Contents

1 Algorithms of Arithmetic

1.1 Lecture 1

1.1.1 Addition/Multiplication

First, let us consider the problem of adding two n-bit numbers, a and b. If both are a different amount of bits from each other, we can pad 0's to the left of the smaller one until it reaches the length of the larger one (note that padding 0's doesn't change the sum). The grade-school algorithm (add column-wise and do carries) has to compute at most $n + 1$ additions, which we can assume are all constant-time. Thus, we say addition has complexity linear in n, or $\Theta(n)$ (we drop the constant because 1 pales in significance to how great n can grow).

But can we do better for addition? The answer is no; since it takes $n + 1$ bits to write our answer, any algorithm returning the sum must require $n + 1$ operations. Thus, we say as a lower bound, addition is $\Omega(n)$ (the full definitions of these terms will come shortly).

Now we turn to the problem of multiplication. How fast is the grade school algorithm? First, we multiply digit-wise and then do a bunch of additions. In binary, multiplying by a 0 or 1 and then right-padding with 0s (bitshifting) corresponds to a constant time operation for each bit of b. Then, we have to add together n potentially $2n$ bit numbers. This adds n^2 time. Thus, the runtime is quadratic in n or $\Theta(n^2)$.

Now, can we do better for multiplication? It turns out we can. We do this by leveraging the idea of divide-and-conquer; breaking our problem into smaller subproblems that we can solve recursively and then using these pieces to build the final solution.

Here is our first attempt:

Algorithm 1.1 (Naive Divide and Conquer)
Suppose the numbers x and y that we wish to multiply are represented in decimal. Assume without loss of generality that both have the same number of digits n, and n is a power of two (if they aren't then we can just pad with 0s on the left until these requirements are met). Then, let x_H be the upper half of the digits of x and x_L be the lower half of the digits of x (and define y_H, y_L analogously). Note that

$$x = x_H \cdot 10^{n/2} + x_L$$
$$y = y_H \cdot 10^{n/2} + y_L$$

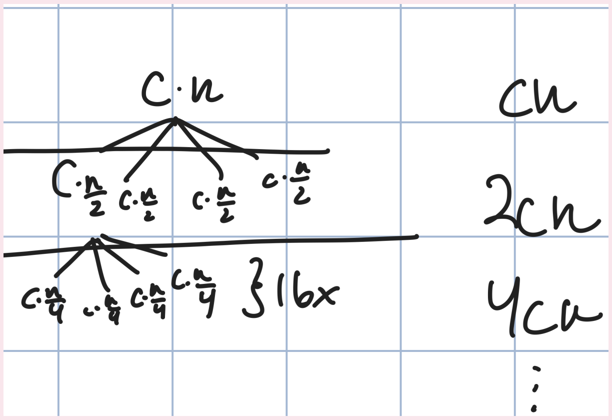
And all the subscripted letters are $n/2$ digits long. Multiplying these together then yields:

$$x \cdot y = x_H \cdot y_H \cdot 10^n + (x_H \cdot y_L + x_L \cdot y_H)10^{n/2} + x_L \cdot y_L$$

Note that to do this multiplication, we have to do 3 n-digit additions as well as 4 $n/2$-digit multiplications (Multiplying by powers of 10 is just digit shifts, which can be done in linear time). To do the additions, let us use the standard grade school algorithm. To do the multiplications, let us have the function call itself recursively on the $n/2$-digit numbers. As a base case, note that if x and y are both one-digit numbers, the multiplication can be looked up in constant time.
Runtime Analysis Now, to analyze the runtime, let the amount of time taken during the base case be c' and let c be another constant. Then, for the total running time $T(n)$, we know that:

$$T(n) = \begin{cases} 4T(n/2) + cn & n > 1 \\ c' & n = 1 \end{cases}$$

Since we make 4 recursive calls on an input of $n/2$ digits, and do a linear amount of work in every call. To analyze the runtime, we should draw a recursive tree.



Summing the nodes in this tree will give us the total runtime. As we can see from the diagram, on layer i (considering the top layer as layer 0), there is $c \cdot n/2^i$ work done at each node, and there are 2^{i+1} nodes in that layer. Call $k = \log_2 n$. Then our total runtime is:

$$
\begin{aligned}
T(n) &= cn + 2cn + 4cn + \cdots + 2^k cn \\
&= cn\frac{2^{k+1} - 1}{2 - 1} \\
&\leq 2cn \cdot 2^k = 2cn^2 = \Theta(n^2)
\end{aligned}
$$

Thus, this algorithm is no better than our traditional grade-school algorithm.

However, there is a way to make this work. Gauss came up with a way to multiply two complex numbers (analogous to this setup) in 3 multiplications instead of 4. Similarly, Russian mathematician Karatsuba applied it to integer multiplication.

Algorithm 1.2 (Karatsuba's Algorithm for Multiplication)
Proceed similarly to last time, defining x_H, y_H, x_L, y_L. However, define the following:

$$A = x_H \cdot y_H, B = x_L \cdot y_L, D = (x_L + x_H) \cdot (y_L + y_H)$$

Then notice that the middle term from before is just $D - A - B$, i.e. we can write:

$$x \cdot y = A \cdot 10^n + (D - A - B) \cdot 10^{n/2} + B$$

This means that we only need to compute three multiplications recursively (and do a few more additions, but since those take linear time, it's fine to have extra of those).
Runtime Analysis The runtime analysis is similar to that of the naive algorithm. However, instead of the amount of work increasing by $4/2 = 2$ every level, instead the work will only increase by $3/2$ every time (3

subcalls of size $n/2$). Let us again define $k = log_2 n$. This gives us total runtime, $T(n)$, being:

$$\begin{aligned}
T(n) &= cn + \frac{3cn}{2} + \frac{9cn}{2} + \cdots + \frac{3^k cn}{2^k} \\
&= cn\frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\frac{3}{2} - 1} \\
&\leq 2 \cdot 32cn\frac{3}{2}^k \\
&= 3c3^k \\
&= 3c\left(2^k\right)^{\log_2 3} \\
&= 3cn^{\log_2 3} \\
&= \mathcal{O}(n^{\log_2 3})
\end{aligned}$$

Note that $\log_2 3 \approx 1.585 < 2$, so this is better than our naive algorithm!

After Karatsuba's algorithm, people have continued to improve the worst case runtime of integer multiplication. In 2019, a $\mathcal{O}(n \log n)$ algorithm was found for multiplying two numbers together. However, this series of algorithms after Karatsuba's had such large constant factors that in practice, grade-school multiplication is still the one implemented most of the time.

As a note, Python currently uses Karatsuba multiplication, but only for numbers that are sufficiently large.

2 Divide and Conquer Algorithms

2.1 Lecture 2

Before we introduce some topics, let us set the stage for our analysis on the next algorithm. We define "Flops" as floating-point operations (additions, subtractions, multiplications, divisions, etc.). Now, we will consider the amount of flops that it takes an algorithm to run. We will use this to measure the runtime of certain algorithms.

2.1.1 Fibonacci Numbers

Consider the problem of computing the nth Fibonacci number. Recall the Fibonacci sequence is defined by the recurrence:

$$F_0 = 0, F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

The simplest algorithm one can do is the follow the recurrence word-for-word.

TODO: Finish this section (Iteration, Fast Powering, Closed Form Solution)

2.1.2 Asymptotic Notation

Consider two functions $f, g : \mathbb{Z}^+ \to \mathbb{Z}^+$. Here is some information about common asymptotic notation used to analyze the size of these functions (these functions can maybe represent the runtime of an algorithm).

Name	Notation	Meaning	Analogy
"Big-Oh"	$f = \mathcal{O}(g)$	$\exists c > 0$ s.t. $f(n) \leq cg(n)$	$\leq$
"Little-Oh"	$f = o(g)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)}$	$<$
"Big-Omega"	$f = \Omega(g)$	$g = \mathcal{O}(f)$	$\geq$
"Little-Omega"	$f = \omega(g)$	$g = o(f)$	$>$
"Theta"	$f = \Theta(g)$	$f = \mathcal{O}(g)$ and $f = \Omega(g)$	$=$

Here is an example to get a feel for how asymptotic notation works.

Example 2.1
Take $f(n) = 3n^3$ and $g(n) = n^4$. Then, notice,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n^3}{n^4} = 0$$

So, $f = o(g)$. We can also conclude more. Realize that the above limit really means that there exists an N such that for all $n \geq N$, we have:

$$\frac{f(n)}{g(n)} \leq 1 \implies f(n) \leq 1 \cdot g(n)$$

(Note that 1 is not important for this argument; we could've chosen any $\varepsilon > 0$). Now, consider the values of $\frac{f(n)}{g(n)}$ for $n < N$; this has some maximum c. Thus, we can conclude that for ALL n,

$$f(n) \leq \max(c, 1)g(n)$$

which implies $f = \mathcal{O}(g)$.

2.2 Lecture 3

2.2.1 Recurrences and Master Theorem

The idea of divide-and-conquer algorithms are to divide the input inot smaller parts, recurse on parts, and combine the parts to build an answer.

To analyze the runtime of divide-and-conquer algorithms, it is useful to derive the following result.

Theorem 2.1 (Master Theorem)
Suppose we have a recurrence

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

then, we have

$$T(n) = \begin{cases} \Theta\left(n^d\right) & a < b^d \\ \Theta\left(n^d \log n\right) & a = b^d \\ \Theta\left(n^{\log_b a}\right) & a > b^d \end{cases}$$

Master theorem can be shown by drawing a recursion tree, and then summing up the work done in each level (proof omitted for brevity). We can also think of the cases as symbolizing the following:

Case	Interpretation
$a < b^d$	The root does most of the work
$a = b^d$	The root and the leaves do an equal amount of work
$a > b^d$	The leaves do most of the work

2.2.2 Matrix Multiplication

Another example of a divide and conquer algorithm is matrix multiplication.

