Eric Du CS 170

Notes **Efficient Algorithms and Intractable Problems** January 15, 2024

Header styling inspired by CS 70: `https://www.eecs70.org/`

1 Introduction

This is a compilation of my notes for CS170 during Fall 2023. The class was taught by Nika Haghtalab and John Wright.

1.1 Goals

- The goal of this class is to design and analyze algorithms:
  - Design: Acquire an algorithmic toolkit
  - Analysis: Learn to think analytically and undersatnd what our algorithm is doing.
  - Understand limitations: There are some some problems that are *very hard*, and it's important to think about what these limitations are. For instance, the fact that encryption is secure is a *scientific fact* that was proven by computer scientists.
  - Communication: Learn to formalize and communicate clearly about algorithms.
- There are going to be 3 fundamental questions we will ask when analyzing algorithms: Does it work? Is it fast? and can we do better?

1.2 Addition

- Let's go back to the addition of integers for instance. How fast is the algorithm that we learned in grade school?
- To answer this, we need to ask how many operations there are, how many computations there are, and stuff like that. In our case, there are at most n computations we have to take (we won't really care about carry over because that only adds a constant factor overhead, which we'll talk about later.) At the end of the day, there are about $2n$ to $3n$ computations.

1.3 Asymptotic Behavior

- In most cases, we only care about asymptotic behavior (i.e. what happens when n grows large). There will be "better" algorithms that work much better for large n, but are slower than some "worse" algorithms for small values of n.
- Formally, we define a function $R(N) \in O(f(N))$ means that there exists a positive constant k_2 such that $R(N) \leq k_2 f(N)$ for all N larger than some N_0. This is also why in the earlier example, we don't really argue between $2n$ and $3n$. So we'd say that addition runs in $O(n)$.

1.4 Multiplication

- Now let's look at multiplication. How many operations do we have here? Well, there are n^2 mulcipliations, with at most n carries, so it runs in $O(n^2)$.

- Now the question is, can we do better than this? This might be a hard question to answer, but we *can* show easily that we can't do better than $O(n)$. This is because we need at least n computations to see all the numbers, so our runtime must be *at least* $O(n)$.

- Another fun algorithm for multiplication is the *Egyptian/Russian Peasant Algorithm*, which goes like this:

  1) Halve the first number (floor divide) and double the second number until the first number is 0

  2) Remove any rows where the first column is even

  3) Add all remaining rows

  At home: try to prove that this is correct!

- There have been improvements!

  – Karatsuba (1960): $O(n^{1.6})$ (we will do this!)

  – Toom-3/Toom-Cook (1963): $O(n^{1.465})$

  – Schönhage-Strassen (1971): $O(n \log(n) \log\log(n))$.

  – Furer (2007): $O(n \log n \cdot 2^{O(\log^*(n))})$

  – Harvey and Van der Hoeven (2019): $O(n \log n)$

1.5 Divide and Conquer

- The idea here is that instead of solving a big problem, we can break up a big problem into smaller subproblems recursively. Eventually, we'll get to a point where the problem becomes trivial, at which point we're going to recurse back up.

- For multiplication, one way we can divide is by breaking up multiplication of two integers with n digits into multiplication of digits with $n/2$ digits. For instance:

$$1234 \times 5678 = (12 \times 100 + 34) \times (56 \times 100 + 78) = (12 \times 56) \cdot 10^4 + (12 \times 78 + 34 \times 56) \cdot 10^2 + 34 \times 78$$

  Here, we've already divided our problems! Notice that now all we have to compute is the products 12×56, 12×78, 34×56 and 34×78! We don't really care about the 10^4 and 10^2 terms, since those are equivalent to a left-shift by some number of digits.

- For any n digit number, we can then break it up as follows:

$$x_1 x_2 \cdots x_n = (x_1 x_2 \cdots x_{n/2}) \times 10^{n/2} + x_{n/2+1} x_{n/2+2} \cdots x_n$$

  So for two products, we can split as follows:

$$x \times y = (a \times 10^{n/2} + b)(c \times 10^{n/2} + d) = (a \times c)10^n + (a \times d + c \times b)10^{n/2} + b \times d$$

- With this, all we have to do now is repeatedly recurse until we get down to 1-digit numbers!

- Unfortunately, this algorithm is also $O(n^2)$! At every step, because we're doubling the number of computations, we are creating $O(n^2)$ 1-digit operations, so this algorithm takes $O(n^2)$ time to run. Next lecture, we'll look at how to improve this.

- How do we prove that this is $O(n^2)$? This can be shown using what's called the "tree method." The approach is to look at the subproblem size, and see how many layers we have until we get down to a problem of size 1. In our case, we see that for a problem of size n, we split it into four subproblems of size $n/2$ every time.

Since we halve the subproblem size at each layer, then we'll have $\log_2 n$ layers in total. Then, since each subproblem generates 4 more subproblems, then the number of subproblems in a layer t is given by 4^t. Therefore, the number of leaves is:

$$4^{\log_2 n} = n^{\log_2 4} = n^2$$

This is how we get our $O(n^2)$ bound.

Is it always true that $a^{\log_b c} = c^{\log_b a}$?

2 Divide and Conquer I, Asymptotics

- Last time, we talked about the motivations for studying algorithms: designing ways to solve problems efficiently.
- We also talked about addition and multiplication, and the latter in particular we saw two algorithms for it, but couldn't break the $O(n^2)$ runtime. Today, we'll try to beat this.

2.1 Karatsuba's Algorithm

- The main issue we ran into with the divide and conquer algorithm is that when we broke a problem down, we didn't actually simplify our life at all – we just created subproblmes for ourselves. What if we can create fewer than 4 subproblems? This is the key idea with divide and conquer: if we can use the results of subproblems to simplify computation at a given layer, we can generate an overall speedup.
- In Karatsuba's case, if we can write the term P2 + P3 in terms of P1 and P4, then we can simplify the number of computations needed.
- Karatsuba's trick is as follows: let's compute only three things:
  - Q1: $a \times c$
  - Q2: $b \times d$
  - Q3: $(a + b)(c + d)$

  Then, the idea is that the middle term P2 + P3 can be written in terms of these smaller subproblems:

  $$a \times d + c \times b = (a + b)(c + d) - ac - bd$$

  Therefore, our multiplication now looks like:

  $$\begin{aligned} x \times y &= \left(a \times 10^{n/2} + b\right)\left(c \times 10^{n/2} + d\right) \\ &= \underbrace{(a \times c)}_{\text{Q1}} 10^n + \underbrace{(a \times d + c \times b)}_{\text{Q3−Q1−Q2}} 10^{n/2} + \underbrace{(b \times d)}_{\text{Q3}} \end{aligned}$$

- With this algorithm, what is the runtime of this algorithm? The problem is almost the same, except we now only have 3 subproblems instead of 4. Therefore, if we employ the tree method (just as last time), then we'll see that we have $\log_2(n)$ layers, which means we have $3^{\log_2 n} = n^{\log_2 3} \approx n^{1.6}$, which is where we get our runtime from.
  - The Toom-3 algorithm mentioned last lecture also uses divide and conquer, but instead it reduces 9 problems into 5 subproblems, which gives a better bound.
- Note that Karatsuba's algorithm also doesn't care about the base we're working in: this works with any base. Note that the factor of $2^{n/2}$ that will appear in the division step is also just adding zeros, but in binary!

2.2 Asymptotic Notations (Formally)

- Suppose an algorithm takes $T(n) = 5n^2 + 20n \log n + 7$ microseconds. Then, we say that $T(n) \in O(n^2)$, or also sometimes written as $T(n) = O(n^2)$.
- Why do we employ this $O(\cdot)$ notation? It's because constants like 5, 20, 7 usually depend on the computer (say your computer is a year newer and has a faster CPU inside), so getting rid of these constants make comparing algorithms much easier. It's also often the case that the constants can be improved via some other smaller, less important optimizations.
- The formal definition of $O(\cdot)$ is as follows:

  Let $T(n)$ and $g(n)$ be functions of positive integers. Think of $T(n)$ as a runtime, so it's positive and increasing with n (usually). Then, we say "$T(n)$ is $O(g(n))$" if and only if for some large enough n, $T(n)$ is at most some constant multiple of $g(n)$. Mathematically:

  There exists c and $n_0 > 0$ such that for all $n \geq n_0$, $T(n) \leq c \cdot g(n)$.

  Note that this $g(n)$ also isn't unique! If a function is in $O(n^2)$, then it's also $O(n^3)$, and it's also in $O(2n^2)$. However, we generally ask for the simplest and the tightest bound, so while these are all technically correct answers, $O(n^2)$ is the "most correct".
- As an example, we can prove that $T(n) = 2n^2 + 2 \in O(n^2)$. Here, we can choose $n_0 = 1$, $c = 4$, so that we have:

  $$2n^2 + 2 \leq 4n^2$$

  All we have to do is prove that this inequality holds for all $n \geq n_0$. We can do this via derivatives, or other equivalent methods.
- There's an equivalent definition for a lower bound: we say that "$T(n) \in \Omega(g(n))$" if and only if there exists c and $n_0 > 0$ such that for all $n \geq n_0$, $c \cdot g(n) \leq T(n)$. Note that this inequality is reversed from the previous one.
- To test asymptotics, one way that's particularly efficient is using limits:

  $$\lim_{n\to\infty} \frac{T(n)}{g(n)} = \begin{cases} 0 & T(n) \in O(g(n)) \\ c \in \mathbb{R} & T(n) \in \Theta(g(n)) \\ \infty & T(n) \in \Omega(g(n)) \end{cases}$$

- The asymptotics of the geometric series is quite important for runtime analysis. Take any constant r and a function $T(n) = 1 + r + r^2 + \cdot + r^n$. We have that:

  $$T(n) = \begin{cases} \Theta r^n & \text{if } r > 1 \\ \Theta(1) & \text{if } r < 1 \\ \Theta(n) & \text{if } r = 1 \end{cases}$$

  Proof: Recall that for a goemetric series with $r \neq 1$, then we have:

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

  For $r > 1$, then this right hand side roughly evalutes to $\frac{r^{n+1}}{r} = r^n$, hence the $\Theta(r^n)$ bound. For $r < 1$, r^{n+1} is going to be very small, and hence $r^{n+1} - 1 < 0$. Overall, this means

  $$\frac{r^{n+1} - 1}{r - 1} \approx \frac{1}{1 - r}$$

  and since r is a constant we have $T(n) = \Theta(1)$. For $r = 1$, then we have $T(n) = n \in \Theta(n)$.

2.3 Formal Proof of Karatsuba's

- Now we formally look at Karatsuba's algorithm runtime. At each layer, we have 3 subproblems, each of size $n/2$. At every layer, we have to do a bunch of things: finding Q1, Q2, Q3, additions, and other stuff. However, all of this stuff runs in $O(n)$ time. To use a specific number, we'll say that the work is $20n$. Therefore, we have the following formula for $T(n)$:

  $$T(n) = 3T\left(\frac{n}{2}\right) + 20n$$

  This is a **recurrence relation.** We should also have a base case: $T(1) = O(1)$. Now, our goal is to find a closed form relation to $T(n)$.

  Now we look at this layer by layer. At the first layer, we have 1 problem, so that has $20n$ units of work. At the second layer, we have 2 subproblems, each of size $n/2$, so we have $3 \times 20 \times \frac{n}{2}$ amount of work from this layer. In general, we have:

  $$\text{work} = (\text{number of subproblems}) \times 20 \times (\text{subproblem size})$$

  This translates into the equation

  $$3^t \cdot 20\left(\frac{n}{2^t}\right)$$

  We now need to sum this for all t, so we have:

  $$\begin{aligned} T(n) &= \sum_{t=0}^{\log_2 n} 3^t \cdot 20 \cdot \left(\frac{n}{2^t}\right) \\ &= 20n \sum \left(\frac{3}{2}\right)^t \end{aligned}$$

  Now recall the geometric series we had from earlier: since $r = \frac{3}{2} > 1$, then the summation is $\Theta((3/2)^{\log_2 n})$, so overall:

  $$T(n) = 20n\left(\Theta\left(\frac{3}{2}\right)^{\log(n)}\right) = O\left(n\left(\frac{3}{2}\right)^{\log 3 - \log 2}\right) = O(n^{\log 3}) = O(n^{1.6})$$

2.4 The Master Theorem

- The tree method is useful, but slightly annoying to deal with sometimes. There's a theorem, called the Master Theorem, that tells us the runtime of $T(n)$, if we have a recurrence relation of the following form:

  $$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

  Then, we have:

  $$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log(n)) & \text{if } a = b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- The Master Theorem only tells us the runtime given this very specific recurrence relation. In fact, as you'll notice with the $O(n^d)$ term, it only works if the work at every step is polynomial. Also, if n/b is not an integer, we can force it to be an integer by enforcing a recurrence relation of the form:

  $$T(n) = a \cdot T\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$$

  Loosely, this is because constants don't matter, so small shifts (of less than 1) in the subproblem size doesn't really change the recurrence relation at all.

3 Divide and Conquer II

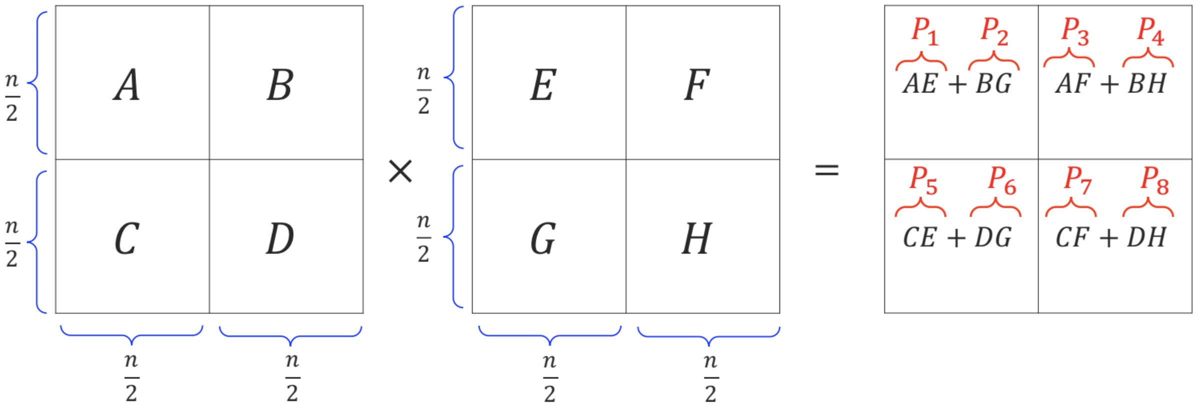
- Last time, we saw Karatsuba's $O(n^{1.6})$ multiplication algorithm, and did a formal recap on $O(\cdot)$ and $\Omega(\cdot)$ notation. We ended with recurrence relations and the master theorem.
- Remember that a is the number of subproblems we have, b is the factor by which the problem size shrinks, and n^d is the amount of computation per node. Note that if the work at every layer we have $c \cdot n^d$ work instead, then the runtimes are instead described by Θ relationships.
- One way to interpret the comparison between a and b^d is that when $a > b^d$, the tree is very large – there are a lot of subproblems. This means that most of the work is at the bottom of the tree, so we have an $O(n^{\log_b a})$ runtime.

  Conversely, when $a < b^d$, then the tree is very narrow, this means that most of the work is at the top of the tree (or alternatively, the work is concentrated in the combination step), which is why we have an $O(n^d)$ runtime.

  When the branches perfectly equals the amount of work per layer, then we get an $O(n^d \log n)$ runtime.

3.1 Matrix Multiplication

- We've shown that integer multiplication can be optimized, but what about matrix multiplication? As a reminder, the product of two $n \times n$ matrices X, Y is an $n \times n$ matrix Z, where every entry z_{ij} is the dot product of row i in X with column j in Y.
- What's the runtime of this multiplication? In this case, we want a runtime in relation to the number of rows and columns. For simplicity, we will deal with square matrices, so we only care about an n. We'll also assume that integers have a small number of bits, so they can be multiplied in constant time.
- To answer this, we first ask: what is the runtime of computing the dot-product of two vectors of size n? Well, there are n multiplications, so the dot product is $O(n)$. Then, since there are n^2 dot products (one per cell in the $n \times n$ matrix, then the total runtime will be $o(n \cdot n^2) = O(n^3)$.
- Let's try breaking up matrices into smaller matrices, of size $\frac{n}{2} \times \frac{n}{2}$:



$\frac{n}{2}$
A
B
C
D
×
E
F
G
H
=
$\underbrace{AE}_{P_1} + \underbrace{BG}_{P_2}$
$\underbrace{AF}_{P_3} + \underbrace{BH}_{P_4}$
$\underbrace{CE}_{P_5} + \underbrace{DG}_{P_6}$
$\underbrace{CF}_{P_7} + \underbrace{DH}_{P_8}$

  At each layer we generate 8 problems, of size $\frac{n}{2}$. At every step, we have work that's on the order of $O(n^2)$; this includes breaking up the problem, adding the matrices, and appending matrices to one another. This gives a recurrence relation:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

Using the master theorem, this still gives us $O(n^3)$ runtime! How do we make this better? We take inspiration from Karatsuba's algorithm, and notice that the way we optimized that was by reducing the number of subproblems. If we can get away with generating less than 8 subproblems, then we will get a speedup.

3.2 Strassen's Algorithm

- The approach is very similar to Karatsuba's, but this time for matrices. He went from 8 subproblems down to 7. He essentially creates the following problems:

$$\begin{aligned}
Q_1 &= A(F-H) \\
Q_2 &= (A+B)H \\
Q_3 &= (C+D)E \\
q_4 &= D(G-E) \\
Q_5 &= (A+D)(E+H) \\
Q_6 &= (B-D)(G+H) \\
Q_7 &= (A-C)(E+F)
\end{aligned}$$

  and combined them in the following way:

$Q_5 + Q_4 - Q_2 + Q_6$	$Q_1 + Q_2$
$Q_3 + Q_4$	$Q_1 + Q_5 - Q_3 - Q_7$

  Yeah, it looks insane and is insane, but it works.

- So now, our recurrence relation is:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

  which from the Master Theorem, this has runtime $O(n^{\log_2 7}) \approx O(n^{2.8})$. Recent advancements have brought this exponent down to $O(n^{2.34})$ approximately.

3.3 Median Selection

- Given an array S of n numbers and $k \in \{1, 2, \dots, n\}$, the k-select problem (written as SELECT(S, k)) asks us to find the k-th smallest element within S. $k = 1$ selects the minimum, $k = n$ selects the maximum, and $k = \lceil n/2 \rceil$ selects the median.

- A brute-force algorithm that runs in $O(n \log n)$ is to just sort the array, then output the k-th element. Can we do better? Specifically, can we do $O(n)$?

- For SELECT$(S, 1)$ an $O(n)$ algorithm would be to just keep track of the minimum element. If we were interested in SELECT$(S, 2)$, then we could run select(S, 1) twice – on the first ieration, remove the minimum element, then running SELECt$(S, 1)$ again returns the second smallest element.

- Does this produce an $O(n)$ algorithm for SELECT$(S, n/2$? No, because this would mean that we would be running SELECT$(S, 1)$ $n/2$ times, and if each is $O(n)$ then we get an overall runtime of $O(n^2)$.

- Let's take the Divide and Comquer approach: imagine we're given a pivot v (like in quicksort), and split the array into three pieces:
  - S_L : elements in S that are less than v
  - S_v : elements in S that are equal to v
  - S_R : elements in S that are larger than v

  Splitting elements in S into these two portions takes $O(n)$ time, since we can just run through the array and place elements accordingly. Effectively, this sorts our array into chunks, which allows us to recurse on smaller subproblems.

- Now suppose we want to compute SELECT(S, k):
  - If $k \leq \text{len}(S_L)$, then the k-th smallest elemet lives in S_L, so we run SELECT(S_L, k).
  - If $\text{len}(S_L) < k \leq \text{len}(S_L) + \text{len}(S_v)$, then the k-th smallest element is exactly the value of the pivot since k doesn't go past S_L and S_v combined, so we return v.
  - If $\text{len}(S_L) + \text{len}(S_v) < k$, then we recurse on S_R, but we have to be careful since the k-th smallest element would be the $k - \text{len}(S_L) - \text{len}(S_v)$ in S_R. So, we return SELECT$(S_R, k - \text{len}(S_L) - \text{len}(S_v))$

- In summary, our recurrence relation is given by:

$$T(n) = \begin{cases} T(\text{len}(S_L)) + O(n) & k \leq \text{len}(S_L) \\ T(\text{len}(S_R)) + O(n) & \text{len}(S_L) + \text{len}(S_v) < k \\ O(n) & \text{len}(S_L) < k \leq \text{len}(S_L) + \text{len}(S_v) \end{cases}$$

  Note that the lengths of S_L and S_R depend on the choice of the pivot, so how do we select a good pivot?

- Ideally, we'd want a pivot such that $\max(\text{len}(S_L), \text{len}(S_R))$ to be relatively small. This means that we effectively want to pick a pivot that's as close to the median as possible.

- Given an ideal pivot, where we select the median, then we know that $\text{len}(S_L) \leq n/2$ and $\text{len}(S_R) \leq n/2$. Therefore, we only recurse on half the array every single time, so our recurrence relation is:

$$T(n) \leq T\left(\frac{n}{2}\right) + O(n)$$

  By the master theorem, this would take $O(n)$ runtime. So in the best case scenario, this way of computing the median does take $O(n)$ time!

