping too much or jumping all over the place when taking the test because you could forget about returning to prior questions that you initially skipped. That's why it helps to skim through the test again after reaching the end so that you didn't miss anything.
- If they put true/false or multiple choice questions at the beginning, read them carefully and know that they can be tricky. Helps to write a few words of justification on the side, even if you don't need to justify it. This is so you remember why you put the answer you did when checking your work.
- But if you keep going in circles, don't waste too much time on a single MCQ worth maybe 2 or so points when you could get points elsewhere.
- For my final, they straight up recycled 2 previous recitation problems, but you had to write the algorithm or proof of correctness from scratch and without citing the recitation. So recitation problems can be helpful review, but actually know why the solutions work and extract the intuition instead of merely memorizing the solutions.
- Be sure to be familiar with how you would do processes like Huffman, DFS traversal, building a max heap, hashing, rebalancing an AVL tree, etc manually. Especially be careful because these are processes where if you make a mistake on one step, you mess up the rest.
- Of course, know your graph algorithms, their runtimes, and their applications. Also, know why they work the way they do so you can modify them to solve a graph algorithm problem with unfamiliar circumstances.
- Remember when we discussed the various attributes of sorting algorithms like stability and in-place? Know that stuff.
- When your target runtime is expected, hashing is safe, but otherwise, it's not.
- Suffix tries could be useful for anything related to substrings in a string.