Consider multiplying two n-by-n matrices A and B.

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nn} \end{bmatrix}$$

Then the resultant C has entries given by:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

The natural implementation is then to to loop this summation over i and j. This means this will be three nested loop (a loop is needed for the summation). In flops, this runs in $\Theta(n^3)$ operations.

We can try to break our input instead into $n/2$-by-$n/2$ blocks, as shown:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

To find the runtime of this algorithm, let us realize there are 8 multiplications (recursively) and then finally a $\Theta(n^2)$ addition at the end. This means our recurrence is:

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

Note that for our Master theorem setup, $8 > 2^2$, so we have

$$T(n) = \Theta\left(n^{\log_2 8}\right) = \Theta\left(n^3\right)$$

so this is no better than our naive approach.

Using a similar realization to Karatsuba, Strassen in 1969 found the following:

Algorithm 2.1 (Strassen's Algorithm)
Consider two matrices X and Y which are both n-by-n. Break them up into block matrix form of $n/2$-by-$n/2$ matrices as follows:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Define the following:

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ &\vdots \\ P_7 &= (A - C)(E + F) \end{aligned}$$

Then,

$$Z = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Analyzing the runtime, we see that there are 7 $n/2$-by-$n/2$ multiplications, and some n^2 additions, meaning the total runtime is

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

which Master theorem brings to

$$T(n) = \mathcal{O}\left(n^{\log_2 7}\right) \approx \mathcal{O}\left(n^{2.81}\right)$$

However, Strassen has such a big constant factor that the normal n^3 algorithm is still the most widely used.

2.2.3 Sorting

Consider the problem of sorting a length n array A.

Algorithm 2.2 (MergeSort)
First, we define a procedure Merge that takes two sorted lists and merges them in linear time. To do this, we first keep a pointer on both lists that starts at the beginning of each list. We then compare the pointed-to elements of each list. The lesser element is then added to the output, and the pointer of the list with that element is incremented by one place. This keeps going until all the elements are used. We then use merge to divide-and-conquer the list as follows:

```
function MergeSort($A[1 \ldots n]$)
    if $n = 1$ then
```

```
        return $A$
    $B \leftarrow$ MergeSort($A[1 \dots \frac{n}{2}]$)
    $C \leftarrow$ MergeSort($A[\frac{n}{2} + 1 \dots n]$)
    return Merge($B, C$)
```

We get the following recurrence for MergeSort:

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

which through master theorem gives us a running time of

$$T(n) = \Theta(n \log n)$$

Another way of implementing this is a bottom up approach:

Algorithm 2.3 (Iterative Merge Sort)

```
function MergeSortIter($A[1 \dots n]$)
    $Q \leftarrow$ Divide $A$ into $n$ lists of size one
    while $Q$.size() > 1 do
        $X, Y \leftarrow Q$.pop(), $Q$.pop()
        $Q$.push(Merge($X, Y$))
    return $Q$.pop()
```

Runtime Analysis Now we can think about the runtime of this algorithm. Think of the algorithm as running in phases. Phase 0 is when lists popped have size 1. Phase 1 is when lists popped have size 2. Phase i is when lists popped have size 2^i. Note that in each phase, each element is looked at exactly once. Thus, the total runtime must be proportional to $n \cdot$ number of phases. How many phases are there? There are $\log n$ phases, giving us the same $\Theta(n \log n)$ runtime!

The best way to see this is with an example:

Example 2.2

Suppose our initial list is:

$$A = \begin{bmatrix} 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

We then split this into sublists of size one in our Q and start iterating:

$$\begin{aligned} Q &= [[8], [7], [6], [5], [4], [3], [2], [1]] \\ Q &= [[6], [5], [4], [3], [2], [1], [7, 8]] \\ &\vdots \\ Q &= [[7, 8], [5, 6], [3, 4], [1, 2]] \\ Q &= [[3, 4], [1, 2], [5, 6, 7, 8]] \\ Q &= [[1, 2, 3, 4], [5, 6, 7, 8]] \\ Q &= [[1, 2, 3, 4, 5, 6, 7, 8]] \end{aligned}$$

Our list has been sorted!

Can we sort faster than $n \log n$? It turns out we cannot do any better with an algorithm that uses comparisons to sort (the problem is $\Omega(n \log n)$).

Theorem 2.2 (Fastest Shorting in Comparison Model, Lower Bound)
We will show that $\Omega(n \log n)$ comparisons are needed even if promised A is some permutation of $\{1, \ldots, n\}$ (all distinct as well).

Proof
The first comparison such an algorithm might make might be: is $A_i < A_j$? Then we branch off into two cases for each, where we require another comparison. We can construct a binary tree that models the situation.
First, notice that each leaf is when the algorithm terminates. Note that since every run of the algorithm with a different input produces a different output permutation of the input, there must be at least $n!$ leaves. Consider the maximum depth of this tree T. There are at most 2^T leaves, meaning that $2^T \geq n!$. This means

$$\begin{aligned} T &\geq \log(n!) \\ T &= \Omega(n \log n) \end{aligned}$$

The last claim can be shown by realizing

$$\begin{aligned} n! &\geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \\ \log(n!) &\geq \frac{n}{2} \log\left(\frac{n}{2}\right) \\ \log(n!) &\geq \frac{n}{2} \log n - \frac{n}{2} \\ \log(n!) &= \Omega(n \log n) \end{aligned}$$

However, there are way more operations than comparisons that can be used for sorting. An example of this is counting sort: if you have n integers and all the integers have values between 1 and b, then you can sort in $\Theta(n + b)$, by just keeping an array of all the elements that fit in each value "bucket" between 1 and b.

The Word RAM model: suppose your machine can store words of size w and you can do any common C operations. The fastest known algorithm following this model (not just comparisons) is: $\mathcal{O}\left(n\sqrt{\log \log n}\right)$. This is also a randomized algorithm.

There are two types of randomized algorithms (which will be revisited). A Monte Carlo randomized algorithm is one whose output may be incorrect with small probability. A Las Vegas algorithm is one whose runtime is fast in expectation, but may be slow with small probability.

2.2.4 Selection/Medians

We now consider the problem of selection. Suppose we want to select the kth smallest integer in a list A. Without loss of generality, assume all elements of A are distinct (since we could replace $A[i]$ with the tuple $(A[i], i)$). For selection, there is Quick Select, which we will explore later. Notably, quick select requires knowing the median in linear time. We will instead focus on that problem, (the same as the selection problem for $k = n/2$).

Algorithm 2.4 (Median of Medians)
Take an array A. Then break up the array into subarrays of size 5. Next, we will recursively compute the median of each subarray. Note that this takes constant time to complete since 5 is a constant. Now we have a $N/5$ size array. We then find the median recursively of this smaller problem. Call this median m_1.
Now, we change the array such that all the elements bigger than m_1 end up on the right of m_1, elements smaller than m_1 end up on the left, and m_1 is in the middle of these two parts (this only requires a linear scan). If m_1 is in position less than $n/2$, then the true median sits on its right, so we can recurse on the right half. If m_1 is in position greater than $n/2$, then the true median sits on its left, so we can recurse on the left half. Finally, if m_1 is at exactly position, $n/2$, then we have found the median.
Runtime Analysis The claim is that at least 30% of the elements are filtered out by comparisons to m_1. To

show this, consider m_1 compared to the other medians. Note that it is bigger than 3 of the elements in every median it is bigger than, so, it is bigger than $\frac{3}{5} \cdot \frac{N}{10} = \frac{3}{10}$ of the elements. Thus, if m_1 is in the first half of the array, then it will filter out at least $\frac{3}{10}$ of the numbers. Similarly, you can make a symmetric argument that if m_1 is in the second half, then it must be less than $\frac{3}{10}$ of the numbers and thus will filter out 30% of them. Either way, we can then produce the following recurrrence:

$$T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + \Theta(n)$$

This recurrence gives $T(n) = \mathcal{O}(n)$. We can give an inductive argument:

Proof
We will show that $T(n) \leq Bn$ for sufficiently large B, which will imply our big-$\mathcal{O}$ runtime.
Base Case: if n is 1, then we just return the input, so if B is greater than the time needed to return then the base case holds.
Inductive Hypothesis: Suppose that the claim holds for $k < n$.
Inductive Step: By the recurrence and the inductive hypothesis, we have that

$$T(n) \leq B\frac{7n}{10} + B\frac{n}{5} + Cn$$

where C is some other constant. Now, we have

$$\begin{aligned} T(n) &\leq \left(\left(\frac{7}{10} + \frac{1}{5}\right)B + C\right)n \\ &\leq \left(\frac{9}{10}B + C\right)n \\ &\leq Bn \end{aligned}$$

as long as $C \leq \frac{B}{10}$. Since C is fixed, we can set $B \geq 10C$ to make this true. ■

2.3 Lecture 4

2.3.1 Polynomial Multiplication

Suppose we have two polynomials as inputs:

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

$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{d-1} x^{d-1}$$

Then we want the output polynomial C in the following form:

$$C(x) = c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2}$$

Define $N = 2d - 1$ for simplicity, and notice that these can all be considered $N - 1$ degree polynomials if we pad A and B with 0 coefficients on higher order terms.

There is a relationship between polynomial and integer multiplication. Given integers α, β, if we want $\gamma = \alpha \times \beta$, we first write them digit-wise as

$$\alpha = \alpha_{N-1}\alpha_{N-2}\ldots\alpha_0$$

$$\beta = \beta_{N-1}\beta_{N-2}\ldots\beta_0$$

$$A(x) = \alpha_0 + \alpha_1 x + \cdots + \alpha_{N-1} x^{N-1}$$

$$B(x) = \beta_0 + \beta_1 x + \cdots + \beta_{N-1} x^{N-1}$$

Note that $\alpha = A(10)$ and $\beta = B(10)$, and $\gamma = (A \cdot B)(10)$ plugging in integers for the polynomials is fairly fast (just some additions and multiplication). This shows that integer multiplication and polynomial multiplication are fairly connected.