- We can't *always* pick the median (obviously), so let's relax our constraints a little and consider a pivot to be "good" when it's between the $\frac{n}{4}$-th smallest and $\frac{3n}{4}$-th smallest element. This makes the maximum length of S_L and S_r to be at most $\frac{3n}{4}$. This is because there are at least $\frac{n}{4}$ elements that are never looked at again if the pivot is good. The recurrence relation for this would be:

$$T(n) \leq T\left(\frac{3n}{4}\right) + O(n)$$

  which by the master theorem, would still be $O(n)$.

- So how do we pick a good pivot? We could just choose one uniformly at random from S, and we'll show later that this gives an $O(n)$ algorithm in expectation. The other alternative is to find a good pivot deterministically, but this is much harder and in practice it's slower than using a random pivot.

- Now let's prove the $O(n)$ expected runtime.

In our case, we know that a good pivot is chosen when our problem size drops to $3/4$ or less than the previous array size, so we can partition the tree layers into phases where this happens.

Because a new phase begins when the array size shrinks by $\frac{3}{4}$, then we know that at phase i, the problem size is at most $(3/4)^i \cdot n$, with equality if we've picked only good pivots up until this point. Let X_i be the random variable that denotes the length of phase i. Per node, we compute $c \cdot n$ operations, so the contribution at any phase i is given by $X_i \cdot c \left(\frac{3}{4}\right)^i \cdot n$. Therefore, now we have:

$$T(n) \leq \sum_{i=0}^{\log_{4/3}(n)} X_i \cdot c \left(\frac{3}{4}\right)^i n$$

Note that the problem size is $\left(\frac{3}{4}\right)^i n$.

Then, in expectation, we're looking for:

$$E[T(n)] \leq \sum_{i=0}^{\log_{4/3}(n)} E[X_i] \cdot c \left(\frac{3}{4}\right)^i n$$

So we want to find $E[X_i]$. Recall that X_i is directly a function of the number of times a bad pivot was chosen, which happens 50% of the time. This means that in expectation, the length of X_i is 2, since we expect that a good pivot is chosen 50% of the time (and thus we enter X_{i+1}). So in expectation, we have:=

$$E[T(n)] \leq \sum_{i=0}^{\log_{4/3}(n)} 2c \left(\frac{3}{4}\right)^i n$$

which you can check using the tree method, does give $O(n)$.

4 Polynomial Multiplication

- Earlier, we talked about multiplying numbers and also matrices quickly. Today, we'll talk about manipulating polynomials.
- Given two polynomials $p(x)$ and $q(x)$, what are the fastest algorithms that add and multiply the two polynomials together?

4.1 Representing Polynoimals

- Typically, we'd write a degree $n-1$ polynomial as $p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1}x^{n-1}$. This is called the **coefficient representation**, represented by an array of numbers $(p_0, p_1, \cdots, p_{n-1})$. Note that p_i are *real numbers*, not necessarily integers.
- We will think of n as being very large, say $n = 10^{10}$, while thinking that $p_0, \cdots, p_{n-1}$ to be very small. So we'll imagine that all these arithmetic operations are going to take $O(1)$ time.
- Goal: measure runtime as a function of n, not on the coefficients themselves.

4.2 Adding Polynomials

- Given two polynomials $p(x)$ and $q(x)$, we want to output $r(x) = p(x) + q(x)$, in its coefficient representation.
- How fast can we do this? To find the coefficients of $r(x)$, we just add $r_i = p_i + q_i$, and since each takes constant time, there are n additions, hence $O(n)$.
- This is like adding integers, but even simpler, since there is no carry over term!

4.3 Evaluating Polynomials

- Given an input $p(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_{n-1}x^{n-1}$, and a real number $\alpha \in \mathbb{R}$.
- We want to output $p(\alpha) = p_0 + p_1\alpha + p_2\alpha^2 + \cdots + p_{n-1}\alpha^{n-1} \in \mathbb{R}$.
- How fast can we do this? There are three algorithms that take $O(n^2)$, $O(n \log n)$, and $O(n)$ respectively. We'll first take a look at the $O(n^2)$ and the $O(n)$ ones.
  - Algorithm 1: We compute the terms individually and add them together:

$$\begin{array}{ll}
\ p_0 & \text{0 multiplications} \\
+\ p_1 \cdot \alpha & \text{1 multiplication} \\
+\ p_2 \cdot \alpha \cdot \alpha & \text{2 multiplications} \\
+\ p_3 \cdot \alpha \cdot \alpha \cdot \alpha & \text{3 multiplications} \\
\vdots & \\
+\ p_{n-1} \cdot \alpha \cdot \alpha \cdots \alpha & n-1 \text{ multiplications} \\
\hline
p(\alpha) & O(n^2) \text{ multiplications}
\end{array}$$

    Notice the repeated computation we have here: when we compute α^3, we don't actually need to multiply α three times, since we've already computed α^2 in the previous step.

  - Algorithm 2: Initialize an array A, and set $A[i] = \alpha \cdot A[i-1]$ for each $i = 1, 2, \ldots$. Therefore, $A = [1, \alpha, \alpha^2, \alpha^3, \cdots]$. So every step here takes 1 multiplication, so to compute the whole array A takes $O(n)$ steps. Now, if we were to evaluate now:

$$\begin{array}{ll}
\ p_0 \cdot A[0] & \text{1 multiplication} \\
+\ p_1 \cdot A[1] & \text{1 multiplication} \\
+\ p_2 \cdot A[2] & \text{1 multiplication} \\
+\ p_3 \cdot A[3] & \text{1 multiplication} \\
\vdots & \\
+\ p_{n-1} \cdot A[n-1] & \text{1 multiplication} \\
\hline
p(\alpha) & O(n) \text{ multiplications}
\end{array}$$

    Therefore, this will take $O(n)$ total steps, hence an $O(n)$ runtime.

    Note here that n (the problem size) refers to the length of the polynomial, and not the size of the coefficients p_i. We assume these to be small for our analysis, but this is not true in practice, and these computations will add up.

4.4 Multiplying Polynomials

- Given two polynomials $p(x)$ and $q(x)$, we want to output $p(x) \cdot q(x)$. For instance, $p(x) = (7 - 5x), q(x) = (1 + 3x + 2x^2)$. If we were to do this by hand, we find that $p(x) \cdot q(x) = 7 + 26x + 29x^2 + 10x^3$.
- How fast can we do this? $O(n^2)$. This is because for every coefficient in p we need to perform n multiplications (one for every coefficient in q), and since p has n coefficients then there are n^2 total multiplications. Hence, the runtime is $O(n^2)$.

  Note also that in doing so, we also increase the degree of the product, with it being a degree $2n-2$ polynomial.

Our goal for the rest of today's lecture is to improve this to $O(n \log n)$ time.

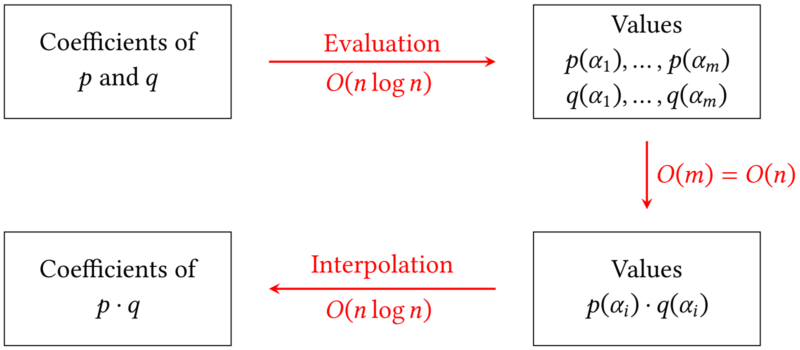
- To do this, we use the fact that n points determine a degree $n-1$ polynomial (recall this from cs70 notes) In other words, if $p(x)$ is degree 1, then 2 points suffice; if $p(x)$ is degree 2, then we need 3 points, and so on.
- This is useful because instead of representing polynomials by their coefficients, we can instead represent them by values given certain inputs. So given some points $\alpha_1, \alpha_2, \dots, \alpha_m \in \mathbb{R}$, the value representation of $p(x)$ is given by $(p(\alpha_1), p(\alpha_2), \dots, p(\alpha_m))$.
- Using the previous fact, as long as $m \geq n$, then the polynomial is uniquely determined. For us, a typical choice of m is $m = O(n)$.

4.4.1 Adding and Multiplying with Value Representation

- Given two polynomials in their *value representation*, how would we add these two polynomials together? We can just add these two values together, and output $(p(\alpha_1) + q(\alpha_1), \dots, p(\alpha_m) + q(\alpha_m))$. This takes $O(n)$ time.
- For multiplication, we output the product of the values: $(p(\alpha_1) \cdot q(\alpha_1), \dots, p(\alpha_m) \cdot q(\alpha_m))$. This is also $O(n)$. However, one thing to note with multiplication is because the degree of the product changes, we need $m \geq 2n - 1$ in order for the result to uniquely specify the product.
- The takeaway is that multiplication is much faster in the value representation compared to the coefficient representation!

4.5 Fast Polynomial Multiplication

- The last section motivates a scheme where we multiply polynomials using their value representation rather than their coefficient representation. Therefore, we need the following scheme:


Coefficients of p and q
Evaluation
$O(n \log n)$
Values $p(\alpha_1), \dots, p(\alpha_m)$ $q(\alpha_1), \dots, q(\alpha_m)$
$O(m) = O(n)$
Values $p(\alpha_i) \cdot q(\alpha_i)$
Interpolation
$O(n \log n)$
Coefficients of $p \cdot q$

If we can do this whole sequence, then that gives us an efficient multiplication algorithm. However, the evaluation step still takes $O(m \cdot n) = O(n^2)$ time, so how can we possibly get this down to $O(n \log n)$? The secret lies in how we pick $\alpha_1, \dots, \alpha_m$ – it is possible to pick them in such a way that the evaluation step takes $O(n \log n)$ time. Same goes for interpolation.

How do we pick $\alpha_1, \dots, \alpha_m$? We use complex numbers!

4.6 Complex Numbers

- A complex number is any number of the form $a+bi$, where $i = \sqrt{-1}$. a represents the real part, and b represents the imaginary part of the number. We can add complex numbers:

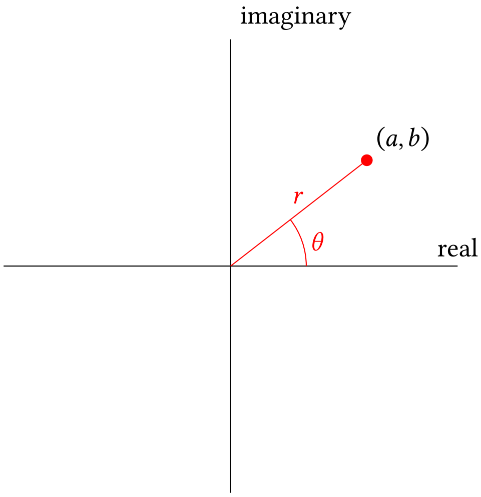
$$(1 + 2i) + (3 + 4i) = (1 + 3) + (2 + 4)i = 4 + 6i$$

so the real parts add and the imaginary part also adds. To multiply:

$$(1 + 2i) \cdot (3 + 4i) = 1 \cdot 3 + 1 \cdot 4i + 2i \cdot 3 + 2i \cdot 4i = 3 + 10i - 8 = -5 + 10i$$

Recall that since $i = \sqrt{-1}$, then $i^2 = -1$.

- We can also represent complex numbers on the complex plane. A number $a + bi$ corresponds to (a, b) on the complex plane. We can also represent it using polar coordinates, using a radius r and an angle θ



With this construction, we can relate polar to cartesian with the relations

$$a = r \cos\theta$$
$$b = r \sin\theta$$

For today, we'll only consider points with $r = 1$. This means that we basically forget about r, and only worry about θ. So with $r = 1$, then we have:

$$a = \cos\theta$$
$$b = \sin\theta$$

And since $r = 1$, we'll be dealing with points on the **unit circle**.

- Consider two complex numbers, the first specified by θ_1, and the other specified by θ_2 (both having $r = 1$). Then, we define the product of the two to be specified by $\theta_1 + \theta_2$. To multiply, we just add the angles.

4.6.1 Roots of Unity

- The n-th root of unity is a solution to the equation $x^n = 1$. So the second roots of uhnity are solutions to the equation $x^2 = 1$, which are ± 1. The 4-th roots of unity are solutions to $x^4 = 1$, which are $\{1, -1, i, -i\}$.
- In general, the n-th roots of unity will have n distinct solutions.
- Graphically, the n-th roots of unity correspond to n equally spaced points placed on the unit circle.
- When we talk about the roots of unity, we will use ω_0 through ω_{n-1} to label them. In particular, we'll focus on ω_1.
  - Note that ω_1 always sits at an angle of $\frac{2\pi}{n}$ for the n-th roots of unity, due to the even spacing.
  - Note that $\omega_2 = \omega_1 \cdot \omega_1$, since multiplying is equivalent to adding the angles together. Therefore, we have the relation that $\omega_i = \omega_1^i$, which we will call the **Generator Fact**.

- So this gives us a nice formula for the n-th roots of unity: they will always sit at angles $k\theta$, where $\theta = 2\pi/n$. As angles, this is represented as the set: $\{\cos(k\ell) + i\sin(k\ell) | \ell = 0, 1, \dots, n-1\}$.

4.6.2 Square Roots

- When we take a square root, remember that they always come in pairs of $\pm\sqrt{a}$. So to get the second roots of unity, we find the square roots of 1, which are ± 1. To get the 4-th roots of unity, then we just need to take the square roots of the previous roots of unity.

  In general, if we take the square roots of the n-th root of unity, then we generate the $2n$-th root of unity. Conversely, if we square the roots of unity then we get the $n/2$-th roots of unity.

  This is the magical fact that we will leverage for the following lecture: squaring the n-th roots gives us the $n/2$-th roots of unity. This is not true for most numbers! For instance, squaring the set $\{1, 3, 5, 7\}$ gives $\{1, 8, 25, 49\}$, which still contains the same number of elements as the original set!

4.7 Fast Polynomial Multiplication Algorithm (Preview)

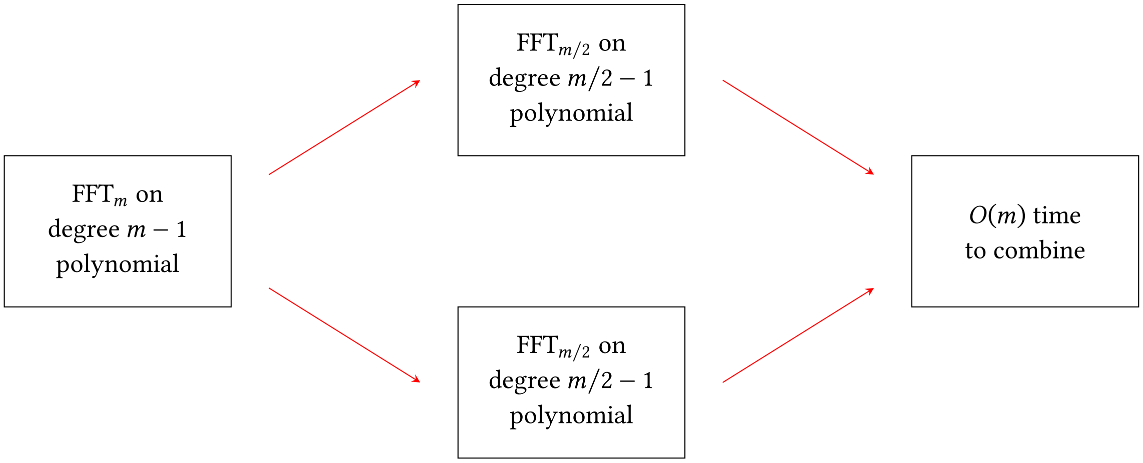
- Recall the evaluation step in our polynomial multiplication: $p \cdot q$ is degree $2n - 2$, so we need $m \geq 2n - 1$. Let m be the first power of 2 such that $m \geq 2n - 1$. Then, we will evaluate p and q on the m-th roots of unity in time $O(m \log m) = O(n \log n)$, using the **Fast Fourier Transform**.

5 Polynomial Multiplication II

- As a recap, we have two polynomials p and q, both of degree $n - 1$, and we want to multiply them.
- We saw the *coefficient representation*, where we specify its coefficients. So for p, we have $(p_0, p_1, \dots, p_{n-1})$. Last lecture, we saw $O(n^2)$ algorithm to multiply the polynomials using the coefficient representation.
- We also saw the *value representation*, where instead of giving the polynomial itself we give a set of m points that the polynomial passes through. As long as $m \geq n$, then this set of points fully specifies the polynomial. With this representation, we saw that polynomials can be multiplied in $O(n)$ time.
- So our main question was: is there a way for us to use the value representation to speed up the coefficient representation multiplication?
- Roots of unity: the set of points we will evaluate our polynomials on.

5.1 Fast Fourier Transform

- Input: m, a power of 2, a $p(x) = p_0 + p_1 x + \cdots + p_{m-1}x^{m-1}$. Our goal is to evaluate $p(\omega_0), p(\omega_1), \dots, p(\omega_{m-1})$.
- We will use a divide and conquer approach:


FFT$_m$ on degree $m-1$ polynomial
FFT$_{m/2}$ on degree $m/2-1$ polynomial
FFT$_{m/2}$ on degree $m/2-1$ polynomial
$O(m)$ time to combine

This will give us a recurrence relation $T(m) \le 2 \cdot T(m/2) + O(m)$, and from the master theorem this gives an $O(m \log m)$ runtime.

5.2 Divide and Conquer

- To figure out how to divide, let's first write out $p(x)$:

  $$p(x) = p_0 + p_1x + p_2x^2 + p_3x^3 + \cdots$$

  Let's split p into two parts, based on the parity of the exponent. We'll call these the even and odd halves:

  $$p_E(x) = p_0 + p_2x^2 + p_4x^4 + \cdots + p_{m-2}x^{m-2}$$
  $$p_O(x) = p_1x + p_3x^3 + \cdots + p_{m-1}x^{m-1}$$

  But notice that we can rewrite the even part a little bit:

  $$p_E(x) = p_0 + p_2(x^2) + p_4(x^2)^2 + p_6(x^2)^3 + \cdots = \text{Even}(x^2)$$

  But this looks like a polynomial with coefficietns $p_0, p_2, p_4, \ldots$ evaluated on x^2! We will call this polynomial Even(z). For the odd part, we factor out an x :

  $$p_1x + p_3x^3 + \cdots = x(p_1 + p_3x^2 + p_5x^4 + \cdots) = x \cdot \text{Odd}(x^2)$$

  The key to note here is that we have a recursion in the fact that we can express the evaluation of the polynomials at the n-th step as a computation involving another polynomial but evaluated on x^2. So, we have

  $$p(x) = \text{Even}(x^2) + x \cdot \text{Odd}(x^2)$$

  The degree of the even part is $(m-2)/2 = m/2 - 1$, and the same goes for odd part.

- Now if we want to compute p on ω_i, then we have $p(\omega_i) = \text{Even}(\omega_i^2) + \omega_i \cdot \text{Odd}(\omega_i^2)$.

- Because we are squaring the arguments, then this means that we are evaluating the Even and Odd parts on the $m/2$-th roots of unity, because of our magical fact!

- So to compute $p(\omega_i)$, we recursively evaluate the even and odd parts at the $m/2$-th roots of unity: $\alpha_0, \alpha_1, \ldots, \alpha_{m/2-1}$.

- To combine, all we need to do is multiply the odd part by ω_i, then add the two parts together. So, the combination step only takes $O(m)$ work. So this fully gives our recursion $T(m) \le 2 \cdot T(m/2) + O(m)$, hence the $O(m \log m)$ runtime.

5.3 Fast Interpolation

- An update on our algorithm scheme:

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{Coefficients of}\\ p \text{ and } q\end{array}} \arrow[r, "\text{Evaluation}", "\begin{array}{c}O(n\log n)\\ \text{via FFT}\end{array}"'] & \boxed{\begin{array}{c}\text{Values}\\ p(\alpha_1),\dots,p(\alpha_m)\\ q(\alpha_1),\dots,q(\alpha_m)\end{array}} \arrow[d, "O(m)=O(n)"] \\
\boxed{\begin{array}{c}\text{Coefficients of}\\ p\cdot q\end{array}} & \boxed{\begin{array}{c}\text{Values}\\ p(\alpha_i)\cdot q(\alpha_i)\end{array}} \arrow[l, "\text{Interpolation}"', "O(n\log n)"]
\end{tikzcd}$$

- Now we want to figure out the interpolation step: given $r(\omega_0), r(\omega_1), \dots, r(\omega_{m-1})$, we want to get back the coefficient representation of $r(x)$. This is called the inverse FFT.

- It turns out that when we do the Fourier transform, we basically get the inverse Fourier transform for free. Recall the Fourier transform:

$$p(\omega_\ell) = \sum_{j=0}^{m-1} p_j(\omega_\ell)^j$$

  This equation comes from replacing all x 's with ω_ℓ. Then, the inverse Fourier transform is written as follows:

$$p_\ell = \frac{1}{m}\cdot\sum_{j=0}^{m-1} p(\omega_j)\cdot(\omega_{m-\ell})^j = \frac{1}{m}\cdot q(\omega_{m-\ell})$$

  Here, $q(x) = p(\omega_0) + p(\omega_1)x + \cdots p(\omega_{m-1})x^{m-1}$. So in essence, this is basically another polynomial evaluation, which we already know happens in $O(m\log m) = O(n\log n)$ time!