Algorithm 2.5 ("Straightforward" Algorithm for Polynomial Multiplication)

$$C(x) = c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2}$$

What are these coefficients in terms of a_i and b_i?

$$\begin{aligned} c_0 &= a_0 b_0 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ &\vdots \\ c_k &= \sum_{j=0}^{k} a_j b_{k-j} \end{aligned}$$

Then the algorithm looks something like this:

- Loop over $k = 0$ to $N - 1$
  - Compute c_k with a loop from $j = 0$ to k

Note that this algorithm runs $\Theta(N^2)$.

However, we can do better, since integer multiplication is close to polynomial multiplication.

Algorithm 2.6 (Karatsuba for Polynomials)
Call:

$$\begin{aligned} A_l(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_{N/2-1}x^{N/2-1} \\ A_h(x) &= a_{N/2}x^{N/2} + \cdots + a_{N-1}x^{N-1} \\ B_l(x) &= b_0 + b_1x + b_2x^2 + \cdots + b_{N/2-1}x^{N/2-1} \\ A_h(x) &= b_{N/2}x^{N/2} + \cdots + b_{N-1}x^{N-1} \end{aligned}$$

Note that $A(x) = A_l(x) + x^{N/2}A_h(x)$ and $B(x) = B_l(x) + x^{N/2}B_h(x)$. Using the Karatsuba trick, you see that you need 3 multiplications, giving the recurrence:

$$T(N) \leq 3T\left(\frac{3}{2}\right) + \Theta(N)$$

which solves to: $T(N) = \Theta(N^{\log_2 3})$

Here is a fact from elementary algebra:

Note 2.1 (Polynomial Interpolation)
A degree $< N$ polynomial is fully determined by its evaluation on N distinct points.

Proof
Represent polynomial $C = c_0 + c_1x + \cdots + c_{N-1}x^{N-1}$ as the vector:

$$c = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix}$$

Suppose your points are represented as $(x_0, y_0), (x_1, y_1), \ldots, (x_{N-1}, y_{N-1})$. Then we want $y_i = C(x_i)$ for all i. This is equivalent to the following matrix vector product

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & \ldots & x_{N-1}^{N-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix}$$

I.e. we can call this $Vc = y$ and solving this equation is only possible if V is full rank.
However, a fact for the "Vandermonde" matrix V is that

$$\det(V) = \prod_{i<j}(x_i - x_j) \neq 0$$

so we can solve $c = V^{-1}y$, so since we chose distinct x_i, there is a unique polynomial that interpolates, represented by the vector c.

This gives rise to the following idea: rather than multiplying directly, we instead evaluate $C(x_0), C(x_1), \ldots, C(x_{N-1})$ for distinct x_i.

To do this evaluation, just evaluate $A(x_i)$ and $B(x_i)$, then finally combine to get $C(x_i)$. Finally, interpolate to set back coefficients from C in terms of these points.

However, interpolation is way too slow. You have to use inversion which takes $\mathcal{O}(n^3)$ flops. Instead, what if we choose V carefully such that it's faster to invert?

Let us establish some types:

1. The Discrete Fourier Transform (DFT) is a **matrix**.
2. The Fast Fourier Transform (FFT) is a **algorithm**.

Definition 2.1 (Discrete Fourier Transform (DFT))
Define $\omega = e^{2\pi\sqrt{-1}/N}$ (primitive root of unity). Now define the DFT matrix F such that $F_{ij} = (\omega^i)^j = \omega^{ij}$. Imagine evaluating a polynomial at points $1, \omega, \omega^2, \dots, \omega^{N-1}$
This gives the Vandermonde matrix:

$$V = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

i.e. the DFT matrix.

Here is the most important property of the DFT matrix:

Note 2.2 (Inverse of the DFT matrix)

$$F^{-1} = \frac{1}{N}\overline{F}$$

Note that we can view the following algorithm in two lenses: either a fast way to multiply by F^{-1}, or a fast way to calculate $P(\omega), P(\omega^2), \dots$. We take the latter interpretation.

Algorithm 2.7 (Fast Fourier Transform (FFT))
The goal of this algorithm is to multiply by the DFT quickly. We will take a polynomial interpretation. Consider our polynomial, assuming without loss of generality that N is a power of 2:

$$\begin{aligned} P(z) &= p_0 + p_1 z + p_2 z^2 + \dots + p_{N-1} z^{N-1} \\ &= \left(p_0 + p_2 z^2 + p_4 \left(z^2\right)^2 + \dots + p_{N-2}\left(z^2\right)^{N/2-1}\right) + z\left(p_1 + p_3 z^2 + p_5 \left(z^2\right)^2 + \dots\right) \\ &= P_{\text{even}}(z^2) + z P_{\text{odd}}(z^2) \end{aligned}$$

For all N roots of unity, squaring any of them will just give you another Nth root of unity. In fact, it'll give you a $N/2$ root of unity. This means that we only have to evaluate P_{odd} and P_{even} on $N/2$ roots of unity.
Runtime Analysis As we stated before, we only have to evaluate P_{odd} and P_{even} on $N/2$ roots, and they also have degree less than $N/2$, so we make two subcalls of size at most $N/2$. Furthermore, multiplying by z and adding two polynomials takes linear time. This means our recurrence becomes:

$$T(N) \le 2T\left(\frac{N}{2}\right) + \mathcal{O}(N)$$

which solves to:

$$T(N) = \mathcal{O}(N \log N)$$

Finally, we give an algorithm for our initial problem of multiplying polynomials.

Algorithm 2.8 (Polynomial Multiplication Via FFT)
First, we use the Fast Fourier Transpose to compute $\hat{a} = Fa$ and $\hat{b} = Fb$.
Then, for $i = 0$ to $N-1$, we compute, $\hat{c_i} = \hat{a_i} \times \hat{b_i}$.
Finally, we have to bring c back into the original basis, i.e. compute $c = F^{-1}\hat{c}$. To do this, notice

$$\begin{aligned} c &= \frac{1}{N}\overline{F}\hat{c} \\ &= \frac{1}{N}\overline{F\overline{\hat{c}}} \end{aligned}$$

which requires just one more use of FFT to multiply $F\overline{\hat{c}}$.
The runtime is thus dominated by the 3 FFTs, giving us: $\mathcal{O}(N\log N)$ runtime, assuming we can multiply/add/-conjugate complex numbers in constant time.

It may seem paradoxical that since it takes N^2 entries to write down F, how come we have come up with a faster way to multiply by F? The heart of this is that we **never explicitly write down** F. Instead, we just use FFT to multiply F by a vector (quickly).

2.3.2 Cross Correlation

Next we consider the problem of cross correlation. Suppose you have two vectors: x and y:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{bmatrix}, y = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

where $n \geq m$. The cross correlation is all shifted dot products of x with y, i.e.

$$\begin{aligned} &x_0y_0 + x_1y_1 + \cdots + x_{m-1}y_{m-1} \\ &x_0y_1 + x_2y_2 + \cdots + x_{m-1}y_m \\ &\vdots \end{aligned}$$

However, we can reduce cross correlation to a problem we have already solved, polynomial multiplication!

Algorithm 2.9 (Cross Correlation Via Polynomial Multiplication)
Define the following:

$$\begin{aligned} X(z) &= x_{m-1} + x_{m-2}z + \cdots + x_0z^{m-1} \\ Y(z) &= y_0 + y_1z + y_2z^2 + \cdots + y_{n-1}z^{n-1} \\ Q(z) &= (X \cdot Y)(z) = q_0 + q_1z + \ldots \end{aligned}$$

Then let us investigate the coefficients of Q:

$$\begin{aligned}
q_0 &= x_{m-1}y_0 \\
q_1 &= x_{m-1}y_1 + x_{m-2}y_0 \\
&\vdots \\
q_{m-1} &= x_{m-1}y_{m-1} + \cdots + x_1y_1 + x_0y_0 \\
q_{(m-1)+1} &= x_{m-1}y_m + \cdots + x_1y_2 + x_0y_1 \\
&\vdots
\end{aligned}$$

Thus, we can multiply the polynomials X and Y, then take the coefficients $m-1$ and bigger to get all of our cross correlation terms.

3 Graph Algorithms

3.1 Lecture 5

3.1.1 Graph Representation

Definition 3.1 (Graph)
A graph is a pair $G = (V, E)$ where V is the set of vertices and E is the set of edges. We generally call $n = |V|$ and $m = |E|$.
If G is a directed graph, then $E \subseteq V \times V$. G is simple if (a, a) (a self loop) is not allowed.
If G is an undirected graph, then E is a set of unordered pairs from V. If there are no self-loops, G is simple.

Example 3.1 (Examples of graphs)
Graphs are often used as convenient ways to represent data.

1. Road network where vertices are intersections, and edges are road segments connecting intersections. (Would be directed, also maybe "weighted" according to distance)
2. Social networks where vertices are people, and edges are friendships. (Facebook would undirected, Twitter/IG would be directed)

How do we represent graphs on a computer? We will assume: $V = \{1, \ldots, n\}$. Then to store edges, we will either:

a Adjacency Matrix: A is an n-by-n matrix where

$$A_{ij} = \begin{cases} 1 & (i, j) \in E \\ 0 & (i, j) \notin E \end{cases}$$

for a weighted graph this becomes:

$$A_{ij} = \begin{cases} w_{ij} & (i, j) \in E \\ \infty & (i, j) \notin E \end{cases}$$

b Adjaceny List: represent E as an array B of linked lists. Then, $B[i]$ is a linked list containing all j such that $(i, j) \in E$.

We can compare the cost of these representations as follows:

	Adjancency Matrix	Adjancency List
Space	n^2 bits	$\Theta(m + n)$ words
$(u, v) \in E$?	$\mathcal{O}(1)$	$\Theta(1 + d_u)$
Print all the neighbors of u	$\Theta(n)$	$\Theta(d_u)$

where d_u is the degree of u, i.e. $|\{w \mid (u, w) \in E\}|$.

Note that we could also choose alternative representations. Remember that graphs are abstractions used to store data, so there is no one-size-fits-all solution.

3.1.2 Depth First Search (DFS)

We discuss graph exploration, i.e. visiting all vertices in a graph.

Algorithm 3.1 (Depth First Search)

```
function DFS($V, E$)
    global clock = 1
    global visited = boolean[n]
    global preorder, postorder = int[n], int[n]
    for $v \in V$ do
        if visited[v] is false then explore($v$)
function EXPLORE($v$)
    visited[v] = true
    preorder[v] = clock
    clock = clock + 1
    for $(v, w) \in E$ do
        if visited $w$ is false then explore($w$)
    postorder[v] = clock
    clock = clock + 1
```

The time window between postorder[v] and preorder[v] is exactly the amount of time we spend in recursive calls from v.

We have a claim about the subroutine EXPLORE. Namely, EXPLORE(u) explores exactly: $\{v \mid \exists \text{ a path from } u \text{ to } v\}$, i.e. the connected component of u.

An argument for this claim is as follows:

Proof

We need to show two directions.

First, if we explored v, there must be a path from u to v, since every call in the recursion is from one neighbor to another. We can construct the path by just following the path in the recursion tree.

In the other direction, for the sake of contradiction, suppose there exists a reachable v that doesn't get explored. Let the path from u to v be:

$$(u \to x_1 \to x_2 \to \cdots \to x_r = v)$$

Let x_j be the first vertex on the path which isn't explored. Then, look at x_{j-1}, which was explored, which means we looped over all of x_{j-1}'s neighbors. This means we had the opportunity to visit x_j, but we didn't. This means we must've visited x_j and failed the if check. This is a contradiction, we visited x_j and didn't. Thus, we must have that all reachable v's are explored.

Now we look at the runtime of the routine. First, note that the outer for loop runs n times. Then, we have to enumerate the neighbors of u for all vertices u in the graph. We only have to do this once, since we visit each vertex only once.

This means that the total running time is:

$$T(n) = \Theta(n) + \sum_{u \in V} \text{time to enumerate neighbors}$$

which depends on our graph representation. With our adjacency matrix, the second term is quadratic. In the adjacency list, it is just the sum of (degrees + 1) which will be exactly $2m + n$. (Since by adding degrees we double count the number of edges).

	Adjancency Matrix	Adjancency List
DFS time	$\Theta(n^2)$	$\Theta(m + n)$

TODO: Add a pictoral example of DFS from lecture.

There are many applications of the DFS:

- Reachability - Identifying the separate connected components
- Articulation points - The set of vertices whose removal disconnects the graph

- Finding biconnected/triconnected components - if there are two/three disjoint paths between any two vertices
- Strongly connected components - In a directed graph, two vertices are strongly connected if there is a path from one vertex to another and from that other vertex back to the original
- Planarity testing - Testing if a graph is planar, i.e. you can draw it without crossings
- Isomorphism of planar graphs - Telling whether two planar graphs are isomorphic, i.e. we can turn one into the other with a bijection

The first set of problems we have already solved. Reachability is everything we visit in one iteration of the outer loop. The amount of connected components is the amount of explore calls in the outer loop.

Definition 3.2 (Preorder, Postorder)
The notation $\text{pre}(u)$ and $\text{post}(u)$ denote the preorder and postorder number of a given vertex u in a run of the DFS algorithm on some graph containing u.

Now let us explore another claim about the DFS.

Note 3.1
For some vertex u, the set of intervals $[\text{pre}(u), \text{post}(u)]$ are either pairwise nested or disjoint. The argument is that pairwise, either two are in the same connected component or different components. In the same component, one must have started before the other, then the recursion finished the inner one, then it must've ended. In different components, the intervals are disjoint.
Let us denote previsiting a vertex u as $[_u$ and postivisiting that vertex as $]_u$.

It helps us to classify the graph edges during a DFS.

Definition 3.3 (Types of Graph Edges)
These are the types of edges in a DFS traversal of G.

1. Tree edge: Traversed edge during DFS (in the DFS tree).
2. Forward edge: Goes from an ancestor to a descendant in the DFS tree, and is not a tree edge.
3. Back edge: Goes from a descendant to an ancestor in the DFS tree, and is not a tree edge.
4. Cross edge: An edge that is not any of the above.

The useful facts about these edges are as follows:

Theorem 3.1
Suppose $(u, v) \in E$. $\text{post}(u) < \text{post}(v)$ if and only if (u, v) is a back edge.

Proof
The backwards part of the claim is simpler. Basically u is a decendant of v, so the order is something like $[_v \ldots [_u$, so $]_u \ldots]_v$ must happen, i.e. the postorder numbers have the given order.

Theorem 3.2
G has a cycle if and only if it has a back edge.
We again take the backwards case first. By the definition of a back edge, there must be a series of tree edges from some vertex u to v and then a back edge from v to u. This implies there is a cycle from u to u.

Proof
Now the forward part of the claim, consider the cycle:

$$u_0 \to u_1 to \cdots \to u_k \to u_0$$

Take u_i with the lowest postorder number. Then the edge:

$$(u_i, u_{(i+1) \mod k})$$

goes from a higher postorder number to a lower postorder number, which means that by the previous result, this edge is a back edge, showing our claim.

3.2 Lecture 6

3.2.1 Topological Sort

We consider the problem of topological sort. We take as input a directed acyclic graph (DAG). We seek to find an ordering of V such that $(u, v) \in E \implies u$ must come before v in the ordering. Note that such an ordering of the graph is not necessarily unique. Intuitively, it represents the order that dependencies must be filled in order to solve a problem.

Algorithm 3.2 ("Brute-force" Search)
Try all permutations of the vertices. Return the first one that is a topological sort.
The runtime is $\mathcal{O}n!mn$, because for each of $n!$ permutations, we have to check m edges that the edge is respected, and finding each endpoint of the edge needs to be found in the sorted order.

To find a faster algorithm, let us define some terms.

Definition 3.4 (Source, Sink)
In a directed acyclic graph, a source is a vertex with no incoming edges. A sink is a vertex with no outgoing edges. Note that every DAG has a source and a sink.

Algorithm 3.3 (Source Peel-Off)
We can iteratively "peel-off" source vertices one at a time. It is possible to do this algorithm in linear time: $\mathcal{O}(m + n)$.
To do so, have n linked list nodes, one for each vertex. Then we keep an array B where the ith entry of the array stores a pointer to node i. Then, we also keep an array B of linked lists where the ith entry of that array is a doubly-linked list that has i incoming edges. To peel off a source, you take anything in the 0th entry of the array and take it out, call it u. Then, for each outgoing edge (u, v), move v to the lower bucket in the array (since it now has one less incoming edge). Rinse and repeat.
This is a bit too complicated. There is an easier linear-time algorithm.

Consider the following pre and postorder traversals (subscripts dropped for clarity):

$$[[][]][][[]]$$

then, it's easier to see that the vertex associated with last closing bracket is a source; it has nowhere else that lead to it. This leads to the following result:

Theorem 3.3 (Finding a Source with DFS)
The vertex v with the largest postorder number must be a source.

Proof
Suppose for the sake of contradiction that v is not a source. That means there must exist a $u \in V$ such that $(u, v) \in E$.
We know that the only possibilities are that $[_u]_u[_v]_v$ or $[_v[_u]_u]_v$. The latter means that (u, v) is a back edge, which would mean there is a cycle. However there are no cycles since this is a DAG, so this cannot be the case. The former means that u and v are disconnected, but there is clearly an edge between them or that v was already visited, which is also not the case.
By exhaustion of cases, we have a contradiction, so v must be a source. ■

Algorithm 3.4 (Topological Sort)
We begin by doing a DFS traversal on G. Then, we sort by postorder number. This exactly a topological sort

of the graph.
This works because peeling off the source causes the DFS to be the exact same, so the order of postordering numbers does not change.
Note that postorder number is bounded between 2 and $2n$, so it is easy to sort in linear time. Thus the runtime is dominated by DFS, $\mathcal{O}(n+m)$.

3.2.2 Strongly Connected Components

Next we try to find strongly connected components in a directed graph.

Definition 3.5 (Strongly Connected)
For some directed graph $G = (V, E)$, $u, v \in V$ are strongly connected if u has a parth to v and v has a path to u.

Definition 3.6 (Strongly Connected Component (SCC))
A strongly connected component in a graph G is a maximal subset of strongly connected vertices.

An easy algorithm to find SCCs would be to do n DFS's from each vertex and check strong connectivity. However, this is fairly slow, about $\mathcal{O}(mn + n^2)$.

Theorem 3.4 (SCC Graph is a DAG)
The graph made from treating the SCCs as nodes is acyclic (a DAG).

Proof
Suppose there was a cycle. Then, all the vertices in those components forming a cycle are strongly connected, which would contradict the fact that the SCCs were maximal. Thus, there cannot be a cycle.

We want to find a sink of this reduced DAG. In particular, running a DFS from the sink SCC, we can peel it off and recurse. This means we want to find the SCCs in reverse topological sorted order. Imagine we had a subroutine that could find a vertex in a sink in constant time. Then, the runtime would be: $\mathcal{O}(n+m)$ because we have to DFS over every single strongly-connected component, visiting every vertex and edge once.