- So, now let's look at the completed diagram:

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{Coefficients of}\\ p \text{ and } q\end{array}} \arrow[r, "\text{Evaluation}", "O(n\log n)"'] & \boxed{\begin{array}{c}\text{Values}\\ p(\alpha_1),\dots,p(\alpha_m)\\ q(\alpha_1),\dots,q(\alpha_m)\end{array}} \arrow[d, "O(m)=O(n)"] \\
\boxed{\begin{array}{c}\text{Coefficients of}\\ p\cdot q\end{array}} & \boxed{\begin{array}{c}\text{Values}\\ p(\alpha_i)\cdot q(\alpha_i)\end{array}} \arrow[l, "\text{Interpolation}"', "\begin{array}{c}O(n\log n)\\ \text{via inverse FFT}\end{array}"]
\end{tikzcd}$$

  So this shows that polynomial multiplication can be done in $O(n\log n)$ time!

5.4 The Matrix Viewpoint

- There's another way to represent $p(x)$, as a column vector:

$$\begin{bmatrix} p(\omega_0) \\ p(\omega_1) \\ p(\omega_2) \\ \vdots \\ p(\omega_{m-1}) \end{bmatrix} = \begin{bmatrix} p_0 + p_1\omega_0 + p_2\omega_0^2 + \cdots + p_{m-1}\omega_0^{m-1} \\ p_0 + p_1\omega_1 + p_2\omega_1^2 + \cdots + p_{m-1}\omega_1^{m-1} \\ \vdots \\ p_0 + p_1\omega_{m-1} + p_2\omega_{m-1}^2 + \cdots + p_{m-1}\omega_{m-1}^{m-1} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \omega_0 & \omega_0^2 & \cdots & \omega_0^{m-1} \\ 1 & \omega_1 & \omega_1^2 & \cdots & \omega_1^{m-1} \\ 1 & \omega_2 & \omega_2^2 & \cdots & \omega_2^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{m-1} & \omega_{m-1}^2 & \cdots & \omega_{m-1}^{m-1} \end{bmatrix}}_{M} \cdot \underbrace{\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{m-1} \end{bmatrix}}_{[p]}$$

- If we didn't know FFT, then this computation is just a matrix multiplied by a vector, whihc would take $O(m^2)$ time. However, FFT basically gives us a way to compute this matrix in $O(m \log m)$ time!

  Note also that this also solves this **without ever writing down M !**

- The Inverse FFT looks much cleaner in this representation:

$$M^{-1} \cdot \begin{bmatrix} p(\omega_0) \\ p(\omega_1) \\ p(\omega_2) \\ \vdots \\ p(\omega_{m-1}) \end{bmatrix} = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{m-1} \end{bmatrix}$$

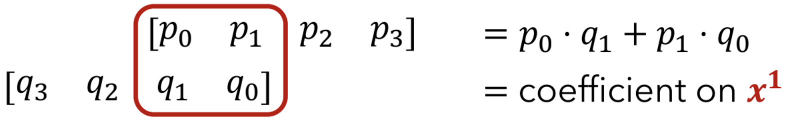
- For M specifically, we have $M_{ij} = \omega_i^j = (\omega_1^i)^j = \omega_1^{ij}$. Then, the inverse is defined as: $(M^{-1})_{ij} = \frac{1}{n} \cdot \omega_{m-i}^j$.

5.5 Applications

- **Cross Correlation:** Given the product of $p(x) = p_0 + p_1x + p_2x^2 + p_3x^3$ and $q(x) = q_0 + q_1x + q_2x^2 + q_3x^3$, the coefficient on x^i in $p(x) \cdot q(x)$ is

$$p_0q_i + p_1q_{i-1} + \cdots + p_{i-1}q_1 + p_iq_0$$

  Now consider two arrays $[p_0, p_1, p_2, p_3]$ and $[q_3, q_2, q_1, q_0]$. Now let's stack them on top of each other and take the dot product of the overlap. Depending on where they overlap, it actually corresponds to the coefficient on some x^i in $p(x) \cdot q(x)$.


$[p_0 \quad p_1 \quad p_2 \quad p_3]$
$[q_3 \quad q_2 \quad q_1 \quad q_0]$
$= p_0 \cdot q_1 + p_1 \cdot q_0$
= coefficient on x^1

  This is called **cross-correlation**, and due to FFT, we can compute these in $O(n \log n)$ time.

- **Integer Multiplication:** Say we wanted to multiply $a = a_{n-1} \cdots a_2a_1a_0$ and $b = b_{n-1} \cdots b_2b_1b_0$. We can write down these two polynomials:

$$A = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$$
$$B = b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1}$$

  and to compute the product, we can write $A(x) \cdot B(x)$ and plug in $x = 10$. Naively we would think that this takes $O(n \log n)$ time, but this is not exactly true. This is because here, our additions and multiplications don't exactly happen in $O(1)$ time anymore, so we have to be careful! In fact, the multiplication can be as large as $\Theta(n)$.

  After keeping track of all this, we get an $O(n \log n \log \log n)$ algorithm.

- **Fourier Transform:** FFT allows us to compute the Fourier transform of $p(x)$ quickly – we can decompose $p(x)$ into its consistuent sines and cosines. There are too many applications of the Fourier transform to even list:

  1. Music software
  2. Heart rate monitor
  3. Signal processing (e.g. cell phones)
  4. Many more!

  The fact that quantum computers can compute Fourier transforms exceedingly quickly is actually one of the main reasons that makes them so powerful.

6 Graphs

Formal definition of undirected graphs:

- Called $G(V, E)$.
- has a set V of vertices, and E of edges.
- Edges are denoted as a pair of vertices $\{v_i, v_j\}$, and undirected mean that each pair of $\{v_i, v_j\}$ is unique.
  - e.g. Facebook friendship graphs, where the friendship must be mutual. (Has nearly 3 billion nodes, and each node has 330 vertices)
- Directed graph is denoted in the same way, except that $\{v_i, v_j\}$ means specifically that $v_i \to v_j$.

6.1 Size of Graphs

- Generally denoted as the number of vertices (denoted by n) and the number of edges (denoted by m).
- m can be at most n^2, in the case that every vertex is connected to every other vertex (in which case we have $n(n-1)$ edges)
- Degree: the number of edges that a vertex has. With directed graphs, we can specify in-degrees and out-degrees.

6.2 Representing Graphs

- Represented either as an adjacency matrix or an adjacency list.
  - Adjacency matrix: have the vertices listed out on both row and column, put a 1 if $\{v_i, v_j\}$ are connected on our graph.
  - Adjacency list: List of length n, and each cell is a linked list that points to all the neighbours of v_i.
- There are tradeoffs for both ways:

	Adjacency Matrix	**Adjacency List**
Storage Size	$O(n^2)$	$O(n+m)$
Checking whether $(u, v) \in E$	$O(1)$	$O(\deg(u))$
Enumerate all of u's neighbours	$O(n)$	$O(\deg(u))$

- We will work with adjacency lists, since the storage size for adjacency matrices get too large too quickly.

6.3 Graph Connectedness

- We first have to solve the problem of graph traversal. Our approach will use "string and chalk," so we mark when we've reached a dead end and the "string" will allow us to backtrack.
- Algorithm description:

  ```
  def explore():
          visited[u] = true

          For all edges of $u$: if visited[v] = false then run explore(v)
  ```

- Explore guarantees that all the vertices that are visited by explore have a path from u to that vertex, and vice versa.
  - We prove the other direction: if there's a path, then that node is visited.

    Assume that this is false: assume $\exists v$ that hasn't been explored but there is a path from u to v. Instead of looking at u to v, we look at the path from u to v_k, the first node along the path from u to v that is unexplored. This means that the algorithm reached v_{k-1}, then failed to explore v.

    This is a contradiction, since explore must have been called on v_{k-1}, but failed to recurse down v_k.

6.4 Depth First Search (DFS)

Essentially calling explore, except it does it recursively on all the remaining nodes that haven't been visited yet.

- We can also use DFS to find the connected components of a graph! When we explore with DFS, we are implicitly exploring connected components, this uses the transitive fact of the `explore()` function.
- The edges that are visited by DFS are categorized into tree edges and back edges. Tree edges are edges that are used when the algorithm runs, and back edges are ones that exist within the original graph but aren't used.
- There's a third class called a *cross edges*, but they cannot exist for an undirected graph.
  - The proof is by contradiction: imagine that it did exist, then DFS would have called explore on that ancestor; but it can't possibly have since u and v do not exist in the same branch.
  - They *can* exist in a directed graph, since we only traverse through out edges (so there can be an in-edge that never gets traversed).

6.4.1 DFS Runtime

`Explore()` is called only once per node, and the runtime for each node for explore is proportional to $\deg(u)$, so the total is:

$$\sum_{u \in V} O(1 + \deg(u)) = O(n + m)$$

6.5 DFS for Directed Graphs

It's the same principle, our explore still only runs recursively on unvisited neighbours, but we need to also keep track of the amount of time at which we start and finish processing that node.

- Every time we enter a node, we stamp it with the time of the start clock. When we come out, we stamp again with the end clock. Every time we progress the clock, we increment it by 1.
- The point of the clock will be revealed later on in future lectures.

- The edges we keep track of here are forward, back and cross edges. Forward means we go down the tree from ancestor to descendant (not its immediate child), back means we go backwards (can be immediate) and cross edges are defined exactly as before.

The edges (pre, post, cross) are useful in determining properties of the graph G.

- Cross edges can only go from a later point in one branch to earlier than the other branch and not the other way around, due to the way that DFS executes depth-first.
- Suppose $(u, v) \in E$ is a tree edge. Then, we know that $\text{pre}(u) < \text{pre}(v)$ (since u is hit first), and $\text{post}(u) < \text{post}(v)$.
- Suppose that $(u, v) \in E$ is a back edge. Then, we have $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$.

7 Strongly Connected Graphs

Recall that we saw earlier that DFS basically just calls explore repeatedly, to find all the vertices reachable from a vertex u. We also introduced two clocks, where when we first visit a node we stamp it with the start clock, and that vertex will have a $\text{pre}(u)$ quantity that stores its value, then increments clock. When we leave the vertex, we stamp again with a $\text{post}(u)$.

- For cross edges (the only thing we didn't finish last time), we have $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$ (we get to v and finish exploring v before we ever touch u).
- As a recap:

Edge type	Relation
Tree edge	$\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$
Back edge	$\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$
Cross edge	$\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$

7.1 Topological Sort

- The process of finding an ordering of vertices so that no edges go backwards. This is used in software package loading, to make sure that things aren't being and that are being downloaded in the chronological order.
- Mathematically, if u comes before v in the ordering, then there is no edge (v, u).
- Two types of special nodes: **sources** and **sinks**
  - **Source:** A node that has no incoming edges and has only outgoing edges.
  - **Sink:** A node that has no outgoing edges and only has incoming edges
- Note that a node with no edge at all is considered both a source and a sink.

7.2 DAGs

- Called a directed acyclic graph, or basically just a graph without any directed cycles. If we have a cycle, then we can't toplogically sort.
- We will find that if we run DFS on a graph, it is a DAG iff it has no back edges. We prove that if we have a back edge, then it cannot be a DAG: since they go from something that's already been visited to something that's been visited earlier, then we have already found a cycle.

  Now we prove that if we don't have a DAG (i,e, has a cycle) , then it has a back edge. Since DFS visits every vertex in a graph, it will eventually enter our cycle at some v_i. Then, it will traverse through the cycle until it visits all the nodes and eventually gets to v_k, the node right before it comes back to v_i. Then, $v_k \to v_i$ will become a back edge, since it was visited earlier in the graph.

- Back edges are really special! Recall that we only have a back edge when post(u) < post(v), since the other edges have post(v) < post(u). We then conclude that if we have a DAG, then it has the property that post(v) < post(u) (since it can't have back edges).

  Does the logic work the other way around? Does this mean that if post(u) < post(v) then we have a DAG?

  Not necessarily. Just because post(v) > post(u) doesn't necessarily imply that a back edge exists.

- Our algorithm for topological sort: do a DFS on graph G, then enumerate all $v \in V$ in the decreasing order of post(v).

7.3 Strongly Connected Components

- To find DAGs on undirected graphs we can run DFS on them, but what about directed graphs?
- **Definition:** Vertices u and v are strongly connected if there is a path from u to v and there is also a path from v to u.
- A graph is *strongly connected* when all of its vertices are strongly connected.
- Generally, we partition a graph into *strongly connected components*, since it's very rare that the entire graph is strongly connected.
- Strong connectivity is an example of an equivalence relationship, since it satisfies all the properties: reflexive, symmetric and transitive.
  - Every vertex is strongly connected to itself
  - If A is strongly connected to B, then B is strongly connected to A.
  - If A is strongly connected to B and B is strongly connected to C, then A is strongly connected to C.
- If we flip all the edge (i.e. reverse the direction), the strongly connected components don't change at all!
- We can then divide this into a *meta graph*, where we group the graph by strongly connected components (try to be as general as possible when you do this).
  - The meta graph can't have cycles! Had a cycle existed, then it would imply that vertices from one strongly connected component can reach vertices of another, and vice versa!
  - Therefore, the meta graph *must* be a DAG.
- Why care about SCC? It is useful in many different fields, since SCCs naturally imply a strong equivalence of objects in a graph.

7.4 Finding SCCs

- **Attempt 1:** Consider all possible decompositions and check (BAD!)
- **Attempt 2:** Consider pairs of nodes, then run explore to see if they reach the other. (ALSO BAD!)
- There exists an algorithm that runs this in $O(n + m)$ time, where we have to run DFS (smartly) only twice!

7.4.1 More Properties of SCCs

- It actually matters where we start our DFS, since sometimes we can't exit the particular connected component. Specifically, if the node is part of a source in the meta graph, then it will visit many nodes, whereas if the node is a sink then we never exit that particular connected component.
- The "right" place to start the DFS is in the **sink of the meta graph!** The key thing is that we shouldn't exit that connected component, so running `explore()` on any vertex within this CC will give us the whole SCC.
- **Idea:** Do the topological sort on the meta graph, then run DFS on the sinks of the meta graph.
- Suppose we run DFS on a graph G. Let C and C' be two connected components such that $C \to C'$ in the meta graph. Then, $\max(\text{post}(v \in C)) > \max(\text{post}(u \in C'))$.

  Proof: Split into cases, based on where in the graph we started:

  Case 1: Suppose DFS visited C first ($u \in C$ is the first node visited by DFS. Then, DFS will explore C' first before finishing its exploration of C. This means that DFS finishes C' before finishing C, meaning that $\max(\text{post}(v \in C)) > \max(\text{post}(u \in C'))$.

  Case 2: Suppose DFS visited C' first. Then, we will never visit C since there is no directed edge between $C' \to C$. Therefore, the post of C' stops before DFS even reaches C, Hence, we have $\max(\text{post}(u \in C')) < \max(\text{post}(v \in C))$.
- Immediate corollary of this: Suppose we ran DFS on a graph G. The highest $\text{post}(v)$ belongs to a node v that is in the source SCC of the meta graph!
  - Imagine $\max(\text{post}(C))$ is not the largest $\text{post}(v)$ value. Then, this means that there exists another C'' whose $\text{post}(C'')$ is larger than that of C!
- So now we have a way of finding the source of the vertices in C, but we want to start from sinks, so we **flip the edges, then run DFS!** The only difference betweeh G and G^R (the reversed graph) is the direction of the edges; the connected components remain the same.

  Instead of flipping edges, why not run the algorithm from the minimum post value instead?

  Because the minimum isn't guaranteed to be a SCC in the same way the max is.

7.5 The Algorithm

- Compute G^R.
- Run DFS on G^R.
- Store the post numbers of this DFS in array called post-r
- Run DFS on G, but explore unvisited nodes in *decreasing* order of post-r. For every DFS we run, we find a new SCC.

8 Paths in Graphs

8.1 Single Source Shortest Path (SSSP)

- We want to compute the distances from a source $s \in V$ to other nodes in V.
- We don't use DFS here because DFS might explore much longer paths first, so it might be very inefficient.
- Solution: use **Breadth-First Search (BFS)**

  - Analogous to a bird's eye perspective, where we explore successively outward in "neighbourhoods."
  - Start at exploring from distance 1, then when everything at distance 1 is explored, continue to explore at distance 2, etc.
- The type of BFS that we use depends a lot on what kind of graph we're dealing with:
  - Unweighted graphs: Ordinary BFS works
  - Positive Weights: Dijkstra's algorithm
  - Negative Weights allowed: Bellman-Ford Algorithm
- Going down this list makes the graph more general, but they are less efficient than the ones above.

  Would a more correct statement be that BFS works if all the edges have the same weight?

8.2 Breadth First Search (BFS)

- Start at s, and add all the neighbours of s to a queue. For every vertex in the queue, we visit all the unvisited nodes from that vertex, and add it to the queue. Repeat until all nodes have been visited.

8.2.1 Runtime of BFS

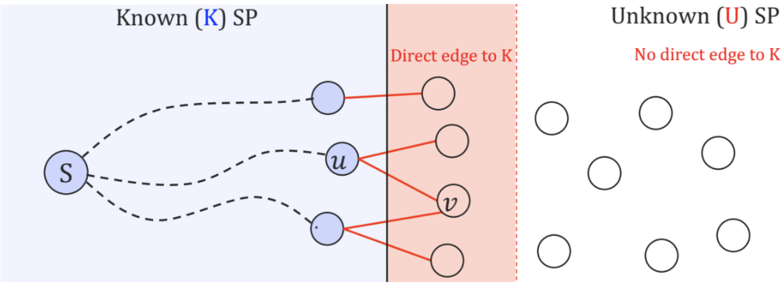
- We enqueue and deque every node exactly once if the node is connected, otherwise we don't do it at all. This takes $O(1)$ time.
- Once an item is dequeued, we need to check all the neighbours of a graph, costing $O(\deg(u))$ time.
- In total, our runtime is:
  $$\sum_{u \in V} O(1 + \deg(u)) = O(n + m)$$
  This is the same runtime as DFS, which is not a coincidence! DFS and BFS are actually related, except the queue is replaced by a stack.
- We didn't implement it as a stack in lecture, but the idea is the same.

8.3 Weighted Graphs

- BFS doesn't work here because it ignores the weights of the graph. It is possible that a graph ends up being shorter but goes through more nodes, a possibility that BFS doesn't catch.
- **Useful Fact:** Any sub path of a shortest path is also a shortest path. This is rather obvious.
- So what we should think about is that to build the shortest path, we build the shortest path from other, shotest paths but add in the shortest edge. This guarantees that our shortest path remains the shortest.

8.4 Dijkstra's Algorithm

- Let K denote the set of "known" nodes where the length of shortest path is computed. To determine node we should add to K, we should select the vertex that gives the smallest $\text{dist}(s, u) + \ell(u, v)$. Visually:



- The red region is the set of nodes that we look at.
- We don't need to recompute all distances at every iteration - instead we can just store the distances as we go along. Initial overestimates are fine, since eventually we will explore the shortest path, and its distance will eventually be updated.
- If we find a shorter path later on, we can update $\text{dist}(s, u)$ to reflect that.
- If we want to find the shortest path from S, then we can add a new variable that stores the previous node in the sequence from S to u. Therefore, when we want to find the shortest path, then we are continually looking backward until we get back to S.

8.4.1 Runtime of Dijkstra's

- The runtime of Dijkstra's depends on the kind of data structure we used to keep track of the distances:

Implementation	Insert	Delete Min	Decrease Key	Runtime
Array	$O(1)$	$O(n)$	$O(1)$	$O(n^2 + m) = O(n^2)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O((n + m) \log n)$
Fibonacci Heap	$O(1)$	$O(\log n)$	$O(1)$	$O(n \log n + m)$

- The best known runtime of Dijkstra's algorithm is $O(n \log \log n + m)$.
- At the end of the day, this is slower than DFS, by the $\log n$ term.

8.5 Negative Weights: Bellman-Ford Algorithm

- Sometimes, having negative weights is possible, for instance when traversing an edge is more beneficial to you in some way.
- Shortest paths don't really make sense if a cycle has negative length (since then we'd be infinitely descending)
- All we need to do is modify Dijkstra's update function!
  - Call an update "safe" if $\text{dist}(w)$ is an overestimate of the true shortest path between s and w. In other words, $\text{dist}(w) \geq d(s, w)$ for all $w \in V$.

see lectures for Bellman-Ford

9 Greedy Algorithms

- Are algorithms that build their solution piece by piece, and always takes the piece that offers the **most obvious and immediate benefit**.
- Some applications of where Greedy algorithms do work: Scheduling, satisfiability, Huffman Coding MSTs.

9.1 Scheduling

- Input: collection of jobs specified by their time intervals $[s_1, e_1], \dots, [s_n, e_n]$. We want to find the largest subset of jobs that have no time conflicts.
- To do this, after choosing an interval, we'd want to choose the next interval that has the *earliest end time*. Jobs that finish earlier give us more opportunities to slot in more jobs later in the day.
  - This is not achieved by selecting the shortest job, because it does not give us freedom in where s_i and e_i are.
  - This is also not achieved by selecting the earliest job, since we don't know where e_i is.
- So our code is as follows:

  ```
  While set of intervals is not empty:
  Add interval $j$ with the earliest finish time $e_j$.
  Remove any conflicted interval $i$ from the set, i.e. $[s_j, e_j] \cap [s_i, e_i] \neq \emptyset$.
  ```

- The runtime of this algorithm is $O(n)$ if the intervals are already sorted by the end time, otherwise, we'd need $O(n \log n)$ time since we'd need to sort the intervals first.
- To show that the greedy algorithm works, we need to show that this algorithm doesn't rule out an optimal solution.
- Induction is very nice to prove these algorithms, since we'd just need to prove that the algorithm selects optimally at every time step! Let's try this for our scheduler:

  Claim: For any $m \leq k$ there is an optimal schedule OPT that agrees with the Greedy solution G on the first m intervals. More formally, if the OPT can be labelled as a list of $i_1, \dots i_m$ and G has a list of $j_1, \dots, j_m$, then we require that $i_1 = j_1, \dots, i_m = j_m$.