To do this, first let us reverse the graph (make all the edges go the other direction). This will turn any source into a sink and vice versa. Now we have to find a source SCC in the reversed graph G_{rev}. The claim is that the highest postorder number still works.

Theorem 3.5
If u has the highest postorder number, then u is in the source SCC.

Proof
Suppose that u is not in the source SCC. That means there is another SCC that point into that SCC of u. This means that $(v, w) \in E$, where w has a path to u and u has a path to w. Since v thus has a path to u but u has the highest postorder number, the intervals cannot be disjoint. Furthermore, the intervals cannot be inside each other, as this would imply u would have a path v; but that would mean they're in the same SCC, which they're not. Thus, the edge (v, w) cannot exist, meaning that u is in the source SCC.

Finally our algorithm to find the SCCs is as follows:

Algorithm 3.5

1. Reverse the graph and run topological sort on it. This will give you the ordering S of sinks in the original

graph.

2. Run DFS on each sink s in order. The vertices reached are in SCC of s, so we can remove them from the graph (mark them) and keep going.
3. The SCCs removed are all the SCCs.

This algorithm requires a topological sort and then small DFS's which amount to a single big DFS. This is $\mathcal{O}(m+n)$.

3.3 Lecture 7

3.3.1 Single Source Shortest Paths

We now consider shortest paths. Suppose we are given a directed graph G and a start vertex s. We wish to find the shortest path from s to all other vertices. More precisely, we want two arrays:

- $\text{prev}[1, \dots, n]$, where $\text{prev}[v]$ is the previous vertex to v on the shortest path from s to v.
- $\text{dist}[1, \dots, n]$, where $\text{dist}[v]$ is the length of the shortest path from s to v.

Note that the prev array has enough information to give us the actual shortest path from s to v.

There are many algorithms for single-source shortest paths. Here are a few:

- Breadth-First Search - Assumes edges all have weight 1 ("unweighted")
- Dijkstra's Algorithm - Assumes edge weights are all non-negative.
- Bellman-Ford Algorithm - Arbitrary edge weights.
- Dynamic Programming on DAGs - Arbitrary edge weights, but G must be a DAG.

Note that depth-first search does not work because going deep first may yield a longer path than some other traversal. Consider the graph:

$$V = \{S, A, B\}, E = \{(S, A), (A, B), (S, B)\}$$

Then suppose the DFS traversal was S, A, B. Then the path to B would seem like it's distance 2, when in reality it's 1, since the (S,B) wouldn't be traversed by DFS.

Algorithm 3.6 (Breadth-First Search)
Here is the main algorithm:

```
function BFS($G, s$)
    $\text{dist}[1 \dots n] \leftarrow \infty$
    $\text{prev}[1 \dots n] \leftarrow$ null
    $\text{vis}[1 \dots n] \leftarrow$ False
    Q $\leftarrow$ queue($s$)
    $\text{dist}[s] \leftarrow 0$
    $\text{vis}[s] \leftarrow$ True
    $Q$.push($s$)
    while $Q$.size> 0 do
        $u \leftarrow Q$.pop()
        for $(u, v) \in E$ do
            if !vis[$v$] then
                $\text{vis}[v] \leftarrow$ True
                $\text{dist}[v] \leftarrow \text{dist}[u] + 1$
                $\text{prev}[v] \leftarrow u$
                $Q$.push($v$)
    return dist, prev
```

Runtime Analysis

Proof
The first 3 operations are linear in n, and the last 4 are constant time. Then note that every vertex is added to

the Q once, and all of its edges are looped over. So the total runtime is:

$$T(n) \leq \Theta(n) + \sum_{v \in V} C \cdot (1 + \text{outdegree}(v))$$

Note that the sum of the outdegrees counts every edge once. Thus, $T(n) = \mathcal{O}(m+n)$.

The interesting thing about BFS is that it is implemented the exact same as iterative DFS, but the stack is replaced with a queue.

To show correctness, we require a few intermediate results:

Theorem 3.6 (BFS Lemma 1)
$\forall v \in V$, $\text{dist}[v] \geq \delta(s,v)$ (the true shortest distance).

Proof
We proceed by induction on k, the number of push operations to Q so far. $k = 1$ is trivial, since the dist of s is 0 and everyone else's distance is infinity, satisfying the inequality. Consider the $k+1$st push (with the induction holding for $\leq k$), during the edge (u,v). We push v when we visit u. By the inductive hypothesis, $\text{dist}[u] \geq \delta(s,u)$, so

$$\text{dist}[v] = 1 + \text{dist}(u) \geq \delta(s,u) + 1 = \delta(s,u) + \delta(u,v) \geq \delta(s,v)$$

completing the induction.

Theorem 3.7 (BFS Lemma 2)
Look at any point in time i. Say $Q = [v_1, \ldots, v_r]$. Then

1. $\forall i, \text{dist}[v_i] \leq \text{dist}[v_{i+1}]$
2. $\text{dist}[v_r] \leq \text{dist}[v_1] + 1$

Proof
The proof is similar to the theorem above. Induct on the number of queue operations.

Now we can show the correctness of BFS.

Theorem 3.8 (The Correctness of Breadth-First Search)
We will show that BFS finds shortest paths from s.

Proof
For the sake of contradiction, suppose the dist arry is incorrect. Then, there is some $v \in V$ such that $\text{dist}[v] \neq \delta(s,v)$. By BFS Lemma 1, we must have $\text{dist}[v] > \delta(s,v)$. This may be the case for many such v. Let us pick the v such that $\delta(s,v)$ is minimum (note that $v \neq s$). Then, let us take the shortest path from s to v.

$$s \longrightarrow v_1 \ldots v_{r-1} \ldots v$$

This means for all intermediate vertices until v_{r-1}, the dist array was set correctly. Look at the point in time when v_{r-1} was popped off Q. But this means that v was not put into the Q (since the distance was resolved incorrectly) at this point. This means that v was visited already by some other vertex u and by BFS Lemma 2, this means that $\text{dist}[u] \leq \text{dist}[v_{r-1}]$. But this would mean that v got set to some value:

$$\text{dist}[v] = \text{dist}[u] + 1 \leq \text{dist}[v_{r-1}] + 1 = \delta(s,v)$$

However, we initially claimed $\text{dist}[v] > \delta(s,v)$, so this is a contradiction! So we cannot have any place where the dist array is incorrect.

3.3.2 Weighted Graphs

Note that if all weights $w(e) \in \mathbb{N}$, then we can reduce finding the SSSP to the unweighted case by just subdividing edges into $w(e)$ fake vertices in the middle. If $w(e) \leq L$, then the BFS runtime is $\mathcal{O}(n + mL)$. We can do better (and also use $w(e) \in \mathbb{R}$).

To do this, we use heaps (or priority queues). A min-heap is a data structure that maintains a set of (key, value) pairs S subject to the following three operations:

1. delMin(), returns (k, v) from S with the smallest k and removes it from S.
2. decKey(O, k'), where O is a pointer to the v object, replaces the old key with a smaller key k'
3. insert(k, v), inserts (k, v) into S

We can now use these heaps to solve SSSPs for arbitrary non-negative weighted graphs.

Algorithm 3.7 (Dijkstra's Algorithm)

```
function DIJKSTRA($G, s$)
    dist$[1 \dots n] \leftarrow \infty$
    prev$[1 \dots n] \leftarrow$ null
    $H \leftarrow$ heap()
    for $v \in V$ do
        $H$.insert$(\infty, v)$
    dist$[s] \leftarrow 0$
    $H$.decKey$(s, 0)$
    while $H$.size$> 0$ do
        $u \leftarrow H$.delMin()
        for $(u, v) \in E$ do
            if dist$[u] + w((u, v)) <$ dist$[v]$ then
                $H$.decKey$(v,$ dist$[u] + w(u, v))$
                dist$[v] \leftarrow$ dist$[u] + w((u, v))$
                prev$[v] \leftarrow u$
    return dist, prev
```

Runtime Analysis We do n insertions and thus must do n deleteMins, and we are doing potentially deg(v) insertions for each, so overall, considering t_I as the runtime of insert, t_{dK} for decKey and t_{dM} for delMin, we have the runtime is something close to: $T(n) = \mathcal{O}(n + m + nt_I + nt_{dM} + mt_{dK})$ which is:

$$T(n) = \mathcal{O}((m + n) \log n)$$

for a binary heap implementation.

Proof of Correctness First realize the following: Any non ∞ dist$[u]$ value corresponds to some $s \rightarrow u$ path. This can be shown formally by induction on the main loop of the algorithm.

The main claim is that the final dist and prev arrays are correct, so for all u, dist$[u] = \delta(s, u)$. Note that if we do this, then the prev array is correct because note that we change the prev arrays exactly when the dist arrays are set, so the path traced out is the correct one.

To show the claim, let

$$A = \{\text{all vertices that have been popped off the heap so far}\}$$

at the end of the algorithm, the heap is empty so all the vertices have been popped off. Thus, we will show the following invariant: that for any $u \in A$, dist$[u] = \delta(s, u)$. We proceed by induction.

Base case: If $|A| = 1$, then only s has been removed off the heap. Since we set its distance initially, we know dist$[s] = 0 = \delta(s, s)$.