  Base case: $m = 0$, this is fairly trivial. Any two schedules agree on 0 things.

  Inductive Hypothesis: This claim holds true for m. Now we show $m + 1$.

  Inductive Step: There are two cases we need to consider:

  - Case 1: when $i_{m+1} = j_{m+1}$, in which case we are done.
  - Case 2: $i_{m+1} \neq j_{m+1}$. Then let's define another schedule OPT' which is the same as OPT except for the fact that i_{m+1} is replaced with j_{m+1}.

    Note that j_{m+1} does not conflict with $j_1, \dots, j_m$, since the greedy algorithm does not produce time conflicts. Also, j_{m+1} does not conflict with i_{m+2} since j_{m+1} ends earlier than i_{m+1} (by the greedy algorithm). Hence, placing j_{m+1} into this algorithm instead of i_{m+1} produces an *equally valid solution* for the schedule, since the size of OPT' is the same as that of OPT. Therefore, OPT' is also optimal, completing the proof.

  Does this proof by induction assume that the Greedy solution gives a correct schedule?

- In essence, the proof is showing that there is no choice that the Greedy algorithm makes which rules out an optimal solution.

9.2 Horn Formulas

- Variables $x_1, \dots, x_n$ are either true or false.
- Clauses:
  - "Implication clause" where $(x_i \wedge x_j \wedge \dots) \implies x_k$. This is equivalent to $\overline{x}_i \vee \overline{x}_j \vee \cdots \vee x_k$.

- "Pure Negative Clauses" where $(\overline{x}_i \vee \overline{x}_j \vee \dots)$

- A Horn formula is an AND of all Horn clauses, which are either implication or pure negative.
- There is a problem called Horn-SAT which asks us to find an assignment of variables that makes all Horn formulas to be true, if an assignment exists.
- Greedy Algorithm:
  - For all i, set x_i to be false.
  - While there exists an implication $(x_i \wedge \dots \wedge x_j) \implies x_k$ being set to false, set x_k to be true.
  - If every pure negative clause $(\overline{x}_i \vee \dots \vee \overline{x}_j)$ is set to true, we return $(x_1, \dots, x_n)$.
  - Otherwise, return "not satisfiable."

9.2.1 Proof of Correctness

- We want to show that whenever the greedy algorithm sets a variable x_i to true, it does not ruin a satisfying assignment. In other words, whenever a satsifying assignment exists, then Greedy will output one.
- We can show a stronger statement: the set of variables set to True by the greedy algorithm has to be set to true in any other assignment. We prove this by induction

  Base case: In the 0th iteration of the while loop, nothing is set to true, so we're fine.

  Inductive Hypothesis: The first m variables set to true by Greedy are also true in every satisfying solution.

  Inductive Step: Let x_{m+1} be the next variable set to True by the greedy algorithm. This means that there was an unsatisfied implication $(x_i \wedge \dots \wedge x_j) \implies x_k$ where the LHS was true, and x_{m+1} is false. This only happens when $x_i, \dots x_j$ are all set to True, by the greedy algorithm on m steps (which we know to match the optimal solution by inductive hypothesis).

  Then the only way to satisfy this condition MUST have x_{m+1} set to true as well, and that completes the proof.
- Now we have to prove correctness. If Greedy outputs a solution, then it must be satisfiable – this is fairly obvious, since the while loop and if condition makes sure that all clauses are satisfied.
- We also want to

9.3 Codes

- Usually (things like ASCII) encode English characters using a fixed length of bits per character.
- If our goal is to save space, then we probably don't want that. Particularly, there are letters that appear more often than other characters, so if we were to use the same space for every character that'd be fairly wasteful.
- Assume that we have four letters with varying frequencies:

Frequency	**Letter**	**Encoding 1**	**Encoding 2**	**Encoding 3**
0.4	A	00	0	0
0.2	B	01	00	110
0.3	C	10	1	10
0.4	D	11	01	111
Total Cost:		$2N$	$N(0.4 + 0.3) + 2N(0.1 + 0.2)$ $= 1.3N$	$0.4N + 2N(0.3) + 3N(0.2 + 0.4)$ $= 1.9N$

- There are issues with encoding 2: it's lossy in the sense that AB is encoded in the same way that BA is coded.

- These issues are solved in encoding 3, and we found that we can still do better than the $2N$ from our naive application where every letter gets the same number of bits.

10 Huffman Coding, MSTs

Recap of Greedy algorithms:

- Our goal is to prove that whenever a choice is made, that an optimal solution still exists, proven to exist via induction.
- Base case: at the beginning, achieving optimal choice is always possible
- Inductive Hypothesis is the same as normal induction, and inductive step is to show that if an alternate choice is made it doesn't violate the optimal solution.

10.1 Prefix codes and Trees

- Prefix codes can be represented as a binary tree with k leaves.
- the code is the "address" of a letter in the tree (i.e. the string of numbers leading from the root to that leaf). We want to order the tree from highest to lowest frequency, so that the letters with the highest frequency uses less characters.
- In general, the cost for such a tree is:

$$\text{cost} = \sum_{i=1}^{n} f_i \cdot \text{depth}(\text{leaf } i)$$

- Our goal is to find an *optimal subtree*. What does such a tree look like?

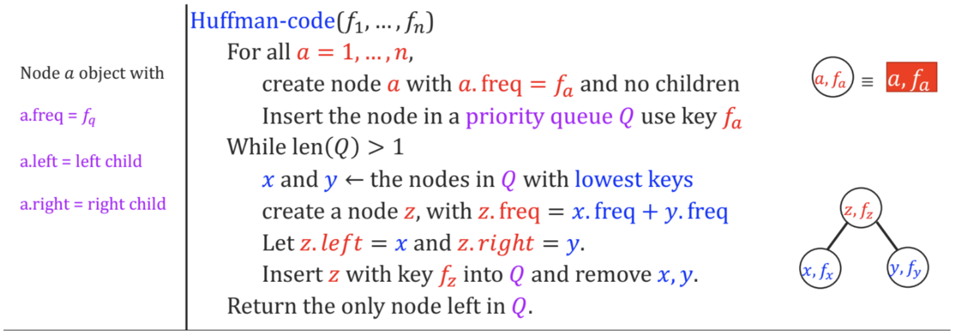
  Answer: Even if we don't know what the frequencies are, the optimal code should be a **full binary tree.**

- Now we need to prove that there exists an optimal tree when the two lowest frequency symbols are siblings of each other.

  Proof: By contradiction, let x, y be symbols with lowest frequencies and assume they aren't siblings. Let a, b be the deepest pair of siblings. Since x, y aren't siblings of each other, then only one of a, b are one of x or y. WLOG, let $x = a$. What happens if we swap x, y and a, b? Well, we know that $f_a, f_b \geq f_x, f_y$, and we've reduced the length of a, b while also reduced frequency of the deepest entries in the tree, meaning that we've ended up with a cheaper tree! Hence, the original tree could not have been an optimal tree.

10.2 Algorithm (Huffman Coding)

See below for pseudocode:



Node a object with

a.freq = f_a

a.left = left child

a.right = right child

```
Huffman-code($f_1, \dots, f_n$)
    For all $a = 1, \dots, n$,
        create node $a$ with $a.\text{freq} = f_a$ and no children
        Insert the node in a priority queue $Q$ use key $f_a$
    While len($Q$) > 1
        $x$ and $y$ ← the nodes in $Q$ with lowest keys
        create a node $z$, with $z.\text{freq} = x.\text{freq} + y.\text{freq}$
        Let $z.left = x$ and $z.right = y$.
        Insert $z$ with key $f_z$ into $Q$ and remove $x, y$.
    Return the only node left in $Q$.
```

- The idea is to recursively generate a tree using the lowest frequencies, and combining them together by adding the frequencies of each tree's children.
- **Runtime Analysis:** Storing in our priority queue can be optimized by using a binary heap, which takes $O(n \log n)$ time. Combination also takes $O(n \log n)$ time, so total runtime is $O(n \log n)$.
  - Inserting into priority queue takes $O(n \log n)$ time.
  - At every step of the while loop, we perform 2 deleteMin instructions, which is constant time.
  - There is 1 insert each time (after the connection). This means that on every iteration, we are halving the number of nodes (hence $O(n \log n)$).
  - So total time complexity is $2O(n \log n) = O(n \log n)$.
- This generates a full binary tree with optimal coding.

  Proof: We show that a greedy selection (which is what Huffman Coding is doing) does not rule out an optimality coding.

  Base case: $n = 2$. We can generate optimal code using 0 for first letter and 1 for second letter. Huffman coding does the same.

  Inductive Hypothesis: Assume that this works for $n - 1$ letters.

  Inductive Step: Let T be the optimal tree for the frequencies $f_1, \dots, f_n$. WLOG, let $f_1 \leq \dots \leq f_n$. Assume that the two lowest frequency codes are siblings (proven from earlier), and merge the two into a single node, where $f = f_1 + f_2$. Looking at the cost of the new tree T', we know that

  $$\text{cost}(T) = \text{cost}(T') + f_1 + f_2$$

  Huffman coding also does this merging process, and our inductive hypothesis guarantees that this tree is optimal on $n - 1$ letters, so when we split back into the two characters it is still guaranteed to be optimal. Formally, if H' is the cost of the reduced tree, then

  $$\text{cost}(H) = \text{cost}(H') + f_1 + f_2$$

  which is the same as the cost relationship with T, so Huffman coding does indeed give an optimal coding.

What if $f_1 + f_2$ happens to be large enough such that $f_1 + f_2 > f_n$?

Apparently it doesn't matter?

10.3 Minimum Spanning Trees (MSTs)

- Tree also has edges, so we can assign a cost as well: $\text{cost}(T) = \sum_{e \in T} w_e$ (the sum of the weights).
- Suppose we're given a graph $G(V, E)$ with non-negative weights. We want to find a set of edges that connects the graph, and has the smallest cost.
- Why do we care? This gives the notion of connectivity in a network, so you can think of cell towers or roads/railways as practical applications.
- We will use the same approach: first we ask about what an MST looks like.
  - It will be an acyclic graph, since removing an edge that's part of a cycle still preserves the connectivity in the graph.

10.3.1 Graph Structures and Cuts

- **Cuts:** a way to partition a graph that splits up the vertices into two groups.
- They're important because cuts go through edges to divide vertices into groups.
- Imagine we've already discovered some edges X of the MST. Consider the cut that doesn't cut any edges X. Now we look at the edges that are being cut. The edge from a larger MST is being cut, and it's the lowest weight edge that we should add to our MST!

  This is a very important property, it shows that *any* cut that we make is an edge that *can* be added to our MST, so therefore regardless of which vertex we start searching at, an MST is still guaranteed.
- Therefore, we should add this edge and its corresponding vertex to our MST.
- We can formalize this argument via a proof, but I'm too lazy to write it here.
- Turns out that any algorithm that fits the following properties forms an MST:
  - Start with X, an empty list.
  - Pick $S \subseteq V$ such that X has no edges from S to $V \setminus S$
  - Choose the lighest weight edge from S to $V \setminus S$.
  - Add edge to X.
- The proof for why this does give an MST can be done via induction.

10.3.2 Kruskal's Algorithm

- Instead of doing S and $V \setminus S$, it instead selects edges, and checks whether the edge forms a cycle. If it does, we don't add this edge. This process of checking a cycle actually does split our graph into S and $V \setminus S$, albeit implicitly.
- We show correctness by showing that Kruskal's algorithm fits the meta algorithm given above.

11 MST Continued, Set Cover

11.1 Prim's Algorithm

- The idea is to draw a tree by greedily adding the cheapest edge that can grow the tree.
- Start from some vertex, and repeatedly pick the lightest edge (u, v) such that $u \in S$ and $v \in V \setminus S$.

  How exactly is this different from Kruskal's algorithm? Aren't both using cuts in the same way?

  Kruskal's doesn't start from a given vertex, but instead just selects edges. Prim's starts with vertices and looks at edges that connect from S to $V \setminus S$
- Remember: the shape of the MST is dependent on the node that we start at, but an MST will always exist no matter which vertex we start at.
- Both Prim's and Kruskal's algorithm works on negative edge weights. This is because the cut property still holds, and the notion *minimum* spanning tree is not broken with negative edge weights.

11.2 Implementation

- The naive implementation of Prim's is actually quite slow, since on every added vertex we are looking for new cuts and checking edges every time.
- We can optimize by using priority queues (basically a max heap based on priority).

  Is this true about priority queues?
- So here are the things we need to keep track of:
  - For every edge $v \in V \setminus S$, check whether v has a direct edge of the set S of "visited" vertices, and also the cost of the lightest edge connecting v to the set S of visited vertices.
  - We had the same dilemma before, with Dijkstra's algorithm!
- So let's follow the same procedure as Dijkstra's!
  - First start with dist(v) set to infinity, and prev(v) to null for every vertex.
  - If a neighbor u is added to S (visited set) and dist(v) $> w_{u,v}$, then we update dist(v) $= w_{(u,v)}$, and set prev(v) $= u$.
  - This is slightly different from Dijkstra's, where the dist array instead marks the minimum edge between two visited nodes, instead of the total distance from a certain vertex.
  - Part of the reason for this is that MSTs don't care about where you start.
- The "cut" in this case is actually the process of adding adjacent edges from visited nodes to unvisited ones into the priority queue.

11.3 Runtime

- For a priority queue: we can either use a binary heap ($O(\log n)$ for each operation) or fibonacci heap (a little bit better, since $\log(n)$ for inserts but $O(1)$ for everything else.
- So because of the constant time for Fibonacci heap, it has $O(m + n \log n)$ time, whereas a binary heap has $O((m + n) \log n)$.
- Comparing both algorithms:
  - Kruskal's: $O((m + n) \log n)$
  - Prim's (with Fibonacci heap) $O(m + n \log n)$.
  - For sparse graphs (so ones with not many edges), both are equally as good.
  - For dense graphs, Prim's is much better.

11.4 Set Cover Problem

- Input: the universe of n elements $U = \{1, \dots, n\}$, and subsets $S_1, S_2, \dots, S_m \subseteq U$, such that $\bigcup_{i=1}^{m} S_i = U$.
- Output: A collection of S_i of minimal size.
- This is an example of a problem where the greedy algorithm is *not optimal!* Instead, it is approximately optimal.
- Claim: if the optimal solution uses k sets, then the greedy algorithm uses at most $k \ln n$ sets.
- We will prove this recursively: let n_t be the number of elements not covered by the greedy algorithm after t choices. Then, we can reframe the problem to be that when $t = k \ln n$, we want $n_t < 1$.

- Subclaim 1: $n_1 \leq n_0 - \frac{n_0}{k}$.

  Proof: the optimal solution requires k sets to cover U, so the average number of elements in any set is $\frac{n}{k}$. Hence, there is a set that counts more than $\frac{n}{k}$ elements (if not equal).

- Subclaim 2: for any t, $n_{t+1} \leq n_t(1 - 1/k)$.

  Proof: This is a natural extension of claim 1.

- With this proven, we introduce an **approximation factor**, which is a way to say that Greedy is optimal, with an approximation factor of $\ln n$.

12 Dynamic Programming I

12.1 Fibonacci Numbers, revisited

- Imagine computing Fibonacci numbers; there's a lot of repeated calculations! For instance, $F(1)$ is computed 2^n times when we're looking for $F(n)$!
- To optimize this, store each successive computation of $F(n)$ into an array that we access, so that we only need to compute each $F(k)$ exactly once.
- This is called **memoization**, where we store things in a "memo," to be accessed by our algorithm later on.

12.2 Elements of Dynamic Programming

- There are a couple hallmarks of DP:
  1. Subproblems, or "optimal substructure". Refers to the fact that large problems can be broken up into smaller subproblems. For Fibonacci, this means that $F(n)$ is recursively expressed in terms of smaller subproblems.
  2. Overlapping subproblems: A lot of subproblems overlap with one another. We recurse to smaller subproblems, and in doing so we see that a lot of computation is repeated. The solution to this is to use memoization, so that each computation is done only once.
  3. There are two ways to do DP:
     - Top-Down: start from the largest subproblem and recurse to smaller subproblems. This often involves recursion.
     - Bottom-up: start from the smallest subproblems then work to larger subproblems. Memoization still happens; we just fill the table from the small to largest problems. In this method, this doesn't need a recursive call.
- The mathematical runtime of top-down and bottom-up are the same.
- The computation structure for DP actually looks awfully similar to a DAG.
- If we view every subproblem as a node in the graph: construct it in such a way that an edge $i \to j$ exists if the solution to subproblem j directly depends on the solution to of subproblem i.
- Consider a topological sort on this DAG: then the bottom-up solution directly follows the conputation of this DAG!
  - In the top-down framework, we are filling up the memo table in topological sort order, since that table is still being filled from bottom up.

12.3 Shortest Paths on DAGs

- We're given a DAG with a source s. We want to find the cost of the shortest path from s to u for all $u \in V$. We also want to do this in linear time, $O(n + m)$.
- We can always run a topological sort on this DAG in $O(n + m)$ time. Our subproblems are the distances from s to u for every node u.
- After ordering in topological sort, we can just go down this graph *in topological order!* This means that the structure of the DP tree is the same as that of the topological sort.
- In terms of our recurrence relation, $\text{dist}(u) = \text{dist}(v) + \ell(u, v)$. Here, $\text{dist}(v)$ is implied to be memoized, since it's already a solved problem.
- This is an $O(n + m)$ solution to this problem!

12.4 DP Recipe

a) Identify the subproblems (i.e. find the optimal substructure)

b) Find a recursive formulation for the subproblems: just try to solve it via recursion and see where it gets you.

c) Design the DP algorithm – fill in a table, starting with the smallest sub-problems and building up.

12.5 Shortest Paths with k

- Here we consider the same problem of finding shortest path, but we're restricted to use at most k edges.

  Fill this out from lecture recordings

12.6 All-Pair Shortest Paths

- Here, instead of finding the shortest path from a singular source node, we want to find it for all pairs of nodes.
- Input: again a graph with no negative cycles.
- Naively, we can run Bellman-Ford on all nodes, but this would take $O(nm)$ a total of n^2 times, so our total runtime could be as large as $O(n^4)$ for dense graphs. Therefore, we're looking for a better algorithm.
- Identify the subproblem: subproblem k: for all pairs, find the shortest $u \to v$ path whose internal vertices (so the path they take) only use nodes $\{1, 2, \dots, k\}$.
  - In other words, there's a collection of k nodes, and the path from $u \to v$ *only* uses these nodes.
- Recursion: When we have the set from $\{1, \dots, k\}$, we want to find the relation between this set and how to expand this set. There are a couple ways that the new node can be added:
  - Case 1: The new node added does not lie on the path: then nothing really changes, so $\text{dist}_{k+1}(u, v) = \text{dist}_k(u, v)$.
  - Case 2: The shortest path uses the added node: this path can be broken into two parts: the shortest $u \to (k + 1)$ path and then the shortest $(k + 1) \to v$ path. Both of these paths are already computed (by definition of them only using the set $\{1, \dots, k\}$), so we just have to add these two up.
  - To combine these two, we find the minimum of these two to find whether the path from u to v has changed or not.
- Runtime: Each update is $O(1)$ time, and we have to loop over u, v a total of k times, so overall $O(n^3)$ runtime.
- This is called the **Floyd-Washall Algorithm.**

13 Dynamic Programming II

We will look more at how to choose subproblems (step 1 of our "recipe" to solve DP problems) Some problems we'll look at today:

- Longest increasing subsequence
- Edit distance
- Knapsack Problem

Pay attention to the information our subproblems need to be storing.

13.1 Longest Increasing Subsequence

- Given an array of integers $[a_1, \dots, a_n]$, and we want to return the length of the longest increasing subsequence of the input. (the selection of the indices **doesn't have to be contiguous**)
- We're going to deal with 1-indexed arrays here.
- Why is this useful? This problem is one processing step used in a lot of other algorithms, and even the game of Solitaire (also called Patience Sorting)

13.1.1 Subproblems

- Which of the following is better?
  - $L[j]$ is the length of the LIS in the array $\left[a_1, \dots, a_j\right]$ for $j = 1, \dots, n$.
  - $L[j]$ is the length of the LIS in array $\left[a_1, \dots, a_j\right]$ that ends in a_j for $j = 1, \dots, n$.

    The second one is far better, because we keep track of a_j information.
- The second subproblem is better, because keeping track of a_j is very valuable when we are trying to recurse back in our DP problem.

  If we don't keep track of a_j, we don't have any information of the last element of our LIS subproblem, so we don't know how to attach it.
- **Whatever the subproblem is not storing/not stating is going to be taken away from you. You can only observe things that the subproblem is storing/stating**
- To add, there are two cases:
  - Suppose $a[j] \leq a[i]$. Then we cannot add a_j because it's not part of the longest increasing subsequence. Therefore, $L[j]$ (the LIS up to j) is the same as $L[i]$.
  - Otherwise, $a[j] > a[i]$, so we can add it to the LIS (so $L[j] = L[i] + 1$).

    How is it possible that $a[j] \leq a[i]$?

    There could be an increasing sequence that grows slower that isn't counted in the i-th iteration
  - Therefore recursively, we have to apply the following:

$$L[i] = \max_{i<j}\{L[i] \,:\, a_j > a_i\} + 1$$
$$L[1] = 1$$

    We want the maximum because we want the *lonegest subsequence* to tack a_j onto.
- In total, there are $O(n)$ subproblems, and each subprolbem has $O(n)$ time, so in total we have an $O(n^2)$ runtime.

13.2 Edit Distance

- Given a string $S[1, \ldots, m]$ and $T[1, \ldots, n]$, we want to find the smallest number of edits to get us from S to T.
- Allowed operations: insert a character, delete character, change character.
- Why is this useful? Autocorrect, autocomplete in search engines and also DNA analysis of similarities.

13.2.1 Cost of Alignment

- Rather than thinking about distance in terms of *moves*, instead we can think about the cost of alignment. In other words, we look at the number of columns that don't agree.

```
S-NOWY
SUNN-Y
```

Alignment of cost 3

```
SN-OWY
SUNN-Y
```

Alignment of cost 4

13.2.2 Subproblems

- We define a 2D array to keep track of subproblems, where

$$E(i, j) = \text{EditDist}(S[1, \ldots, i], T[1, \ldots, j])$$

  So this defines the cost of optimal alignment for strings $S[1, \ldots, i]$ into $T\ [1, \ldots j]$.

- What are the different ways we can align the subproblems?
  - $S[i]$ is dangling, $T[j]$ is dangling, $S[i]$ and $T[j]$ are both fully aligned. Visually:

Case 1

$S[1 \cdots i-1]$	$S[i]$
$T[1 \cdots j]$	$-$

Case 2

$S[1 \cdots i]$	$-$
$T[1 \cdots j-1]$	$T[j]$

Case 3

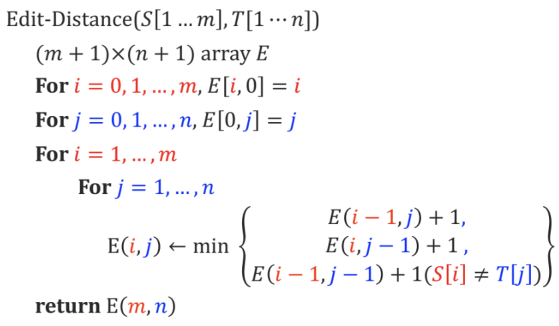
$S[1 \cdots i-1]$	$S[i]$
$T[1 \cdots j-1]$	$T[j]$

  We add in one at a time, so there can only be one dangling letter (or none) at a time!

- We recurse based on the cases:
  - Case 1: $S(i, j) = S(i-1, j) + 1$.
  - Case 2: $S(i, j) = S(i, j-1) + 1$.
  - Case 3: $S(i, j) = S(i-1, j-1) + 1(S[i] \neq T[j])$.
  - Base cases: $S(i, 0) = i$, $S(0, j) = j$.

    The 1 represents an indicator function that counts the number of misaligned characters.

- During the recursion step, we want to fill $S(i, j)$ with the *minimum* value of these three, since we want to minimize the number of edits. We will store this information (memoizing) in a 2D array.
- We have to traverse our array either row by row, column by column, or diagonally. This is because we need to ensure that all three subproblems that we're considering have already been computed.
- Pseudocode:



```
Edit-Distance($S[1 \dots m], T[1 \cdots n]$)
    $(m+1)\times(n+1)$ array $E$
    For $i = 0, 1, \dots, m$, $E[i, 0] = i$
    For $j = 0, 1, \dots, n$, $E[0, j] = j$
    For $i = 1, \dots, m$
        For $j = 1, \dots, n$
            $E(i,j) \leftarrow \min \left\{ \begin{matrix} E(i-1,j)+1, \\ E(i,j-1)+1, \\ E(i-1,j-1)+1(S[i] \neq T[j]) \end{matrix} \right\}$
    return $E(m, n)$
```

- Runtime: there are $O(mn)$ subproblems, since we have a 2D array of dimension mn. At each subproblem, we are only computing a minimum, which is $O(1)$ runtime, so we just have $O(mn)$ runtime.

  Isn't the indicator also computed at this step? Why is that not accounted in the runtime?

  The indicator is run in constant time, since we're only looking at the i-th character in comparison to the j-th character.

13.3 Knapsack (with repetition)

- A weight capacity W and n items with weights and values (w_i, v_i). We want to output the most valuable collection of items whose total weight is at most W.
- We will be selecting with repetition here, but there is an easier variation where we don't consider repetition.

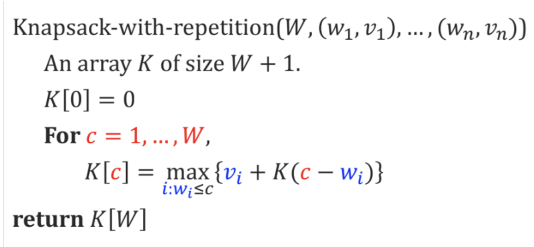
13.3.1 Subproblems

- For all $c \leq W$, we want to consider the best achievable arrangement for knapsack of capacity c.
- Recurrence: For a given item i, then once we put it in the knapsack there are only $c - w_i$ capacity that remains to be optimized. This is our recurrence relation.

$$K(c) = \max_{i:w_i \leq c}\{v_i + K(c - w_i)\}$$

  This is a maximum over the value because we want to maximize the value being put in our knapsack.

- We will store this information in an array of size $W + 1$, since we need a $K(0)$ element.



```
Knapsack-with-repetition($W, (w_1, v_1), \dots, (w_n, v_n)$)
    An array $K$ of size $W + 1$.
    $K[0] = 0$
    For $c = 1, \dots, W$,
        $K[c] = \max_{i:w_i \leq c}\{v_i + K(c - w_i)\}$
return $K[W]$
```

- Runtime: There are $O(W)$ subproblems, and at each subproblem, we have maximally n items we need to check. So in total, we have $O(nW)$ runtime.
- Generally we want to think of the runtime in terms of the input. For graph problems, we looked at the input size $|V|$ and $|E|$. But for this problem, W takes $\log(W)$ bits to represent W. So the input size is $\log(W)$. For the weights of the items themselves, they also only have at most $\log(W)$ bits, Therefore, the total runtime is $O(n \log W)$.

This kind of algorithm is polynomial in n, but not W. This is called a pseudo- polynomial algorithm, since it's an algorithm that's polynomial given the numerical value of the input but not in the input size.

14 Dynamic Programming III

We'll look at more examples today of DP.

14.1 Knapsack (without repetition)

- Start with a recap with knapsack: had a weight capacity W, and a set of items with individual weights (w_i, v_i), and we wanted to look at the most valuable combination of items.
- Now, we're going to look at this problem with the with the constraint that *we cannot choose with repetition*
- To solve, look at how we solved the problem with repetition: introduced $K(c)$ which gets us the best achieveable value for a capacity $c \le W$. The issue with trying the same thing is that our subproblems don't track which items have already been used. Why not keep track of both?

14.1.1 Subproblems

- Introduce a 2D array: essentially solve the problem for smaller knapsacks and also smaller capacities. Then expand in two directions: in terms of the number of items and also the capacity.
- So keep track of all weights $c \le W$ and all items $j \le n$. Define $K(j, c)$ to be the optimal solution to the knapsack for capacity c and items $\{1, 2, \dots, j\}$. (It doesn't need to use all the items from 1 to j.
- For each $K(j, c)$, we recurse smaller subproblems:

  Case 1: The optimal solution on items 1 through j doesn't use item j. Here, $K(j, c) = K(j - 1, c)$.

  Note that this is not equivalent to $K(j - 1, c - w_j)$, since the w_j could be distributed among other items.

  Case 2: the optimal solution on items 1 through j uses item j. Here, $K(j, c) = K(j - 1, c - w_j) + v_j$. We add v_j to K since we're now using item j.

  The intuition here is that we use the optimal solution without item j, then add in item j at the end.

14.1.2 Implementation

- So let's formalize this:

  $$K(j, c) = \max\{K(j - 1, c), v_j + K(j - 1, c - w_j)\}.$$

  with base cases $K(0, c) = 0$ and $K(j, 0) = 0$. The base cases make sense since with no items our optimal value is 0, and with no allowed weights then the optimal value is also 0.
- Looking at $K(j, c)$ it only relies on the subproblmes $K(j - 1, c)$, or $K(j - 1, c - w_j)$, so we're only looking at row $j - 1$, and different elements in that row. This tells us about the order in which we should be solving the subproblems: we could either do this row by row or column by column.
- For runtime, ther eare $O(nW)$ subproblems, and in each subproblem we're doing constant work (memory access), so therefore the total runtime is $O(nW)$, just like knapsack with repetition.
- For space complexity, notice that each K only depends on the previous row, so once we've moved onto the 3rd row, we no longer need the first. We can delete this from memory, so the optimized space complexity is $O(W)$.

14.2 Traveling Salesperson Problem

- A notoriously difficult problem, and DP helps us get a *slightly* better runtime.
- Input: Cities $1, \dots, n$ and pairwise distances d_{ij} between cities i and j. We want to find a "tour" of minimum total distance (so we need to visit every city exactly once and return to the city we started at).
- The naive brute-force algorithm basically is the one where we have to go through all possible tours: there are $n! \in O(n^n)$ possible tours, which makes this computation very expensive.
- Dynamic programming gives us $O(n^2 2^n)$. (this is nearly optimal, beating $O(n2^n)$ is theorized to be impossible)
  - To give an illustration of the difference DP makes, if $n = 25$, then $O(n!) \approx 10^{25}$, whereas $O(n^2 2^n) \approx 10^{10}$, so we're already better by 15 orders of magnitude.

14.2.1 Subproblems

- One challenge of TSP is that subproblems aren't exactly solving the problem. If we just look at TSP for a subset of our graph, that doesn't necessarily give us a solution to the larger problem, since we're looking for cycles. Instead, we think of "partial solutions" to our graph.
- We think of the subproblmes as starting from city 1, ends in city j, and passes thorugh all cities in a set S (which includes city 1 and j). Visually:

  $$1 \to i_1 \to i_2 \to \cdots \to j$$

  So we want to formally define $T(S, j)$ to be the length of the shortest path visiting all cities in S exactly once, starting from 1 and ending at j.

14.2.2 Recurrence Relation

- How to compute $T(S, j)$ using smaller subproblems? Well, look at the string again:

  $$\overbrace{\underbrace{1 \to i_1 \to i_2 \to \cdots \to i}_{T(S \setminus j,i)} \to j}^{T(S,j)}$$

  To actually talk about $T(S, j)$, then we need to add d_{ij} onto every $T(S \setminus j, i)$. However, what is annoying is that we actually don't know which city is second to last, so we'll need to consider every possible city $S \setminus j$.
- So, we'll have to pick the minimum over all $i \in S$ such that $i \neq j$. Formally:

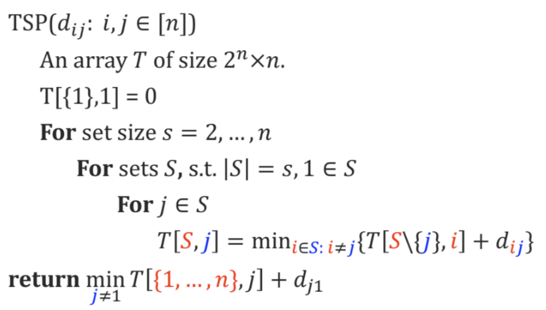
  $$T(S, j) = \min\{T(S \setminus j, i) + d_{ij} | i \in S \wedge i \neq j\}$$

- Our base cases are $T(\{1\}, 1) = 0$, this is fairly trivial. We also want that $T(S, 1) = \infty$. The reason we want this is because we're talking about incomplete paths, so $T(S, 1)$ is not a valid non-cycle. Hence, we want to set it to ∞.
- We're not done though, because we have to do something to get us back to a cycle! At the end of the recursion step, we'll want to add the final edge $(j, 1)$ back, but adding only the minimum:

  $$T(S, 1) = \min_{j \neq 1}\{T(\{1, \dots, n\}, j) + d_{j1}\}$$

14.2.3 Implementation

- Want an array of size $2^n \times n$, and start with base cases. Then work on the recursion:



```
TSP($d_{ij}$: $i,j \in [n]$)
    An array $T$ of size $2^n \times n$.
    T[{1},1] = 0
    For set size $s = 2, \dots, n$
        For sets $S$, s.t. $|S| = s, 1 \in S$
            For $j \in S$
                $T[S,j] = \min_{i\in S:\, i\neq j}\{T[S\setminus\{j\}, i] + d_{ij}\}$
return $\min_{j\neq 1} T[\{1, \dots, n\}, j] + d_{j1}$
```

- For runtime, there are $O(2^n \times n)$ subproblems, and on each layer we're doing $O(n)$ work, since we're checking the minimum across n nodes every iteration. So, we have $O(n^2 2^n)$ as the final runtime.

  How do we explain that $n^2 2^n$ is the number of subproblems?

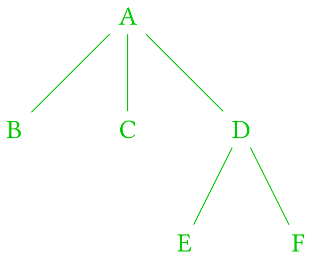
  $O(n)$ work at every step, then $n \times 2^n$ subproblems. There are 2^n subsets, and in each subset we can choose a j to exclude, which we can upper bound by saying that there are n of these. So $n \times 2^n$ is a tight upper bound on the number of subproblems.

14.3 Independent Sets in Trees

- We're given an undirected graph $G = (V, E)$, and want to output the largest independent set of G.
- Recall that a set $S \subseteq V$ is considered independent if there are no edges between $u, v \in S$.
- This is also a notoriously hard problem, for general graphs. There isn't a polynomial time algorithm that does this. But for trees, we're in luck!

  Why isn't the solution just selecting every other layer?

  There are instances where we can pick from two consecutive layers and still not have an edge. Consider the tree:

  Our greedy algorithm would select either $\{A, E, F\}$ or $\{B, C, D\}$, but the optimal set is actually $\{B, C, E, F\}$, so this proves that our algorithm isn't optimal.

- For trees, we know that they don't have cycles, so we can pick any node and say that that is the root. By doing this, we can get a "natural ordering" of the subproblems.

14.3.1 Subproblems

- Let $I(v)$ be the size of the maximum independent set in the subtree that is rooted at v.
- Why is this a good subproblem? Becuase it's easy to write a recursion relation for it!
- For the subproblems, there are two cases:

Case 1: v (the root of the tree) is part of the optimal independent set. This means that the children aren't allowed to be part of the independent set. So if we take v, we can't take any of the subproblems. So we need to look instead at the *grandchildren* of v to join. Here, we'd write this as:

$$I(v) = 1 + \sum_{u \in \text{grandchildren}} I(u)$$

We add 1 here because we're including v now.

Case 2: v is not part of the optimal independent set. Here, we would just take the maximum of the children. Then:

$$I(v) = \max_{u \in \text{children}} \{I(u)\}$$

So we'll take the max of these two cases:

$$I(v) = \max\{1 + \sum_{u \in \text{grandchildren}} I(u), \sum_{u \in \text{children}} I(u)\}$$

Also, base cases is that $I(\text{leaf}) = 1$.

14.3.2 Implementation

- We need a data structure to store the tree easily, and also make sure that every child is processed before the parents are. Well, we can iterate through the graph in post decreasing post order!
- The runtime of DFS on trees is $O(|V|)$, and each edge is looked at ≤ 2 times – once for the children and also once for its grandchildren, so each subproblme takes constant time.
- So that the total work is $O(|E|) = O(|V|)$, since $|E| = |V| - 1$.

15 Linear Programming

- Just like dynamic programming, there are different paradigms of programming, so for the following lectures we'll explore linear programming.

15.1 Example: Making Cake

- Suppose we're trying to make cake, and each item has a cost and associated profit. How much of each item should we produce in order to maximize profit?
- We're also given constraints, in terms of our supply of raw ingredients.
- This falls under the category of *constrained optimization.* They essentially ask us to maximize a quantity, while satisfying certain constraints. We've actually done this before!
  - When finding MSTs, the constraint was that our algorithm should have outputted some spanning tree.
  - For longest increasing subsequence, the constraint was that it was a subsequence and it is an increasing one.
- The difference between these and linear programming is that the objective function (our goal) is a minimalization (or maximization) of a linear function of the decision variables.
- Our goal is to find an algorithm that can solve all linear programs efficiently.

15.2 Decision Variables

- With decision variables we generally say that they are real values, and we **cannot** have integer constraints in linear programs. If they do exist, then it's now called an Integer Linear Programs (ILP), which has a completely different solution method.
- We now relax the constraints of x and y and allow them to take any value within $\mathbb{R}$

15.3 LP Standard Form

- A linear program in *standard form* can be written as follows:

  We want to maximize the equation: $c_1x_1 + c_2x_2 + \cdots + c_nx_n$, subject to the constraints:

  $$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \end{aligned}$$

  and also $x_1, x_2, \ldots x_n \geq 0$. In general, we say that we have m constraints and n variables to satisfy. We can also write this in matrix form, where we want to maximize $\mathbf{c}^\top x$, subject to the equations $A\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq 0$. Here, $\mathbf{c}$ and $\mathbf{c}, \mathbf{x}$ are $1 \times n$ column vectors and b is a $1 \times m$ column vector.

  Why aren't we taking $c^\top x^\top$?

15.3.1 Classroom Allocation

- We have a set of courses and possible classrooms, and every course needs a classroom, but not every course fits in every classroom! We want to find the maximum number of courses allocated to a classroom.
- We can construct a graph $G = (V, E)$ where course c can be assigned to classroom r if and only if $(c, r) \in E$.
- We can make our decision variable $x_{c,r}$ (ideally, boolean of 0 or 1, but this condition must be relaxed) for each pair $(c, r) \in E$.
- Now for the constraints:
  - Instead of constraining $x_{c,r}$ to 0 or 1, we can constrain $x_{c,r} \in [0, 1]$. As we'll see, even though we've assumed $x_{c,r}$ can be real, they will turn out to be integers.
  - We want the room to not be simultaneously assigned to more than one course (fully) at a time. So, for all rooms r,

    $$\sum_{c\,:\,(c,r)\in E} x_{c,r} \leq 1.$$

    In other words, for every room, the number of courses assigned to it must be less than 1.
  - We want every class to exist in one classroom only (so we don't have a situation where half the class is in Wheeler and the rest is in Pimentel) , so for all classes c,

    $$\sum_{r\,:\,(c,r)\in E} x_{c,r} \leq 1.$$

    Why isn't the constraint of $x_{c,r} \in [0, 1]$ sufficient to guarantee this condition?

    Because there's multiple $x_{c,r}$ being assigned to *every* pair! We want the sum of all of these to be less than 1, so that part of the class isn't somewhere else.

- Our objective is to find:

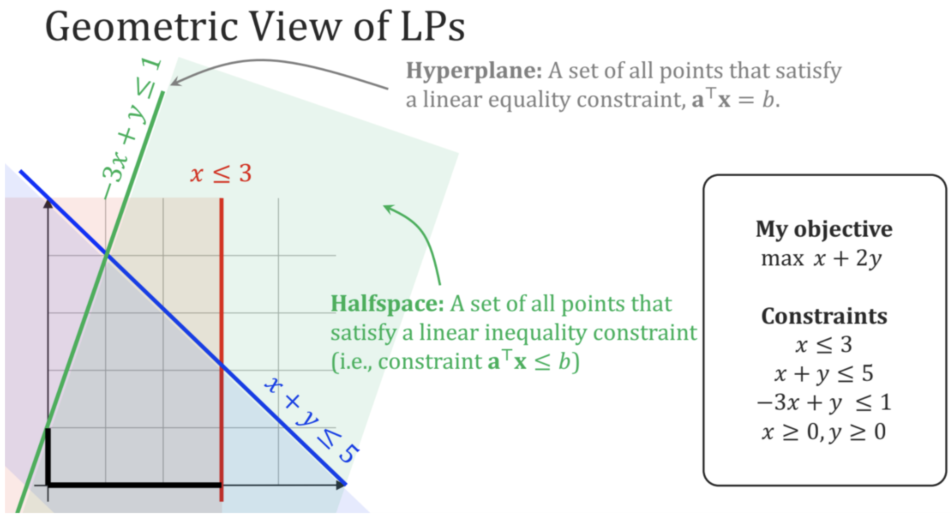
$$\max \sum_{(c,r)\in E} x_{c,r}.$$

15.4 Geometric Intuition

- Suppose we have the following LP when we're maximizing $x + 2y$ with the following constraints:

$$\begin{aligned} x &\le 3 \\ x + y &\le 5 \\ -3x + y &\le 1 \\ x &\ge 0 y \ge 0 \end{aligned}$$

We can graphically draw all of these constraints out, which will look as follows:



Geometric View of LPs

Hyperplane: A set of all points that satisfy a linear equality constraint, $\mathbf{a}^\top \mathbf{x} = b$.

Halfspace: A set of all points that satisfy a linear inequality constraint (i.e., constraint $\mathbf{a}^\top \mathbf{x} \le b$)

My objective
max $x + 2y$

Constraints
$x \le 3$
$x + y \le 5$
$-3x + y \le 1$
$x \ge 0, y \ge 0$

The overlap of these three regions defines the set where all our constraints are satisfied. We will call this region the **feasible region**. This region is always going to be a **convex region.**

- *Definition:* A set S is *convex* if for any two points $x, y \in S$, all points $z = \alpha x + (1 - \alpha)y$ for $\alpha \in [0, 1]$ satisfies $z \in S$.

  In other words, all points on the line connecting x and y are also in S. We can prove (proof in slides) that every linear program will have a feasible set that is convex.

- Suppose now we're trying to maximize the equation $x + 2y$. To find the optimal solution, we have to find *level sets* of the feasible region, by considering $x + 2y = c$ and incrementing c.