Inductive step: Say $|A| = k + 1$, so we just popped a vertex v off the heap. So, we just need to make sure dist$[v]$ is correct; the rest are correct by inductive hypothesis ($|A| = k$ without v). Then, v must've been the minimum key in the heap, so its dist must've been set last by some $u \in A$ (since it was popped off previously). Thus, there is an edge (u, v). Furthermore, there is a path from s to u called p by the inductive hypothesis. We claim that appending (u, v) to p yields the shortest path. Suppose it wasn't and we could do better, with some other path p'. Then at some point, this path must leave A (since v was still outside A at this point). Consider the last vertex before it leaves A as x and call the first vertex after it leaves as y. Let the subpath from s to x be q. Then, the length of p' is as follows:

$$\begin{aligned} L(p') &\geq L(q) + w(x, y) \\ &= \delta(s, x) + w(x, y) \\ &= \text{dist}[x] + w(x, y) \end{aligned}$$

But since y is a neighbor of x, then we had to have updated it in the past, at least through the update at vertex x. Thus, the dist value is at most the last line, since that is what an update in dijkstra's looks like.

$$L(p') \geq \text{dist}[y]$$

However, since v was the last thing popped off the heap and y wasn't, its dist value is lower.

$$\begin{aligned} L(p') &\geq \text{dist}[v] \\ &= L(p) \end{aligned}$$

Which means that p' can't be any shorter than p, completing the induction and the proof.

3.4 Lecture 8

3.4.1 Minimum Spanning Trees

We look at finding the minimum spanning tree in some connected graph G. First, we define these notions.

Definition 3.7 (Tree)
A tree is an undirected graph that is connected and acyclic.

Definition 3.8 (Spanning Tree)
The spanning tree of some undirected graph $G = (V, E)$ is a subgraph of G called $T = (V, E')$ with the same vertices and edges $E' \subseteq E$.

Then, the notion of a minimum spanning tree is finding a spanning tree whose edge weights added up have minimal cost.

Here is a nice result about trees.

Theorem 3.9
Consider some graph T. Any of the following two implies the third:

1. $|E(T)| = n - 1$.
2. T is connected.
3. T is acyclic.

As a corollary, any two of these define a tree.

Here is a slow algorithm to tackle the problem.

Algorithm 3.8 (Brute Force MST)
Take all subgraphs with $n - 1$ edges. Next, check if each is connected (using DFS), if it is connected, then calculate the weight. Finally take the minimum of the weights.
The runtime is roughly:

$$\mathcal{O}\left(\binom{m}{n-1}(n-1+n)\right) \leq \mathcal{O}\left(\binom{n^2}{n-1}n\right) \leq \mathcal{O}(n^n)$$

Next, let's discuss a faster algorithm; it's a meta-algorithm that we can use to make other algorithms by filling in key implementation details.

Algorithm 3.9 (MST Meta-algorithm)

```
function MetaAlg($G$)
    $X \leftarrow \emptyset$
    while $|X| < n - 1$ do
        Pick $S \subsetneq V, S \neq \emptyset$ such that no edge in $X$ cross $(S, V \setminus S)$
        Let $e = (a, b)$ be one of the min-weight edges in $G$ crossing the partition
        $X \leftarrow X \cup \{e\}$
    return $(V, X)$
```

Proof of Correctness We will show at all points in time, there exists an MST T containing X. If we show this, this means that at the end of the while loop, since T conatins X and is the same size, X becomes T. We proceed

by induction on $|X|$.
Base case: $|X| = 0$ works because a connected graph G always has an MST, which trivially contains the empty edge-set.
Inductive step: Suppose that for X so far, it was contained by T. Now, we are adding one more edge $e = (a, b)$ where $a \in S$. There are two cases:
Case 1: T had the edge (a, b). Then X is still contained by T.
Case 2: T did not have the edge (a, b). Note that since T is a spanning tree, there must be a path from a to b, i.e. it must pass the partition V and $V \setminus S$ somewhere. If we combine this path with (a, b), we get a cycle. Then, there must be a family of edges which all cross the partition; removing any one of them forms a tree. Suppose now we remove any edge (x, y). We define

$$T' = T \cup \{(a, b)\} \setminus \{(x, y)\}$$

However, the weight of $T' \leq$ weight of T since the weight of (a, b) is at most the weight of (x, y) because by construction it is the lightest edge across the partition. This means that T' is also an MST, which also contains X, so the claim holds.

Now, we are left with a choice: how do we construct our paritioning set S?

Algorithm 3.10 (Prim's MST Algorithm)
Prim's partitions the set based off a starting vertex s.

```
function Prim($G, s$)
    $X \leftarrow \emptyset$
    $S \leftarrow \{s\}$
    for $n - 1$ times do
        $(a, b) \leftarrow$ min-weight crossing($S, V \setminus S$), where $b \notin S$
        $S \leftarrow S \cup \{b\}$
        $X \leftarrow X \cup \{(a, b)\}$
    return $(V, X)$
```

To implement this efficiently, we get the min-weight crossing by using a heap. We greedily take the cheapest edge that's not yet in our tree; this is precisely that min-weight crossing. Every time we add something new to S, all we should do it scan the edges and update it's distance from our tree accordingly. The implementation is ommitted here.
Runtime Analysis The runtime becomes the exact same as Dijkstra's algorithm, since we update distances and pop things off the heap the same way (except updating distances is the distance to the tree, instead of doing an addition), so with a binary heap, the runtime is the same $\mathcal{O}((m + n) \log n) = \mathcal{O}(m \log n)$ since $m \geq n - 1$ since the graph is connected.

To present another approach, we first discuss a new data structure.

Note 3.2 (Union-Find Data Structure)
Suppose we want to maintain a partition of $\{1, \ldots, n\}$ subjects to 2 operations.

1. find(a) returns the name of the partition that a is in.
2. union(a, b) merge the partitions containing a and b.

Now we discuss an even more greedy approach.

Algorithm 3.11 (Kruskal's Algorithm)
Kruskal's algorithm uses a global approach.

```
function Kruskal($G$)
    Sort $E$ in increasing order of weight.
    $X \leftarrow \emptyset$
    Initialize a UnionFind data structure.
    for $(a, b) \in E$ (in increasing order of weight) do
        if find$(a) \neq$ Find$(b)$ then
            $X \leftarrow X \cup \{e\}$
            union$(a, b)$
    return $(V, X)$
```

4 Greedy Algorithms

4.1 Lecture 9

A greedy algorithm is not super mathematically well-defined property. Generally, we can separate the problems that greedy can solve into two types of problems:

1. Search problems: which try to find an object in a large set.
2. Optimization problems: which try to find the object (or objects) in a large set that have some maximal or minimal property.

So, we then define:

Definition 4.1 (Greedy Algorithm)
A greedy algorithm is one that builds the solution to a problem iteratively using a sequence of local (or locally optimal) choices.

Note that brute-force search is not greedy; in a greedy algorithm, you generally stick to your local decisions, never looking back.

We now discuss three problems that one can solve with a greedy approach.

4.1.1 Scheduling

We take as input as a collection of jobs that need to be done. Each job has a time interval $[x_1, y_1], \ldots, [x_n, y_n]$ where $x_i < y_i$. Now, we want to find the maximal amount of jobs that can be done without any time conflicts (e.g. with one thread).

To be greedy, what local choice should be make?

One idea is to greedily keep doing the shortest possible remaining job; it unfortunately does not work. Consider the jobs $[1, 10], [9, 11], [11, 20]$, where this algorithm will choose only the middle interval as one job, but clearly the optimal solution is taking the first interval and the last one.

Instead, let us make a different choice.

Algorithm 4.1 (Scheduling)
Greedily pick the next job to have the shortest y_i (end time) without conflicting with already-done jobs.
Runtime Analysis We can implement this by sorting by the end-times. Then take the first element $[x, y]$ and consider the next element $[x', y']$. Since $y \leq y'$ then checking for no conflict is just checking that $x' \geq y$. Thus, we have to just do a for loop over the array to implement the above logic. The bottleneck is the sort, giving us $\mathcal{O}(n \log n)$.
Now, to prove correctness for greedy algorithms, we generally use an exchange argument, where we postulate the existence of a solution strictly better to the output produced by the algorithm, and show a contradiction.
Proof of Correctness Suppose our algorithm gives J, giving jobs $j_1, j_2, \ldots, j_k$ (in sorted order by endpoint) and this not optimal. Take some optimal solution S such that $|S \cap J|$ is maximum. We will show there exists another optimal solution S' such that $|S' \cap J| > |S \cap J|$, giving us a contradiction of maximality.
Let us sort by endpoint on S, making $s_1, s_2, \ldots, s_k, s_{k+1}, \ldots$. The j's are not the same as the s's because otherwise the greedy algorithm would've taken more s's (since the s's are not conflicting and sorted by end time). Thus, there must exist some first t such that $j_t \neq s_t$. Now, let us introduce a new solution S' with s_t is replaced by j_t. j_t's end-time is earlier or at the same time as s_t, so there cannot be any conflicts after j_t (otherwise there would've been a conflict with s_t) and there cannot be any conflict before j_t (because the rest of the s's agree with j, and there is no conflict among the j's). Furthermore, $|S'| = |S|$. Thus, S' is also an optimal solution; but $|S' \cap J| = |S \cap J| + 1 > |S \cap J|$, as required for contradiction.

Now, working through the example above, we will select $[1, 10]$ first, then the second interval won't be added since it's conflicting, then the third interval will be added since it's not conflicting.

4.1.2 Huffman Encoding

We now discuss the problem of efficient encoding. Suppose we have some alphabet Σ, $|\Sigma| = n$ (e.g. $\{A, \dots, Z\}$) and some text w whose characters come from Σ. Now there are character frequencies:

$$freq(\sigma) = \text{Number of occurences of } \sigma \text{ in text}$$

Now, how can we encode the text as efficiently as possible using these characters (minimizing codeword length)? One concern for encoding is that certain codewords may have the same word that created it. We want our encoding to be unambiguous; one way to ensure this is using prefix-freeness.

Definition 4.2 (Prefix-Freeness)
An encoding is prefix-free if no character's encoding is a prefix of any other character's encoding.

Note that any code that is prefix-free cannot be ambiguous, because the unique prefix will tell you which character to look at. Thinking in terms of bits, the following encoding:

$$A = 0, B = 01, C = 001, \dots$$

can be represented by a binary trie where each branch is a 0 or 1; the leaves are all characters.