- *Definition:* A vertex (or extreme point) is a point found at the intersection of hyperplanes in $\mathbb{R}$.

- *Claim:* Any linear program has an optimal solution that coincides with one of these vertices. Otherwise, the linear program could achieve unbounded values.

  So what exactly happens when one of the hyperplanes is parallel to our objective equation?

  We can reasonably take any point on that face, but note that a corner will appear there too, so there's no reason to not output the corner. In fact, almost all algorithms end up finding corners anyway, since that's where the extreme points are guaranteed to exist.

15.5 Algorithm for Finding Vertices

- Given m constraints, for each subset of n constraints, we find the intersection x^* of these constraints (do this by switching the inequalities with equalities and performing Gaussian elimination). If x^* satisfies all constraints, then add it to the list of extreme points.
- We could just go through all vertices and find the one with the highest payoff, but this would be incredibly slow:
  - There are $\binom{m}{n}$ different vertices, and this number grows very quickly.
  - In fact, with $m = O(n)$ constraints, we can generate a situation where we have 2^n different vertices: hypercubes!

15.6 Simplex

- An algorithm that allows us to find the best neighboring vertex. Starting at x^*, look at all neighbours of x^*. If x^* is larger than all of these, return x^*. Otherwise, move to the highest neighbour and repeat.
- In higher dimensions, we define the neighbour is to say that two points are neighbours if they are formed by swapping out a *single* constraint.
- Simplex is quite interesting, in the sense that in the worst case it does exactly what the naive solution does, so its worst case runtime is still exponential. However, practically speaking, Simplex is one of the fastest algorithms we know of!

  It's even faster than some algorithms that provably run in polynomial time!

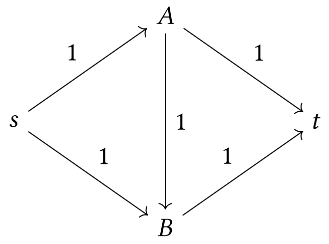
16 Network Flow

- Recently declassified (1999) document about the USSR's shipment capacity from east to west. This was crucial information at the time since had a war broke out, the US could identify which supply routes they could bomb.
- They devised a greedy algorithm called "flooding," but this algorithm wasn't really optimal. It was finally solved by Ford and Fulkersson, and is now called the Ford-Fulkersson algorithm.
- Given a directed graph $G = (V, E)$, one source vertex s and a sink t, and for each edge $e \in E$, we're given a capacity c_e which are integers.
- We want to find the maximum amount of water from $s \to t$.
- *Definition:* A flow assigns a number f_e to each directed edge $e \in E$ such that:
  - nonnegativity: $f_e \geq 0$
  - capacity: $f_e \leq c_e$
  - flow in and flow out are equal: $\sum_{u \to v} f_{u,v} = \sum_{v \to w} f_{v,w}$
- Let's also define the size of the flow f to be the total quantity set from s to t. Using this definition, then the maximum flow is the one that maximizes size(f). This can be solved using linear programming!

16.1 Greedy (suboptimal) algorithm

- We'll find a path P from s to t, and send flow until it's saturated. We'll do this as much as we can. We repeat this until we run out of paths.

- This algorithm fails on some graphs, because it uses edge $A \to B$ when that edge is suboptimal! Consider the graph:



Our algorithm just looks at flow rate, so a possible path to take is $s \to A \to B \to t$, but this is clearly suboptimal! Instead, we should be going from $s \to A \to t$ and $s \to B \to t$.

16.2 Greedy Fix

- We instead consider a residual graph, where we subtract the flow given by greedy ($s \to A \to B \to t$), and also generate a back edge that travels in the reverse order of the flow given by greedy, so that we can backtrack if needed.
- Formally, given a graph G and a flow f on G, the residual raph G_f is defined as: For all edges (u, v), if f goes from $u \to v$, then the residual graph will flow from $v \to u$ and the edge will have capacity $c_{u,v} - f_{u,v}$.

  By doing this, we allow our graph to backtrack along our suboptimal path if needed.
- This is the approach that Ford Fulkerson uses to find the optimal flow.

16.3 Ford-Fulkerson Algorithm

- Find a path P from s to t in the residual graph which is not yet saturated, and send more flow along P. We keep repeating this until everything's saturated, and this happens when all edges along one particular cut is zero.
- To show that this algorithm terminates, lets' first define an $s - t$ cut is a partition of the graph into two sets of vertices L and R such that $s \in L$ and $t \in R$. We define the capacity of this cut to be the sum of all capacities from the edges that cross from L to R.
- Therefore, for any flow f and any cut (L, R), then $\text{size}(f) \leq \text{capacity}(L, R)$. Then, the flow is actually upper bounded by the minimum cut along this graph (this is our "bottleneck" introduced at the outset)
- Then, this means that the max flow is also given by the minimum cut, and we can show that Ford-Fulkerson outputs a maximum flow by considering this relation between the flow and a cut. The proof of this is outlined in lecture.

  Review the proof for this later

16.4 Runtime

- The number of augmenting paths must be less than U, where U denotes the maximum flow, so the update is less than $O(m + n) \cdot U$.
- But what this means
- There are other algorithms out there that optimizes this a little more: Edmonds-Karp gives us a runtime of $O(nm^2)$, which is much better than what we have.

- The best runtime was discovered last year, where we have $O(m^{1+o(1)} \cdot \log U)$

17 Network Flow II

We're going to explroe more LP problems today.

17.1 Bipartite Perfect Matching

- Input: a bipartite (undirected) graph $G = (L, R, E)$ with $|L| = |R| = n$ (so each node has a corresponding pair), and want to output a perfect matching from L to R.

  A perfect matching is one where we can pair every vertex from L matches with exactly one vertex in R.
- Use cases include matching courses and classrooms: you can model a graph where L denotes the courses and R are classrooms, and E denotes whether a classroom can fit a course. So the example of perfect matching is basically asking whether we can assign every course to a classroom.
- To solve this, we convert this problem into one of max flow.

17.1.1 Algorithm

- First copy the graph G to make G', and make all the edges directed from L to R.
- Introduce a source vertex s that connects to every vertex in L, and introduce a sink that every vertex in R connects to. The capacity of each edge will be 1, so this is called "unit flow."
- *Claim:* G has a perfect matching if and only if the max flow on G' is n. (intuitively this also makes sense, since had it not been n, then it would mean that some vertex can't be matched)

  Proof: First assume that G has a perfect matching, and let M denote that perfect matching. We can use this to construct a flow of size n, by putting 1 unit of flow along every edge in M. Then add 1 unit of flow going from s to every vertex, and add one unit of flow going from every vertex in R to t. There are n pairings, so the max flow here is n.

  First recall from last lecture that if the capacities on our graph are integral, then the max flow is also integral. Now, let f be an integral flow of size n in G'. If it's an integral max flow, then we can only assign an integral amount to every edge, and since every edge has capacity 1, then our flow basically selects a subset of the edges (since flow is either 0 or 1 along the edges).

  Each vertex $u \in L$ alwyas has 1 unit of flow on 1 outgoing edge, since the flow from $s \to u$ is 1, and that flow needs to go somewhere. Similarly, each $v \in R$ has 1 unit of flow on 1 incoming edge. So one vertex in u has one outgoing edge in R and one vertex in R has a corresponding matching in L, so this specifies a matching of size n.
- This is a technique called a **reduction** from perfect matching to max flow.

17.2 LP Duality

- Recall that we said last lecture that the max-flow in a graph corresponds to the min-cut on that graph. So we could prove that a flow was optimal by showing a cut of the same value. This idea of correspondence is called **duality**.

- Suppose we wanted to maximize $5x_1 + 4x_2$, subject to the constraints:

$$\begin{aligned} 2x_1 + x_2 &\leq 100 \\ x_1 &\leq 30 \\ x_2 &\leq 60 \\ x_1, x_2 &\geq 0 \end{aligned}$$

  If we were to solve this, we'd get $x_1 = 20, x_2 = 60$ for a maximum value of 340. (You can check that this is a valid assignment). How do we check that this is the maximum value? Well, we can combine the inequalities in a clever way, which would end up getting us $5x_1 + 4x_2 \leq 340$, which proves that our solution was optimal.

  To show that this was optimal, we essentially multiplied the inequalities by values y_1, y_2, y_3 that gave us the optimal value. This meant that we were generating the equation:

$$(2y_1 + y_2)x_1 + (y_1 + y_3)x_2 \leq 100y_1 + 30y_2 + 60y_3$$

  We want to set y_1, y_2, y_3 such that the LHS is larger than our objective function, while making the RHS as small as possible. Basically, this means that we want to minimize $100y_1 + 30y_2 + 60y_3$ while also requiring that

$$\begin{aligned} 0 &\leq y_1, y_2, y_3 \\ 5 &\leq 2y_1 + y_2 \\ 4 &\leq y_1 + y_3 \end{aligned}$$

  So this is basically *another* LP problem! So the original problem is called the *primal LP*, and the second one is called the *dual LP*. Because of the way we set this up (with the maximization/minimization), it guarantees that any satsifying assignment of the primal LP will be less than that of the dual LP. This is the *magic trick*: the fact that we can always convert any maximization LP into a minimization in the dual LP.

  Turns out that if we take the dual twice, we get back the original LP.

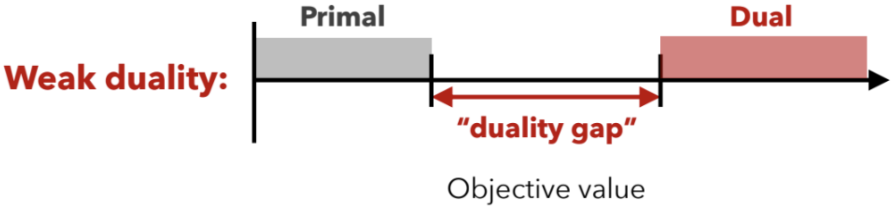
- This is a fairly standard exam problem: given a primal LP, compute the dual.
- There is also a matrix representation of the same problem. Recall the standard form for expressing LPs as maximizing $c^\top \vec{x}$ subject to the constraints given by $A \cdot \vec{x} \leq \vec{b}$ and $\vec{x} \geq 0$.
- To construct the dual, we want to minimize the quantity $\vec{b}^\top \cdot \vec{y}$, where b used to be the constraints earlier. Our constraints are given by $A^\top \cdot \vec{y} \geq \vec{c}$ and $\vec{y} \geq 0$
- The theorem we saw earlier is called **weak duality**, which says that all feasible solutions x to the primal LP are less than the solutions y to the dual LP. The proof is as follows:

  The primal LP maximization is written as $\vec{c}^\top x$, but we know that $\vec{c}$ has another representation as $A^\top \vec{y}$, so we can write:

$$\vec{c}^\top \vec{x} \leq (\vec{y}^\top A)\vec{x} \leq y^\top b$$

  Then, since $y^\top b = b^\top y$, then we have the inequality $c^\top x \leq b^\top y$.

- Graphically we can visualize this as:



- *Theorem:* If the primal LP is bounded (by any number), then the optimal solution to the Primal LP is equal to that of the dual LP. So in this case, the duality gap is 0

  Doesn't this apply to most cases?

- The max-flow min-cut principle we had earlier is an example of an LP problem and a corresponding dual. So another way of proving what we did was to show that the min cut can also be written as a LP. Then, we can show that they are the duals of each other, and since the max-flow problem is always bounded, then we know that the optimal solution is when they are equal by strong duality.

17.3 Zero-Sum Games

- Input: a "payoff" matrix M, and two players: a row and column player.
- The matrix M specifies the amount that the row player wins, and the column specifies how much the column player wins. An example of this is rock paper scissors:

	Rock	Paper	Scissors
Rock	0	-1	1
Paper	1	0	-1
Scissors	-1	1	0

- In general, the row player selects a row r and the column player picks a column c. Then, we say that the row player wins $M_{r,c}$, while the column player wins $-M_{r,c}$.
- This is a zero-sum game because the sum of every row and column is 0.
- There are two strategies the players are allowed:
  - Pure strategy: pick one row/column and play their selection (e.g. row player always picks rock)
  - Mixed strategy: a probability distribution over pure strategies (e.g. we can have $P[\text{rock}] = \frac{1}{3}, P[\text{paper}] = \frac{1}{3}, P[\text{scissors}] = \frac{1}{3}$)

    Also notice that the average score across all these strategies is 0, no matter what the column player does.

17.3.1 Game 1

- The game is described as:

	P1	P2
P1	3	-1
P2	-2	1

- We want the row player to announce their strategy first, then the column player announces their strategy second. Because this is slightly unfair to the row player, we let the row player announce a mixed strategy $P = (p_1, p_2)$ where p_i is the probability that row i is chosen.
- The column player will respond by choosing a mixed strategy of their own, with distribution $Q = (q_1, q_2)$ where q_i is the probability of choosing row i.
- Then, the row player's average score is $S(p, q)$ is the expected value that they get. So in this case:

$$S(p,q) = 3p_1q_1 - p_1q_2 - 2p_2q_1 + p_2q_2$$

  So, the column player's best strategy is to minimize $S(p, q)$, while the row player wants to maximize $S(p, q)$.

- For the column player, they will choose the best strategy for themselves, after having seen p. So, since the column player knows what p the row player chose, then their minimization is the same as just minimizing over pure strategies:
$$\min_{\text{mixed strategies}} \{S(p,q)\} = \min_{\text{pure strategies}} \{3p_1 - 2p_2, p_2 - p_1\}$$

- The row player goes first, so their strategy is that they would want to maximize the column player's response. So for them they want to compute:
$$\max_{\text{mixed strategies}} \{\min\{3p_1 - 2p_2, p_2 - p_1\}\}$$

- We'll see next time that this is exactly the same as computing two LPs, which are duals of each other!

18 Zero Sum Games

18.1 Game 1

- We saw last time that given a ZSG structured like this:

	P1	P2
P1	3	-1
P2	-2	1

  then we can calculate that the payoff for column 1 is $3p_1 - 2p_2$, and the payoff for column 2 is $-p_1 + p_2$.

- This meant that the column player's best strategy was to look at these two values, and minimize this.

- The row player will then pick p_1, p_2 in order to maximize what the column player tries to minimize. Therefore, the row player wants to calculate:
$$\max_{\text{mixed strategies } p} \{\min\{3p_1 - 2p_2, -p_1 + p_2\}\}$$

- As we said last lecture, this can be solved using LP! We'll also see that the row player going first doesn't hurt them.

- So we can formulate our LP as follows:

  Maximize a quantity z, subject to the constraints:
$$\begin{aligned} z &\leq 3p_1 - 2p_2 \\ z &\leq -p_1 + p_2 \\ 1 &= p_1 + p_2 \\ 0 &\leq p_1, p_2 \end{aligned}$$

  Note that $z = \min\{3p_1 - 2p_2, -p_1 + p_2\}$, so by maximizing z subject to these two constraints means that z is constrained to the smaller of these two.

18.2 Game 2

- The exact same game board, except we allow the column player to go first and the row player goes second.

	P1	P2
P1	3	-1
P2	-2	1

- We do the exact same thing, by considering the possibilities from the row player's perspective. This is the same as looking from the column player's perspective in the previous problem.
  - From the row player's perspective, payoff of row 1 is $3q_1 - q_2$, and row 2 is: $q_2 - 2q_1$. So we want to choose the max of these two, i.e.:
  $$\max\{3q_1 - q_2, -2q_1 + q_2\}$$
  - The column player will try to minimize this score, so they'll want to choose:
  $$\min_{\text{mixed strategies } q} \{\max\{3q_1 - q_2, -2q_1 + q_2\}\}$$
- This is another linear program! Here, we want to minimize z, subject to:
$$\begin{aligned} 3a_1 - a_2 &\le z \\ -2q_1 + q_2 &\le z \\ q_1 + q_2 &= 1 \\ q_1, q_2 &\ge 0 \end{aligned}$$

  Again, note that z is equal to the larger of the two inequalities, using the same logic as before.
- Now we compare the two problems, and compare the final quantities that we wanted to compute.
  - In game 1, we wanted to find $\max_p\{\min_q\{S(p,q)\}\}$
  - In game 2, we wanted to find $\min_q\{\max_p\{S(p,q)\}\}$
- Given this construction, we have the inequality
$$\max_p\{\min_q\{S(p,q)\}\} \le \min_q\{\max_p\{S(p,q)\}\}$$

  If the dual of the dual is the original, doesn't this mean that we get a situation like $x \le y \le x$?

  No, becuase the inequality flips again, so we get $x \le y \ge x$, a valid inequality.

  The best way to see that this is true is that it's always better to be the player that goes second. Therefore, finding the minimum over q is better because they're able to "react" to the column player's moves.
- It turns out that these two LPs are actually duals of each other (the proof is to construct the second LP from the first one). Now, recall the property of strong duality, which holds for any LP that is bounded. This game is clearly bounded, so we know that there is an optimal value of the game (denoted by Value(game)) is the same for both games!

  This is called the **min-max theorem.**
- This tells us that the order of play doesn't change the value of the game, and there is an optimal probability distribution that the row player can choose without caring about what the column player does.

  Note that this is a zero sum game not by the game table, but because the gain of one player is an equal loss in the other player. So the numbers in the table can be anything!
- This also says that all zero sum games are strongly dual.

18.3 P vs. NP

- So far, we've seen a lot of algorithms: polynomial multiplication, MSTs, APSP, etc.
- In theoretical CS, we consider all these problems to be "efficiently solvable," We define this to be the case when a problem can be solved in **polynomial time.**

  This is only in theoretical CS. In practice, we'd want to get everything down to $O(n)$ time, if possible. Even $O(n^2)$ is quite bad, since given an input of size 10^9 (as is the case with the facebook graph), then the computation is on the order of 10^{18} (very bad)!

- We define P (stands for polynomial) to be a set of computational problems that are considered to be efficiently solvable.
- We define NP (stands for non-polynomial) to be another complexity class that aren't efficiently solvable themselves, but whose solutions can be efficiently checked.
  - Example: 3-coloring problem. We want to find a 3-coloring on this graph. Naively, we can brute-force and try all possible combinations, which would correspond to checking 3^n possible graphs.
  - The best known algorithm for this solves the problem in 1.3289^n time.
  - So this algorithm is not in P, but it is in NP, since any solution can be verified in polynomial time (by checking all edges)
  - Example 2: Factorization. Given an n-bit integer N, we want to factorize it into two numbers $p, q > 1$ such that $pq = N$.
  - Naively, we can divide N by every number from 1 to $\sqrt{N}$. In terms of time complexity, this is $O(\sqrt{N})$, which we can simplify this to $O(2^{n/2}$ due to the way numbers are represented as bitstrings.

    The best algorithm runs in time $C^{n^{1/3}} \log(n)^{2/3}$, so this is not a problem that's known to be in P.

    However, the problme is in NP, because we can just verify any solution by multiplying them together, which takes at most $O(n^2)$ time.

19 P, NP, Reductions

- Recall that P is a "complexity class" of problems that are efficiently solvable (by efficient, we mean polynomial time), and NP being the complexity class of problems that can be verified efficiently. (again, polynomial time)
- Examples of problems in NP are: 3-color problem, TSP, factoring integers.

19.1 Rudrata Cycle (Hamiltonian cycle)

- Input: a graph $G = (V, E)$, and we output a tour that visits each vertex exactly once.
- We can just run through all $n!$ cycles in the graph, which is highly inefficient. The best known algorithm is $O(1.657^n)$ (this is faster than the DP solution), so this is not a problem in P!
- This problem is in NP, since given a string of vertices to visit, we can just go to our graph and check. This can be done in polynomial time.
- There are often modifications we can make to the problem that seemingly don't make that big of a difference, but drastically simplify the problem: finding an Eulerian tour (which adds the constraint that every edge must be visited), for instance, is a problem in P.

19.2 Traveling Salesperson Problem (TSP)

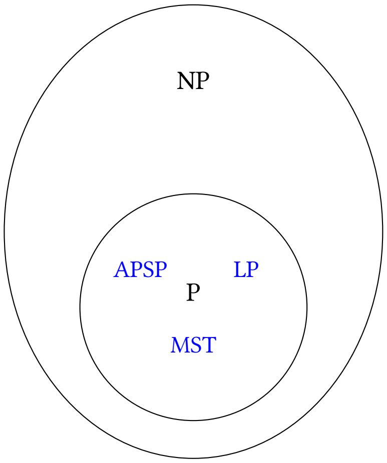
- Input: a graph $G = (V, E)$ with edge weight, and there are three variations to this problem:
  - **Optimization TSP:** We're asked to find the tour with the minimum total weight. Our DP solution gets a runtime to $O(n^2 2^n)$, but this isn't polynomial time, so this isn't in P.

    It's also not in NP, since there's no way to check whether a given tour is minimized without knowing the weights of all other tours.
  - **Search TSP:** Find a tour with total weight $\leq B$, where B denotes our "Budget". This problem is not in P since coming up with a tour is still hard, but it is in NP, since verifying that the budget is less than B, can be done in polynomial time.

    Notice that if we can solve Search TSP in polynomial time, this also gives a solution to Optimization TSP in polynomial time!
  - **Decision TSP:** Asks whether there exists a tour with weight $\leq B$. Again, just like Search TSP, this problem is not in P but is in NP.
- There are a list of problems that are known to not be in NP. Examples are: some optimization version of problems, counting problems (counting number of 3-colorings, for example), the halting problem.
- Formally, NP is only defined for **decision problems**. (CS172 material) For this class, we'll be looser and allow search problems as well.

19.3 P vs. NP

- **Theorem:** P ⊆ NP. In other words, any problem that can be *solved* in polynomial time can also be *verified* in polynomial time. The verification algorithm can literally just be to solve the problem again and check that the solutions match.
- A complexity diagram is usually used to show where problems lie:



- Largest open problem in theoretical computer science: is P = NP?

19.4 Reductions

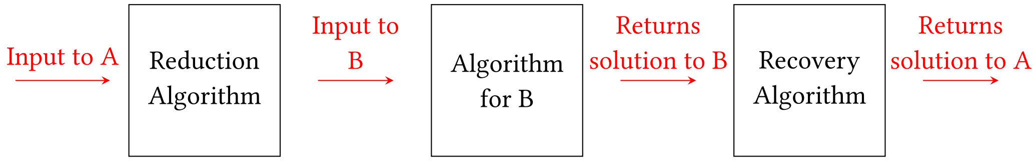
- We say that a problem A *reduces in polynomial time* to problem B if you can use any efficient algorithm for B to efficiently solve A. Mathematically, we'd write this as

$$A \preceq_p B$$

Another way to say this is that “A’s difficulty is less than B’s difficulty”, or that “A is at most as hard as B.”

19.4.1 Zero Sum Games

- Recall the input: a payoff matrix M, and we want to output the Row player’s optimal strategy.
- **Theorem:** zero sum games $\preceq_p$ Linear programming
  - In essence, we transfer the ZSG into a linear programming problem via a reduction algorithm. Then, take the LP input and run the LP algortihm on it (which is polynomial time), then run a recovery algorithm to go from the LP back to the ZSG.
  - As a general picture, we have



19.4.2 Rudrata Cycle vs. Min TSP

- It can also be shown that finding a Rudrata cycle reduces to the min-TSP problem, since both problems require us to find tours, then finding a Rudrata cycle is at most as hard as min-TSP.
- This is because finding a Hamiltonian tour is effectively finding a cycle on this graph that visits all vertices, which is exactly what min-TSP is doing except with more restrictions.

20 Reductions II

- Recap: Two computational problems A and B, and A reduces (in polynomial time) to B is written as $A \preceq_p B$. This means that if an algorithm exists to solve B in polynomial time, then that same algorithm can be used to solve A in polynomial time.

  Is this restricted to only polynomial time? Shouldn’t any feasible algorithm that solves B also solve A?

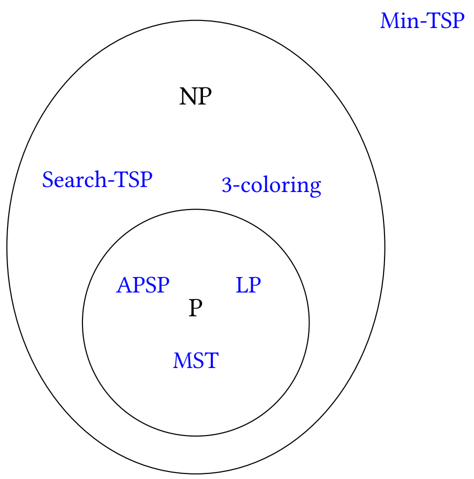
  Why are we so concerned about polynomial time? Do similar problems exist if we define “efficient” to be exponential time?
- Also recall the diagram we made to represent this process of reducing A to B, via a polynomial time reduction and recovery algorithm. Note that these two algorithms **must** execute in polynomial time.
  - We can prove that $A \preceq_p B$ even if A, B are not known to be efficient.
- We also saw two reductions: zero sum games reducing to LP, and Hamiltonian cycle reducing to min-TSP.
- Transitivity: If $A \preceq_p B \preceq_p C$, then $A \preceq_p C$.

20.1 Common mistakes in Reductions

- If we’re asked to prove that $A \preceq_p B$, we need to come up with an algorithm that takes A to B, not B to A. Make sure you check that you’re proving the correct direction!

20.2 Landscape of Problems

- We’re going to use the below diagram to show the problems:



- We're not going to prove this, but it has been shown that factoring reduces to the 3-coloring problem. Similarly, factoring also reduces to the Rudrata Cycle problem.
- It turns out that every problem in NP reduces to Rudrata cycle!
- These are the most difficult problems in NP, and it can be shown that every problem in NP reduces to an NP-complete problem
- **NP-Hardness:** A problem A is NP-hard if every problem B in NP reduces to A.
- **NP-Completeness:** A problelm A is NP-complete if $A \in$ NP and A is NP-hard.
- Problems in NP that aren't NP-complete are called an **NP-intermediate** problem
- **Fact:** Given two problems that are NP-complete, then $A \preceq_p B$ and $B \preceq_p A$. So this means that you can basically think of A and B are basically equivalent problems.

  Is this a biconditional?

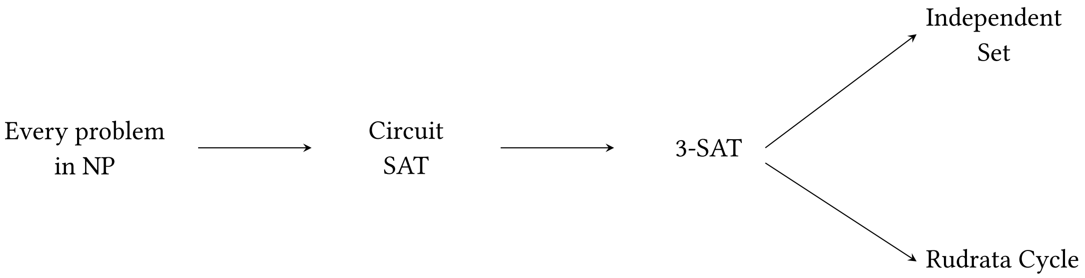
  No, consider two problems in P: they can be reduced to one another, but they are not in NP.

  - There are thousands upon thousands of NP-complete problems, and by notion of reduction, they are (in some sense) the same problem.
  - This also means that if there exists a polynomial time algorithm for any NP-problem, then this would imply that P = NP.

  How is it that if P = NP then every problem becomes NP-complete?

20.3 Proving NP-Completeness

- Cook-Levin Theorem: showed that every problem in NP reduces in polynomial time to a circuit SAT problem.
- It can then be shown that circuit-SAT reduces to 3-SAT, making 3-SAT an NP-complete problem. In terms of a diagram:



Finish this Diagram Later

- To show that a problem is NP-complete, we first show that $A \in \text{NP}$, then pick some problem B that is known to be NP-complete and show that $B \preceq_p A$.

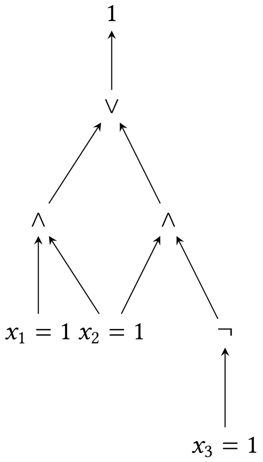
  What if we show that $A \preceq_p B$?

  We don't need to, since $A \preceq_p B$ is true already because B is an NP-complete problem!

20.4 Circuit SAT

- A Boolean circuit is a directed acyclic graph with:
  - Input nodes $x_1, \dots, x_n$
  - one output node, with an output $C(x)$
  - gates marked OR, AND, NOT: $\vee, \wedge, \neg$

  A possible graph is:



- The input to circuit SAT is a circuit C with n inputs and m referring to the number of gates. We want to output an assignment of $(x_1, \dots x_n)$ such that $x_i \in \{0, 1\}$ such that $C(x) = 1$.
- By the Cook-Levin theorem, circuit-SAT is NP-complete. As for a bit of intuition on why this is true, you can think of every problem as basically a collection of logical inputs, which basically means that every problem can be reduced to some complex circuit of logical gates.

20.4.1 3-SAT

- Here, we're given n Boolean variables $x_1, \dots, x_n$ such that $x_i \in \{0, 1\}$, and $m \leq 3$ variable clauses that join the variables together.

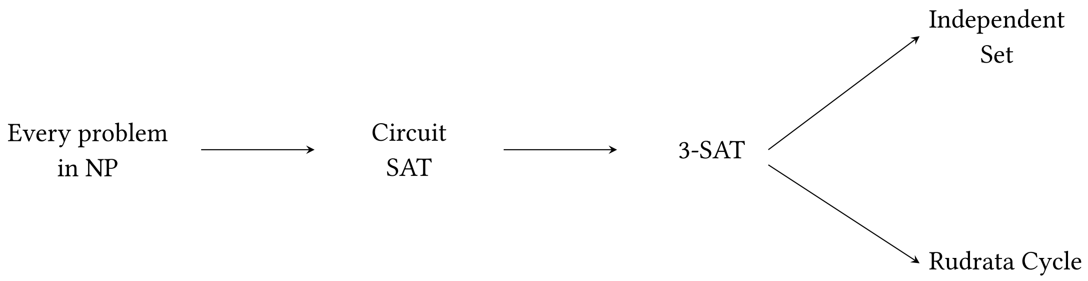
- We want to output an assignment of $x_1, \dots, x_n$ that satisfies all the clauses.
- **Theorem:** Circuit-SAT reduces to 3-SAT

  Proof: Suppose we're given an input to a circuit-SAT problem.

21 NP Completeness

- Let's revisit the independent set problem. Earlier we did this on trees, but now we're going to move to a general graphs.
- Given a graph $G = (V, E)$ with n vertices and m edges, along with an integer $1 \leq k \leq n$
- We want to return an independent set of size k.
- A naive algorithm for this problem is to just try all sets of k vertices. This takes $O\binom{n}{k} \in \Omega(n^k)$
- This is good if k is small, but gets significantly worse as k gets large, say $k = n/2$. In this case, then the algorithm runs in approximately $O(n^n)$ time.
- In the real world, what we try to show is that this is an NP-hard problem – in other words, we should stop trying to look for an efficient solution.
  - The approach hinges on the fact that problems in NP-hard are not believed to have efficient solution algorithms, so we should stop trying basically.
  - By showing that a problem is NP-hard, it also shows that it's NP-complete.

    Is this because every NP-complete problem reduces to a NP-hard problem?
- Our approach to showing that a problem is NP-hard is to show that a well known NP-complete problem reduces to our problem of interest.
- Recall our tree of problems from last lecture:



  And also recall our approach to showing NP-completeness:

  1. We first show that $A \in$ NP.
  2. We show NP-completeness by showing that an NP-complete problem B reduces to A.

  By doing this, we show that a polynomial algorithm that solves A will also solve B.

21.1 NP-hardness of Independent Set

- We want to choose an NP complete problem to reduce to independent set. Here, we'll choose the 3-sat problem.

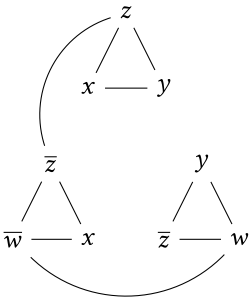
  Are all NP-hard problems also NP-complete?

- Recall the 3-sat problem, where we're given an instance ψ :

$$(x \vee y \vee z) \wedge (\overline{z} \vee \overline{w} \vee x) \wedge (y \vee \overline{z} \vee w)$$

  and we want to find an assignment for x, y, z, w such that this instance returns true.

- To show reduction to independent set, we go to each clause and make a "subgraph" consisting of the variables in the clause. We do this for all three clauses:



- Note that in our graph, we don't connect the two y 's that appear, since they are part of different clauses.
- For the variables that are set to 1, this corresponds to selecting that vertex for our independent set instance.
- For all variables that contain the literal and its negation, we should also connect an edge between those because we want only one of x and $\overline{x}$ to be selected.
- Note that for every set of 3 vertices, we can only select one of the three vertices in order for the clause to remain independent.
- This also suggests our polynomial-time recovery algorithm: whenever x is chosen in the independent set, then set $x = 1$. If $\overline{x}$ appears, then set $x = 0$. If it doesn't appear at all, then we don't care what it's set to.

21.1.1 Formal Proof of Reduction

- There's always two parts these kinds of proofs: we have to show that the reduction algorithm works, and also that the recovery algorithm works.
- For the first part: we want to show that if the 3-sat instance ψ has a satisfying assignment $A : \{x_1, \ldots, x_n\} \to \{0, 1\}$, then graph G has an independent set instance I of size $k = m$.
  - For each clause $x_i \vee \overline{x_j} \vee x_\ell$, we know that A sets one of $x_i, \overline{x}_j, x_\ell$ to 1.
  - WLOG choose x_i. Now, we'll add node x_i to the independent set.
  - This means that our instance I is indeed of size $m = k$.

    why is this true?

    because k is the number of clauses, and we have to set one of these to 1 for every clause!
  - We know that our selection rules state that we never select both x_i and $\overline{x_i}$, so therefore we guarantee that our set is independent.
- Now for part 2: Let I be an independent set in G of size k. We show that the recovery algorithm outputs a satisfying assignment.

  Recall that our recovery algorithm has variables x_i, and it sets $x_i = 1$ if $x_i \in I$, and $x_i = 0$ if $\overline{x_i} \in I$.
  - For each i, our recovery algorithm either finds x_i or $\overline{x_i}$ (and never both), so our recovery algorithm is well defined.

  - Further, based on the way that we constructed our graph, only one of the nodes in any subgraph will be selected, meaning that it makes that particular clause return true. Therefore, we do output a satisfying instance.

21.2 Clique

- Here, we're given a graph G and an integer $1 \leq k \leq n$, and we want to output a clique of size k.
- To show that this problem is also NP-hard, we show that independent set reduces to Clique.
  - To reduce, we "invert" the graph by creating a graph $\overline{G}$, which contains the same vertices v but the edges not in G. Then, if we can find an independent set on $\overline{G}$, then we know that in the original graph G it must be a clique.
  - The recovery algorithm is basically the same thing but in reverse: we just return the vertices found in the independent set.

21.3 Rudrata (s, t)-path

- Given a graph $G = (V, E)$ and vertices s, t, we want to find a path starting at s, end at t, while also visiting each vertex exactly once.
- Here, we're going to show that this problem reduces to another NP-complete problem, instead of the other way around. Specifically, we're going to show that this reduces to the Rudrata cycle problem.

  We're doing this only as an example to further show reduction, this does not have anything to do with proving hardness
- *Proof:* Given a Rudrata (s, t)-path instance, we can generate a reduction algorithm as follows:

  Review this later

21.4 Double Cycle

- Given a graph $G = (V, E)$ with n vertices, with the restriction that n is even.
- We want to find two disjoint cycles of size $n/2$ vertices, which will visit each node exactly once.
- Again, we will show that this reduces to rudrata cycle.

22 Approximation Algorithms

- Suppose that you have a problem that is NP-hard. What now? There's a couple things we can still do:
  1. Learn more about its inputs
  2. Come up with a Heuristic
  3. Approximation Algorithm (this is what we'll study today)
- For a minimization problem, an algorithm A is an α-approximation problem if $A(I) \leq \alpha \cdot \text{OPT}$, where OPT is the optimal value. For a maximization problem, then the inequality reads $A(I) \geq \alpha \cdot \text{OPT}$.

22.1 Vertex Cover

- Given an input graph $G = (V, E)$, and we want to return a vertex cover $C \subseteq V$ of minimal size.
- A **vertex cover** is a set of vertices such that every edge (u, v) is incident on one of them.

- This is indeed an NP-hard problem – it reduces from Independent set.

  The NP-hardness comes from the fact that we want C of minimal size

22.2 Approximations to Vertex Cover

22.2.1 Algorithm 1: Maximal Matching

- The question we want to ask is: what is the easiest problem we can compute that is similar to the problem we ultimately aim to solve?
- In this case, it turns out to be the maximal matching problem.
- **Definition:** A *matching* in a graph G is a set of edges with no overlapping vertices. The matching is then maximal if we can't add any more edges to the matching.
- We can compute this using a greedy algorithm: add edges to the matching, and delete all edges adjacent to it.

  Is this problem also NP-hard, but there turns out to be a greedy algorithm that does it in a relatively simple manner? Also, why can't we just use a greedy algorithm on the original problem?

  Consider the “star graph" consisting of a central node connected to others. The maximal matching would output a size of 2, but a greedy algorithm could potentially pick $n-1$ nodes (going around the circle). Hence, it's not optimal. Even with the modification to choose the vertex with the highest degree, this is still not optimal, see textbook for counterexample.

- Now we convert this into a vertex cover C by outputting both endpoints of every edge found in the matching M. We now prove that this is a valid vertex cover and that $|C| \leq 2 \cdot \mathrm{OPT}$.
- We prove this in two parts:

  Claim 1: C is a vertex cover.

  Proof: Assume for contradiction that C is not a vertex cover. Then, there is some edge $(u, v) \in E$ where $u, v \notin C$. But this is a contradiction, because then the Greedy algorithm would have selected this edge as well.

  Claim 2: $|C| \leq 2 \cdot \mathrm{OPT}$.

  Proof: Any vertex cover covers every edge in the matching M. Therefore, the vertex cover includes one of (u, v) or potentially both. Therefore, the worst case is if both u, v are selected in the matching instance when only one was required, but this gives us an upper bound of $2 \cdot \mathrm{OPT}$.

  Can we do better? Why can't we get rid of adjacently selected vertices at the end?

  There is no restriction on adjacency of vertices in vertex cover, which is why the output actually works as a solution to vertex cover.

22.2.2 Algorithm 2: LP

- We can also formulate this problem as a linear programming problem.
- Assign a variable x_i for each vertex i, and ideally, we want to set x_i to 1 if vertex i is in the vertex cover, and 0 otherwise.
- To formulate this as an LP, we'd want to minimize $\sum_i x_i$, while subject to:

$$0 \leq x_i \leq 1$$
$$x_i + x_j \geq 1 \ \forall (i, j) \in E$$

- *Claim:* The optimal value for the LP $\leq$ the optimal vertex cover.

  Proof: The optimal value for the LP could give us an infeasible (but smaller value) solution because it's allowed to give us fractional x_i, but the optimal set cover is a feasible solution to the LP, so the LP either finds the optimal vertex cover, or an inadmissible value.

- We now want to convert this to an optimal set cover, by doing an operation called **rounding.**

- Let $\{x_i^*\}$ be the optimal LP solution, and we will round based on the following rule:

$$x_i^* \geq \frac{1}{2} \implies \text{Include in } C$$
$$x_i^* < \frac{1}{2} \implies \text{Do not include in } C$$

  Claim: C is a vertex cover.

  Proof: For every edge (i, j), we have a constraint that $x_i^* + x_j^* \geq 1$, meaning that $\max(x_i^*, x_j^*) \geq \frac{1}{2}$, so our rounding table would select one of these two vertices to be in C.

  Claim: $|C| \leq 2 \cdot \text{OPT}$.

  Proof: For all vertices, the actual vertex cover pays 1 unit to some x_i, while our LP rounding algorithm pays $x_i^* \geq \frac{1}{2}$ every time, so we're never doing worse than twice the optimal LP. Therefore, $|C| \leq 2 \cdot \text{LP-OPT} \leq 2 \cdot \text{OPT}$.

  What is this concept of "paying" here? I'm not sure I understand this argument.

  Paying is probably not the best way to understand it. Another way to phrase this is that the worst case scenario for our LP approximation is if it assigns $x_i^* = \frac{1}{2}$ to all nodes, in which case our rounding would output all the nodes for the vertex cover. However, we know that at most we only need half of these nodes in the vertex cover (one per edge), hence the approximation factor of 2.

22.3 Metric TSP

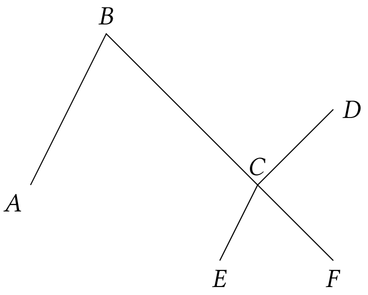
- Recall the TSP problem, where we have n cities and pairwise distances d_{ij}.
- We want to output a minimum distance tour while still visiting every node exactly once.
- It's known that you can't even solve this with an approximation factor!
- But now we impose the restriction of the triangle inequality: for all cities i, j, k, then we have:

$$d_{ij} + d_{jk} \geq d_{ik}$$

  and $d_{ij} \geq 0 \; \forall i, j$. With this approximation, then we *can* derive an upper bound!

22.3.1 Algorithm: MST

- Notice that with this restriction, this is very similar to the MST problem, except MST finds a tree rather than a cycle.
- So we first find the MST on G. Then, we know that cost(MST) $\leq$ cost(OPT), since the TSP also visits every vertex, and MST is known to be the set of edges that has the minimum weight.



How is this even a tour?

It's not, the point is that we first do a DFS here and then find a tour by removing the repeated vertices.

- Now we explore the tree T using DFS, starting at A. Then, the DFS visits all the vertices then comes back to A at the end. Because it takes every edge in the MST twice, then we know that

$$\text{cost}(\text{DFS}) = 2 \cdot \text{cost}(T)$$

- Finally, we now skip over all the repeated vertices in the traversal: suppose we visited node D, and we explore E, and DFS gives the path $E \to D \to F$, then we're wasting time by revisitng D, and it's much more optimal to just go from $E \to F$ by the triangle inequality.

  Then, we have the inequality:

$$\text{cost}(\text{TSP}) \leq \text{cost}(\text{DFS}) \implies \text{cost}(\text{output}) \leq 2 \cdot \text{OPT}$$

Can we do better than 2?