Now, we want to find the optimal prefix-free encoding. Naively, we could find all binary trees on n leaves ($2n - 1$ nodes) and assignments of characters to leaves. The number of binary trees is exponential, about 16^n and there $n!$ assignments. We also have to do a linear check of length to check cost; this means our total runtime is $\mathcal{O}(n!)$. Note that even we assigned all characters to leaves greedily, this would still be an exponential-time algorithm!

Furthermore, being fully greedy and letting the highest frequency character hang at the top of the tree is also not optimal. In the case of equal frequency, we want a perfectly balanced tree (the best compression of 26 characters is to use 5 bits, if they are equal frequencies).

Here is a better way:

Algorithm 4.2 (Huffman Encoding)
We try to build the tree bottom-up. Note that at the lowest depth, any node v has a sibling (otherwise we could delete the parent of v and replace v with that to have less bits). At these two leaves, let us place the least-frequent characters here. We claim this is optimal; suppose there was another optimal way to do things where a different pair was at the bottom of the tree, then you can switch the one of the least-frequent to the bottom, saving you characters in your encoding.
To do this algorithm greedily, we simply repeat this choice over and over iteratively. To see this in action, we present an example.
First, notice that we have to do $n-1$ iterations. To find the smallest characters in the set, we can keep a min-heap, requiring 1 $\Theta(\log n)$ insertion and 2 $\Theta(\log n)$ removals. Then, building the tree is done in constant time in each iteration. Our total runtime is $\Theta(n \log n)$.

Example 4.1
Consider the following frequencies on the alphabet of capital vowels:

$$A : 60, E : 70, I : 45, O : 50, U : 20, Y : 30$$

Placing the two smallest nodes U and Y at the bottom of the tree then creates a certain "meta-character" $\{U, Y\}$ with frequency $20 + 30 = 50$. Our frequencies then become:

$$A : 60, E : 70, I : 45, O : 50, \{U, Y\} : 50$$

Now we take the next two smallest nodes (I and $\{U, Y\}$) and place it at the bottom of tree, creating:

$$A : 60, E : 70, O : 50, \{I, U, Y\} : 95$$

Continuing, we have:

$$\begin{aligned}&A : 60, E : 70, O : 50, \{I, U, Y\} : 95\\&E : 70, \{I, U, Y\} : 95, \{A, O\} : 110\\&\{E, I, U, Y\} : 165, \{A, O\} : 110\\&\{A, E, I, O, U, Y\} : 275\end{aligned}$$

Then, we've build the tree; we can traverse with DFS to then get the encoding per character.

4.1.3 Set Cover

Now we consider another problem. We are given a Universe $\{1, \ldots, n\} = [n]$ and a collection $C = \{S_1, \ldots, S_m\}$ of subsets of $[n]$. We want to find a minimum-size subcollection to cover $[n]$. Generally, we don't expect there to be a polynomial-time algorithm to solve this. The natural greedy approach (where we take the biggest subset every time) does not give the optimal solution (easy to see with a counterexample).

Let the optimal number of sets be OPT. Feige showed that if there exists a polynomial-time algorithm for set cover, which always uses less than $OPT \times \ln n$ sets, then 3-SAT can be solved in time $\leq n^{\log\log n}$ (which we don't believe to be true).

This makes greedy even more tantalizingly close:

Theorem 4.1
The natural greedy approach uses $\leq OPT \times \lceil \ln n \rceil$ sets.

The greedy algorithm for set cover is as follows:

Algorithm 4.3

```
function GreedySC($C$)
    $A \leftarrow \{1, \ldots, n\}$ // Not covered yet
    $B \leftarrow \emptyset$ //Sets taken
    while $|A| > 0$ do
        Let $j \in [m] \setminus B$, be s.t. $|A \cap S_j|$ is max
        $A \leftarrow A \setminus S_j$
        $B \leftarrow B \cup \{j\}$
    return $B$
```

Now we prove our earlier claim. Say that OPT uses k sets. We want to bound $|B|$.

Proof
Let A_t be A after t times through the main loop. Note $|A_0| = n$. There must be some set S_t^* in OPT covering $\geq \frac{1}{k}|A_t|$ elements in A_t, because on average, every set covers that many elements on A_t, so there must be at least one that is at least the average. This means that greedy took some set covering $\geq \frac{1}{k}|A_t|$ elements in A_t. Therefore:

$$|A_{t+1} \leq \left(1 - \frac{1}{k}\right)|A_t|$$

By induction on t, we just have:

$$|A_L| \leq \left(1 - \frac{1}{k}\right)^L |A_0| = n\left(1 - \frac{1}{k}\right)^L$$

Now, we just want to understand when this quantity is less than 1. Since $1 + x < e^x$, then:

$$|A_L| < n(e^{-L/k}) < 1$$

Which means $|B| = L^* \leq \lceil k \ln n \rceil$.

4.2 Lecture 10

4.2.1 Union Find

We now investigate how to build a union find data structure that we described earlier (disjoint sets where find($\cdot$) tells you the name of the set that some number is in, and union($\cdot$, $\cdot$) unions together the associated sets). See the Kruskal's section for more information.

First, think of a naive approach. We could potentially store an array $A[1\dots n]$, where $A[x]$ tells you the name of x's set. find(x) is easy, just return $A[x]$. This runs in constant time. To union(x, y), we find $a = A[x]$, $b = A[y]$ and then change all of the indices having value a to b (by looping over them). This is linear time.

Secondly, let's try an array of linked lists. Represent x by a linked list node. $A[x]$ is a pointer to a linked list node that corresponds to x (which is itself part of some circular doubly-linked list). Thus, we end up with a collection of linked lists; make each of these one of the sets.

Then, to find(x), we follow the pointer $A[x]$ all the way up its linked list to the head. This is worst-case linear time. To union(x, y), then put the head of one linked list to point to the tail of another linked list. This also takes linear time to traverse, and then changing the pointers takes constant time.

Let us think of an optimization. How about if every node stored a pointer to its list's head node? Well, during a union, we may need to update all the head pointers in some list; this would make union linear time and find constant time. Another optimization is to make sure the smaller list always goes under the bigger list. To do this, we can have the head pointers store their size.

Now the claim is that this optimized version, any sequence of operations consisting of at most m finds and at most n unions takes total time $\Updownarrow + \backslash \log \backslash$. This is slower than the sorting step in Kruskal's and is actually enough to get optimal runtime.

Proof
First, doing a find takes constant time. But a single union takes:

$$\mathcal{O}(1) + \mathcal{O}(\text{head pointer changes})$$

Thus, the total runtime is:

$$\mathcal{O}(m + n + \text{total head pointer changes}) = \mathcal{O}\left(\sum_{x=1}^{n} (\text{number of times I changed } x\text{'s' head pointer})\right)$$

Note that whenever I change a head pointer, the size of the list that x is a part of at least doubles. Thus, the maximum amount of times that list can double is $\log_2 n$, which will thus be at least amount of times I changed each node's head pointer. Thus, we end up with:

$$\mathcal{O}(m + n \log n)$$

as we wanted to show.

However, this isn't the best we know. We know an even better algorithm, with a disjoint forest. But first, we must digress to amortized runtime, to formalize the idea of "good" average cost.

Definition 4.3 (Amortized Runtime)
Suppose a data structure supports operations $O_1, O_2, \dots, O_k$. Then, we say the amortized cost of each operation t_j if for any sequence of operations with N_i of operation O_i over all i, the total time is

$$\leq \sum_{j=1}^{k} t_j N_j$$

Now, we look at the best way we know of implementing union find.

Algorithm 4.4 (Disjoint Forest Union Find)
Let us represent sets as collections of rooted trees. First, we consider the path compression optimization. What this means is that whenever we traverse a find from some node to a root, then for every node we pass along the way, we make its pointer point directly to the root.
Then, we do an analogous "length" optimization for union, where you make the shallower "list" (now trees) just point its root to the deeper tree's root (again, by storing the depth of the tree e.g. in an array). Then, to update the depth; it doesn't change if the depths of the trees are different, and if they are the same, then the depth of the root increases by 1.
However, doing both of these optimizations creates a problem: we have to update the depth every time we do path compression. We just ignore updating the heights during a path compression (and maybe call this number rank instead of height).
Here is the pseudo-code.

```
$p[1 \dots n], r[1 \dots n]$
  function MakeSet($x$)
      $p[x] \leftarrow x$
      $r[x] \leftarrow 0$
  function Find($x$)
      if $x = p[x]$ then
          return $x$
      $p[x] \leftarrow$ Find($x$) return $p[x]$
  function Union($x, y$)
      $x \leftarrow$ Find($x$)
      $y \leftarrow$ Find($y$)
      if $x = y$ then
          return
      if $r[x] > r[y]$ then
          Union($x, y$)
      $p[x] \leftarrow y$
      if $r[x] = r[y]$ then
          $r[y] \leftarrow r[y] + 1$
```

Runtime Analysis Let $\log^*(n)$ be the amount of times you have to take the $\log(n)$ before you get down to 1 (for all practical purposes, this is no more than 5). We claim any sequence of at most m Finds and n Unions/Makesets takes total time

$$\mathcal{O}((m+n)\log^*(n))$$

This gives us amortized $\log^*(n)$ runtime for all the operations.
Before we do this, we claim some properties:

1. Any root node x has $\geq 2^{r[x]}$ elements in its tree. (Can prove with induction; it's an invariant)
2. Ranks strictly increase as you follow parent pointers. (Can prove with induction; it's an invariant)
3. The number of nodes with rank exactly k is $\leq \frac{n}{2^k}$.

First, let us show claim 3; this is a bit tricky.