We can do $\frac{3}{2}$ using linear programming, and as of last year we now have a $\left(\frac{3}{2} - \epsilon\right)$-approximation (but ϵ is very small

23 Sampling and Streaming

23.1 Sampling

- Suppose you have a question you want to ask like "do you approve of the U.S. Congress?" that admits a yes (1) or no (0) answer. Our goal is to estimate the fraction of population who say yes.
- The naive approach would be to just ask everybody what their opinions are: and we can be certain of our result.
- Instead of doing this, we take a sample population of k people chosen at random, and collect their answers $x_!, \ldots, x_k \in \{0, 1\}$, and output the fraction of those that say yes.

$$\hat{p} = \frac{1}{k} \sum_{i=0}^{k} x_i$$

- Our goal is to pick k such that with probability of $1 - \delta$, then

$$|\hat{p} - p| \leq \epsilon$$

  where p denotes the **true value** of $\hat{p}$.
- The larger the value of k, the smaller our ϵ can be.

- **Chernoff Bound:** Suppose $x_1, \dots, x_n \in \{0, 1\}$ are independent and identically distributed random variables where $P(x_i = 1) = p$ and $P(x_i = 0) = 1 - p$, then

  $$P\left(\left|\frac{1}{k}\sum_{i=1}^{k} x_i - p\right| \geq \epsilon\right) \leq 2e^{-2\epsilon^2 k}$$

  How does this relate to the central limit theorem?

- So in order to get $P(\text{error} \geq \epsilon) \leq \delta$, then we choose:

  $$k = \left\lceil \frac{1}{2\epsilon^2} \ln\left(\frac{1}{2\delta}\right) \right\rceil$$

  There's a mistake in the notes where it says $\log_2$, but it should be ln instead.

23.2 Streaming

- Suppose there's a stream and we're watching fish go by, and we want to know some things about the fish in the stream:
  - How many fish went by?
  - What fraction of fish were red?
  - How many fish species are there?
- The issue is, there are too many fish in the stream to record them all (basically no access to memory). Further, we can never rewind the clock: once a fish goes by, it doesn't go by anymore.
- Streaming algorithms are actually used in the real world! (e.g. tracking packets over the internet)

23.3 Sampling from a Stream

- Given a stream $s_1, \dots, s_i$ (we don't know how many elements there are beforehand), and we want to output a uniformly random element from the stream.
- One way to do this is to record all elements and output a random s_i. However, this takes up a lot of space and isn't preferred.
- If L (length of stream) is known, then we can pick a random index $i = \{1, \dots, L\}$ and output a random s_i.
- Neither of these two models work for our problem, because we don't know L and the first one is just downright slow.

23.3.1 Reservoir Sampling

- We start with $r = s_1$.
- Then, for each new stream element, we flip a biased coin with probability $p_i = 1/i$ of heads and tails with $1 - p_i$. If heads, replace $r = s_i$. Otherwise, leave r as is.

  Some intuition on why we might expect $1/i$: at every step i, we have to handle the case where the stream stops there, in which case we want s_i to be selected with probability $1/i$.

- When the stream stops, we output r.

- This probability is actually uniform! In order to output s_i, then the coin would have flipped heads on element i, then tails at every i after that. This happens with probability:

$$P = \frac{1}{i}\left(1 - \frac{1}{i+1}\right)\left(1 - \frac{1}{i+2}\right)\dots\left(1 - \frac{1}{L}\right) = \frac{1}{i}\cdot\frac{i}{i+1}\dots\frac{L-1}{L} = \frac{1}{L}$$

  Note that the flips before i don't matter, since if flip i is heads then r is replaced anyway.

23.3.2 Distinct Elements

- Again, given a stream s_i, we want to estimate the number of distinct elements in the stream.
- To solve this, we pick a random hash function $h : \{1, \dots, N\} \to [0, 1]$. As the stream goes by, we compute $h(s_i)$ for each element. We only keep one value, that value being the minimum of $h(s_i)$, and call it α. Then, when the stream ends, we output $1/\alpha$.

  Intuition for why $1/\alpha$ should be returned: suppose there's k elements in the stream. As our algorithm goes, then we're only going to see the hash values for those k elements that have been seen. Call these values $r_1, \dots, r_k$. Since the hash function is random, we know that they should be (in expectation) evenly distributed on $[0, 1]$. Therefore, the distance between the hash values is $\frac{1}{k+1}$ and since α is the minimum value then $\alpha = \frac{1}{k+1}$. Therefore, $\frac{1}{\alpha} = k + 1$, which works as an estimation.

  You can subtract 1 off the result, but because we're just estimating it really doesn't matter.

  How does the hash function know from 1 to N?

  N is given to you beforehand.
- There are a couple issues with this, mainly that computers can't store arbitrary real numbers, and the hash function takes a lot of memory to store. One solution (preview for next time) is to use a *pseudorandom hash function*.
  - One way to circumvent the first problem is to have a hash function h map from $\{1, \dots, R\}$ where R is some very large number. This way, $h(i)/R$ is "random enough".
  - For the second issue, we make h a pseudorandom hash function. This takes less space, while still guaranteeing that we get the randomness we want.

24 Streaming II

- Recall the way we computed the number of distinct elements using a random hash function $h : \{1, \dots, N\} \to [0, 1]$.
- There were problems with this approach! There were two:
  - Computers can't store arbitrarily real numbers

    Solution: Pick a hash function that maps $h : \{1, \dots, R\}$ instead of $\{1, \dots, N\}$ so that $h(i)/R$ is basically a random number between 0 and 1. This is fairly easy to implement.
  - If the hash function is uniformly random, then we would require $N \log R$ bits to store this – this takes up a lot of memory, especially as R grows large!

    Solution: Make the hash function h "pseudorandom". We will define a *hash family* $\mathcal{H} = \{h_1, \dots, h_m\}$, and we will choose from this family for our hash function. We will use $h \sim \mathcal{H}$ to denote choosing a random h_i. This approach only requires $\log m$ bits to store, so this is much better than our earlier implementation.

Each of h_i are basically constant functions, we've changed the randomness of the hash function to *choosing* the hash function itself.

So are the h_i just random bits, then how are we guaranteeing that we can still get the number of distinct elements?

Just in the same way the hash function expects a spacing of $\frac{1}{k+1}$, we expect that the randomly generated hash family is also $\frac{1}{k+1}$.

24.1 Pairwise Independence

- This is the way we're going to define randomness.
- **Definition:** A hash family $\mathcal{H} = \{h_1, \dots, h_m : \{1, \dots, N\} \to \{1, \dots, R\}\}$ is *pairwise independent* if:
  - For all $x \neq y \in \{1, \dots, N\}$ and $i, j \in \{1, \dots, R\}$, then

    $$\Pr_{h \sim \mathcal{H}}[h(x) = i \text{ and } h(y) = j] = \frac{1}{R^2}$$

    where Pr denotes the probability. In other words, it means that our i, j look like two independent draws from $\{1, \dots, R\}$.
- If this is true, it also implies that $\Pr_{h \sim \mathcal{H}}[h(x) = i] = \frac{1}{R}$ for all x and i.

24.1.1 Generating Pairwise Independence

- Our approach will utilize modular arithmetic, and to make things easy we will do modular arithmetic using a prime p.
- Then, for each $a, b \in \mathbb{Z}_p = \{0, 1, \dots, p-1\}$, we will define $h_{a,b} : \mathbb{Z}_p \to \mathbb{Z}_p$ such that $h_{a,b}(x) = ax + b \pmod{p}$.
- This makes $\mathcal{H} = \{h_{a,b}\}_{a,b \in \mathbb{Z}_p}$ is pairwise independent. Note also that there are p^2 hash functions, so $\mathcal{H}$ takes up $O(p)$ space.

  This is very close to what you'd do with the streaming algorithm, but not exactly.

  Proof of Independence (simplified): Let $x, y, i, j \in \mathbb{Z}_p$ such that $x \neq y$. We want to show that

  $$\Pr_{a,b}[ax + b = i \text{ and } ay + b = j] = \frac{1}{p^2}$$

  We will do it for $x = 0, y = 1$ and the general case is left as an exercise. In this case, we'd want to prove:

  $$\Pr_{a,b}[b = i \text{ and } a + b = j] = \frac{1}{p^2}$$

  But given this construction, we know that b and $a+b$ is a random pair in $\mathbb{Z}_p^2$, so the probability is indeed $\frac{1}{p^2}$. Note that this is not 3-wise independent: given $x = 0, y = 1, z = 2$, then $h(z) = 2a + b = 2(a + b) - b = 2h(1) - h(0)$. Since there's a way to construct $h(z)$, this is not independent.

24.2 Modified Distinct Elements

- Instead of throwing our value into a hash function, we will pick a pairwise independent hash function $h : \{1, \dots, N\} \to [0, 1]$.
- We compute the t-th smallest value of $\{h(s_1), \dots, h(s_L)\}$, which would be the t-th smallest value of $\{r_1, \dots, r_k\}$.
- Then, we output t/α.

- The reason we want to use the t-th smallest is because want our algorithm to be tolerant to outliers.

  Why not just use the $N/2$-th smallest? Shouldn't this be the most tolerant value to outliers?

  It is, you can repeat the same analysis for exceedingly large values.

24.2.1 Analysis

- We have the property that (based on our algorithm):

$$\Pr[\text{alg outputs} \geq 2k] = \Pr\left[\alpha \leq \frac{t}{2k}\right] = \Pr\left[\sum_i C(i) \geq t\right]$$

  How do we go from the 2nd to the 3rd step?

  $C(i)$ is defined to be the number of outliers, so the probability that we output something larger than $2k$ is if there are more than t outliers.

  Here, we define $C(i)$ as follows:

$$C(i) = \begin{cases} 1 & r_i \leq \frac{t}{2k} \\ 0 & \text{otherwise} \end{cases}$$

  In other words, $C(i)$ is an indicator that counts the number of outliers. Then, we have:

$$E\left[\sum_i C(i)\right] = \sum_i E[C(i)] = \sum_i \Pr\left[r_i \leq \frac{t}{2k}\right] = \sum_i \frac{t}{2k} = \frac{t}{2}$$

  Where the second step is reached via linearity of expectation. This means that the expected number of outliers is $\frac{t}{2}$.

  Is this a good value?

- Now we compute the variance to get a good idea of the spread of $\sum_i C(i)$:

$$\text{Var}\left[\sum_i C(i)\right] = \sum_i \text{Var}[C(i)]$$

  Since each $C(i)$ is associated with an r_i which is derived from $h(s_i)$, each $C(i)$ is also pairwise independent. This allows us to take the summation out of the variance. Then:

$$\text{Var}[C(i)] \leq E[C(i)^2] = E[C(i)] = \frac{t}{2k} \implies \text{Var}\left[\sum_i C(i)\right] \leq k \cdot \frac{t}{2k} = \frac{t}{2}$$

25 Randomized Algorithms

- Described as an algorithm which uses random bits to solve problems. Generally, this is done using some function called RandomInt(a, b) which returns a random integer from the interval (a, b).
- Generally, algorithms will output a *pseudorandom* number (random enough for our purposes) But when we model randomness mathematically, we refer to a genuinely random number.
- Here, algorithms will be allowed to fail with some probability (say 5%).
- Oftentimes, randomized algorithms lead to much more elegant solutions when compared to deterministic algorithms. They can also sometimes be much faster than deterministic ones!

25.1 Background: Integer Factorization

- This is not a problem we know how to solve using a randomized algorithm. Given a number N, we want to find its prime factorization $N = p_1 p_2 \cdots p_k$.
- Naively, we could just test every single number from 1 to $\sqrt{N}$, but this takes $O(\sqrt{N})$ time (if $N \sim 10^{500}$, then our algorithm would take longer than the number of atoms in the universe to compute.)
- The best algorithm we know today is called a *General Number Field Sieve*, which factors an n bit number in time $C^{n^{1/3} \log(n)^{2/3}}$. (This C is not very large, but still not small.)
- This is a very important problem! RSA (the company) literally puts out numbers on the internet with cash prizes attached to them if you can factor them. RSA-250 is a 250-digit number, which was recently factored in 2020. RSA-290 has a $75,000 cash prize attached to it.
- *Aside:* this is a problem that is easily solved by quantum computers.

25.2 Primality Testing

- Given a number N, our only goal is to figure out whether N is prime or composite.
- This is very similar to the prime factorization problem! We *could* take N and factor it, but this is obviously very hard. But we can solve this relatively efficiently using randomness!
- We use another property of prime numbers: Fermat's little theorem! It says that if N is prime, then

  $$a^{N-1} \equiv 1 \pmod{N}$$

  for all $a \in \{1, 2, \ldots, N-1\}$.
- So we define a test called FermatTest(N). It picks a value $a \in \{1, 2, \ldots, N-1\}$ uniformly at random. If $a^N \equiv 1 \pmod{N}$, then we output "prime", otherwise we output "composite".
  - This algorithm will always output "prime" for prime N. For values that are coprime with N, it's harder to see whether they pass FermatTest, and ones that aren't are certainly going to fail FermatTest.
  - Becuase this test relies pretty heavily on the fact that a is coprime, our test is only as good as the number of coprime a that we get.
  - *Aside:* There do in fact exist are *composite* numbers that satisfy $a^{N-1} \equiv 1 \pmod{N}$ for all a coprime to N! In other words, these numbers will always pass FermatTest. These are called *Carmichael numbers*, but for the purposes of this class, we will pretend like these don't exist.
- **Theorem:** Suppose N is composite and not carmichael. Then, we have $P(\text{FermatTest}(N) = \text{composite}) \geq 1/2$.

  Proof: If the number is not Carmichael, this implies the existence of some coprime b such that $b^{N-1} \not\equiv 1 \pmod{N}$, and thus there is a single b that fails FermatTest.

  Subclaim: Suppose some value a passes FermatTest. Then, $ab \pmod{N}$ will fail FermatTest.

  Proof: This is because ab is no longer coprime to N, so this gives us a 1-1 correspondence between a value that passes FermatTest and one that doesn't. More rigorously, we have:

  $$\begin{aligned}(ab)^{N-1} &\equiv a^{N-1} \cdot b^{N-1} \pmod{N}\\ &\equiv b^{N-1} \pmod{N}\\ &\not\equiv 1 \pmod{N}\end{aligned}$$

where we use the fact that $a^{N-1} \equiv 1 \pmod{N}$. Then, the one-to-one correspondence follows. Then, since there are values that *just fail* FermatTest, then this means that there are more values that fail than pass. Hence, $P(\text{fail}) \geq 1/2$, as desired.

- Now, let's repeat this algorithm k times. On any given test, we know that if FermatTest returns "composite" then N is definitely composite, so the only case where we get a fail is if N is composite but FermatTest outputs "prime". This occurs with probability

$$P(N \text{ composite but outputs prime}) = 1 - \frac{1}{2^k}$$

  This means that when our test outputs "prime", there is a $1 - \frac{1}{2^k}$ chance of it being a false positive.
- We can add another check to detect Carmichael numbers – we won't really delve into this here. This gives the Miller-Rabin primality test, developed in 1976.
- Since then, there have been improvements: the AKS Primality test (2002) gave a deterministic polynomial time $O(n^{12})$ algorithm for primality testing.

25.3 Randomized Complexity Classes

- Some problems have both efficient randomized and deterministic algorithms, but there are others that only have efficient randomized algorithms. (e.g. polynomial identity testing)
- This implies the existence of two possible worlds:
  1. Every efficient randomized algorithm has a corresponding deterministic solution. (P)
  2. Some problems only have efficient randomized algorithms. (BPP)
- The question of which world we live in is called the P vs. BPP problem. P is the class of efficiently deterministic problems (same P as before), and BPP is the class of efficient, randomized algorithms. We think that these complexity classes are equal.

25.4 Minimum Cut

- Consider a (unweighted, undirected) graph $G = (V, E)$, and we want to find the minimum cut of this graph. Here, the heuristic we're using to quantify the size of the cut is the number of edges it crosses through.
- Recall from our max-flow problem, we found the minimum $s-t$ cut via our flow algorithm. It turns out that we can use the flow algorithm to solve this deterministically via a reduction, but today we'll talk about another algorithm: Karger's algorithm.
- For every $i = 1, \ldots, n-1$, it will:
  1) Pick a uniformly random edge e, and "contract" the edge.
  2) It will return the cut specified by the remaining two supervertices.
- The way to contract an edge is to combine the two vertices connected by e, and combine them into one supervertex. It will preserve all the possible edges, which means that the graph doesn't have to remain simple in this process. Then, once only two vertices are remaining, then that's the cut that we output. Notice that because we don't delete any edges in this process, then the size of the "supercut" is equal to the size of the cut in the original graph.
- This won't return the minimum cut, but it will return the minimum cut with fairly good probability.

25.4.1 Intuition

- So suppose we have a graph with a bunch of edges, then with one singular edge outside, like this:
- If Karger's algorithm is to find the minimum cut, then it should never contract edge e. Basically, it leverages the fact that becuase there are more edges outside of the minimum cut, then the algorithm shouldn't select the edges along the minimum cut.
- **Theorem:** Let $C = (S, \overline{S})$ be a minimum cut of size k. Then, the probability that Karger's algorithm outputs the cut C is given by:

  $$P(\text{Karger's outputs } C) \geq \frac{1}{\binom{n}{2}} = \frac{2}{n(n-1)}$$

  While this is indeed bad for a single iteration, it just means that in expectation we just need to repeat this n^2 times to get the minimum cut!
- *Proof:* First, we state four facts:
  - Let G_i be the state of the graph G at the *start of the i-th* iteration. Then, because C is the minimum cut with size k, then we know that the minimum cut of $G_i \geq k$.
  - Further, the number of vertices in G_i is $n - (i - 1) = n - i + 1$.
  - The degree of each vertex in $G_i \geq k$, because of the fact that the minimum cut is k. Had the degree been less than k, then the minimum cut would be less than k, since the min-cut would just be to cut that vertex with degree less than k.
  - The number of edges in G can be calculated as:

    $$\begin{aligned} N &= \frac{1}{2} \sum_{v \in G_i} \deg(v) \\ &\geq \frac{1}{2} \sum_{v \in G_i} k \\ &= \frac{1}{2} k |G_i| \\ &= \frac{1}{2} k(n - i + 1) \end{aligned}$$

  Now we complete the proof. Essentially, we want to compute the probability that the algorithm never contracts a cut in C. Suppose at step G_i, we haven't contracted any edges in C. Then, the probability that we don't contract an edge in C is:

  $$P(\text{don't contract an edge in } C = 1 - P(\text{contract an edge in } C) = 1 - \frac{k}{\frac{1}{2}k(n - i + 1)}$$

  Thus, the probability that we never contract an edge in C is the probability that this event occurs on every iteration i, at the step G_{i-k} :

  $$P(\text{never contract edge in } C) \geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \cdots = \frac{2}{n(n-1)}$$

  So therefore, the probability that we succeed is lower bounded by $\frac{2}{n^2}$.

26 Quantum Algorithms

26.1 Brief History

- In the world of physics, there was a problem: we couldn't simulate quantum systems. What does that even mean?
- Suppose we have n electrons, each with spin $|\uparrow\rangle$ or $|\downarrow\rangle$. This gives us a total of 2^n possible configurations. The fastest algorithm known at the time was one that took $O(2^n)$ time.
- Richard Feynman suggested in a 1981 talk that we could potentially build a computer out of electrons, and instead of computing the quantum mechanics itself, we just let the electrons simulate themselves.
- How powerful are quantum computers actually? Very powerful! A small list of algorithms that are very powerful:
  - Deutsch-Jozsa algorithm: solves the Deutsch-Josza problem
  - Bernstein-Vazirani algorithm
  - Simon's algorithm
  - Shor's algorithm
  - Grover's algorithm

26.2 Deutsch-Josza Problem

- Given an input function $f : \{0,1\}^n \to [0,1]$, the function is one of the following:
  i) $f(x) = 0$ for all x
  ii) $f(x) = 1$ for all x
  iii) $f(x) = 0$ for half of x, and $f(x) = 1$ for the other half.
- We proved that classical deterministic algorithms have to check at least $2^{n-1} + 1$ values for $f(x)$ to determine. This means that the algorithm runs in exponential time. However, the quantum algorithm can do this process in $O(n)$ time!
- There is also a very simple randomized algorithm for this: we can pick a subset of x at random, and this allows us to determine the identity of $f(x)$ also fairly quickly.
- So this problem tells us the power of quantum computers over deterministic algorithms, but doesn't show the power of quantum algorithms over random algorithms.

26.3 Bernstein-Vazirani Problem

- Given an input function $f : \{0,1\}^n \to \{0,1\}$ that takes a subset of x and adds the values up modulo 2, we want to figure out which bits were added in the sum.
- Classically, we can look at all the possible configurations for f, which would require $n+1$ looks at f. However, the quantum algorithm only needs to look at f once!

26.4 Shor's Algorithm

- We know that if we have an n-digit number, the best classical algorithm factorizes a number in $O(e^{1.9n^{1/3} \cdot \log(n)^{2/3}})$, but Shor showed that we can factor numbers in $O(n^2)$ time on a quantum computer.

- This is significant, because RSA encryption relies factoring large numbers and this algorithm basically gives us a way to break that encryption scheme.

26.5 Grover’s Algorithm

- This algorithm solves circuit-SAT in $O(\sqrt{2}^n)$ time on a quantum computer, wherea s a classical algorithm takes $O(2^n)$ time. This also solves many other problems (just due to their relationship to circuit-SAT).

26.6 Speedups

- There are three main types of speedups:
  - Shor-Type: these are exponentially faster than classical algorithms, but only work on a very restricted subset of problems.
  - Grover-Type: This speeds up things polynomially, but with the benefit that they work on a large number of problems.
  - Physics Simulation: Exponentially faster, but can only work for physics problems.
- One of the main reasons why quantum computers outperform classical ones is mainly because they can compute very fast Fourier transforms, which are used everywhere in computation.
- However, quantum computers are not able to solve NP-complete problems efficiently.