Proof
Say the items of rank exactly k are $x_1, \dots, x_q$. We will show there exists disjoint subsets of the universe of nodes $S_1, \dots, S_q$ such that $|S_i| \geq 2^k$. We claim this is enough. Consider

$$\left| \bigcup_{i=1}^{q} = S_i \right| \leq n$$

but also that

$$\left|\bigcup_{i=1}^{q} = S_i\right| = \sum_{i=1}^{q} |S_i| \geq q \cdot 2^k$$

meaning,

$$q \leq \frac{n}{2^k}$$

Thus, we need to just find these S_i. Let S_i be the set of items in x_i's tree at the moment in time when x_i becomes rank k. This means that x_i merged with a $k-1$ rank tree, giving 2^k nodes at least under x_i. We now have to just argue that the S_i's are disjoint. Consider another x_j and when it became rank k, (which happened at some other time than x_i). WLOG, suppose x_i became rank k before x_j. Suppose FSOC the two trees under x_i and x_j shared a node z; this may mean x_i is under x_j, but this is impossible because by claim 2, rank must strictly increase when following parent pointers. Thus, z must be under something of rank higher than x_i; but none of these exist in the x_j tree. Thus, the trees under x_i and x_j must be disjoint. Thus, the sets S_i are disjoint.

Now, here is the proof of the main runtime:

Proof
Imagine every number from 1 to n either pays cash or charges its credit card (in terms of running time, every time it does an operation). Then, to find the total running time (cost) we add up this "cash" and add up all the credit card dues.
Look at our n items at any point in time. They all have some ranks; the ranks go from 0 to $\log n$. We can place these items into buckets based on rank, in the following way:

$$[0, 1), [1, 2^1), [2, 2^2), [2^2, 2^{2^2}), \ldots$$

The largest rank that can exist is $\log_2(n)$ (by claims 1 and 2), so the number of groups is $\log^*(\log_2(n)) = \log^*(n) - 1$.
Now let's do some "accounting." Find is generally what takes time; the total time is:

$$\mathcal{O}(m + n + \#\text{ of parents pointers I follow})$$

Suppose we follow a parent pointer from u to v. Then, we have a few cases:

1. v is the root of its tree. Let's pay cash here. In every find/union operation, we do this exactly once, so this is $\mathcal{O}(1)$ per op.

2. u, v are in different groups (as defined above) with neither as the root. We will again "pay cash". how many times do we change groups? Since rank increases as we follow parent pointers, so the maximum amount of times we can change groups while going up is $\mathcal{O}(\log^*(n))$ per op.

3. u, v are in the same groups (as defined above) with neither as the root. We charge these operations to u's credit card; we will account for this at the end.

Now, let us understand what our "credit card debt" is from case 3. Let $g[u]$ be the group of u after all the operations. Then, the total debt is:

$$\text{debt} = \sum_{u=1}^{n} \#\text{ of times } u \text{ was charged}$$

$$\text{debt} = \sum_{u=1}^{n} \sum_{t=0}^{g[u]} \#\text{ of times } u \text{ was charged while in group } t$$

Now suppose u is in the group, $[k, 2^k]$. In case 3, u's new parent after a find is the root (due to path compression), which has a strictly higher rank than v. If u is charged again while staying in this group, then its parent is again replaced with a new root (which must have an even higher rank). So, we can only charge case 3 while still in this group at most $|[k, 2^k]| = 2^k - k \leq 2^k$ times.

$$\begin{aligned}
\text{debt} &\leq \sum_{u=1}^{n} \sum_{t=0}^{g[u]} \mathcal{O}\big(2^t\big) \\
\text{debt} &\leq 2 \sum_{u=1}^{n} 2^{g[u]} \\
\text{debt} &\leq 2 \sum_{\text{groups}} 2^k \cdot \#\text{ items in group } k \\
\text{debt} &\leq 2 \sum_{\text{groups}} 2^k \cdot \sum_{j=k}^{2^k-1} \frac{n}{2^j} \\
\text{debt} &\leq 2 \sum_{\text{groups}} 2^k \cdot 2 \cdot \frac{n}{2^k} \\
\text{debt} &\leq 4n \cdot \#\text{ of groups} \\
\text{debt} &\leq 4n \log^*(n)
\end{aligned}$$

Thus, all operations are $\mathcal{O}((m+n)\log^*(n))$ combined. Thus, the amortized runtime of find and union are: $\mathcal{O}(\log^*(n))$.

5 Dynamic Programming

5.1 Lecture 11

Dynamic Programming can be thought of in two structures:

1. Top-down DP (Memoization): Recursion and a lookup table to not have to recompute $f(\cdot)$ multiple times on the same arguments.
2. Bottom-up DP: Fill up the lookup table iteratively instead of recursively.

You may have seen this in the past with the Fibonacci Sequence. We can implement top-down Fibonacci with the recurrence relation:

$$a_n = a_{n-1} + a_{n-2}$$

but include a lookup table that checks if a_n has already been calculated. However, bottom-up Fibonacci just has the base case $a_0 = 0, a_1 = 1$, and then iteratively calculates:

$$a_{i+2} = a_{i+1} + a_i$$

When considering the recursive version of Fibonacci, we can draw a recursive tree. Furthermore, this is a dependency DAG! In an optimization problem, each choice may have a certain weight attached to it. We should fill in the lookup table for the DAG in reverse topological sorted order, and then output the last entry in the table.

5.1.1 Shortest Paths on a DAG

This algorithm works even when edge weights are negative. We use the same dynamic programming approach as we did with Fibonacci (which is fundamentally a DAG).

We will solve this using a general framework for DP.

1. Think of a function $f(\cdot)$ that can be computed recursively such that I can extract the answer by looking at $f(x)$ for some x.

   Here, we define $f(t)$ as the length of the shortest path from s to t.

2. Make a recurrence relation for $f(\cdot)$ (it's fine if this is brute-forcy).

   $$f(t) = \begin{cases} 0 & t = s \\ \infty & t \text{ is a sink, } t \neq s \\ \min_{(u,t)\in E} w(u,t) + f(u) & \text{otherwise} \end{cases}$$

   Note that this runs VERY slowly. Suppose you have d diamonds in your DAG, so there are $3d + 1$ vertices; i.e. $2^d = 2^{n/3}$ paths, giving us exponential time.

3. Optimize the recurrence relation with a lookup table.

   In this case, let's just store $f(a)$ if we've computed it already. This means that the runtime the sum of time spent on input summed over all the inputs:

   $$sum_{u\in V} C \cdot (1 + \text{indegree}(u)) = \Theta(m + n)$$

4. Potentially run the algorithm in some natural order (in the order of dependencies; i.e. a reverse topological sort). This may save space.

Here's a way you might run this algorithm:

Algorithm 5.1 (Shortest Paths in a DAG)
Reverse the graph to get G^R. As base cases, if s is t, return 0; if t is a sink (source in the main graph) return ∞. Otherwise, loop over all outgoing edges of t in G^R and recurse on those. Take the min of the subproblems and adding the weight of the incoming edge to get $f(t)$. Save the result of $f(t)$ in a memoization table.

5.2 Lecture 12

The first worry is that our dynamic programming examples often return only a cost, instead of the optimal "solution" (i.e. maybe the edges in a path instead of just a length). However, we argue it is easy to modify dynamic programming algorithms to recover this.

Thinking about the dependency DAG of the subproblems, we can see how to do this. We simply find the shortest/longest path in a DAG, which we already solved! To find the path explicitly, we can store the 'argmin' of the edge that minimized the shortest path, in a new lookup table ('prev' or 'choices').

5.2.1 Bellman-Ford

We once again revisit the problem of single-source shortest paths on directed graphs. We already know Dijkstra's algorithm; however, this algorithm does not work if some of the edge weights are negative. Furthermore, if there is a negative cycle in your graph, then the shortest path to certain nodes may be ∞ (keep following the cycle over and over to get an arbitrarily small length). Let us look at an algorithm which can find the shortest paths if they are well defined (even if edge weights are negative) or find the existence of a negative cycle.

We first realize the following recursive thinking. Any shortest path from source s to vertex t has a last edge v on the path. However, the last recurrence relation will not work because of the cycles possible in the graph. Furthermore, any shortest path to t, if it exists, has at most $|V| - 1 = n - 1$ edges (otherwise there is a cycle in that path we can remove which is not negative). Thus, this is what we develop at first.

Algorithm 5.2 (Bellman-Ford (DP))
Define $f(t, k)$ as the shortest path from s to t using $\leq k$ edges. We want to find $f(t, n-1)$. Let us develop a recurrence:

$$f(t,k) = \begin{cases} \infty & k = 0, t \neq s \\ 0 & k = 1, t = s \\ \min\{f(t,k-1), \min_{(v,t)\in E}\{w(v,t) + f(v,k-1)\} & \text{otherwise} \end{cases}$$

```
function BF($G$,$s$)
    $T[1 \dots n][0 \dots 1] \leftarrow \infty$ for all values
    $T[s][0] \leftarrow 0$
    for $k = 1$ to $n - 1$ do
        for $t = 1$ to $n$ do $T[t][1] \leftarrow T[t][0]$
            for $(v,t) \in E$ do
                $T[t][1] \leftarrow \min(T[t][1], w(v,t) + T[v][0])$
        for $t = 1$ to $n - 1$ do
            $T[t][0] \leftarrow T[t][1]$
    return $T[1 \dots n][1]$
```

Memory Analysis The memory table is $\mathcal{O}(n^2)$ size because t can range from $1, n$ as can the path length. However, note that $f(\cdot, k)$ only depends on $f(\cdot, k-1)$ values, so we need to only have to remember a 1-d array of $f(\cdot, k)$ values. So, we can reduce that to $\mathcal{O}(n)$ with bottom-up DP.
Runtime Analysis The runtime is n times total time to compute all $f(\cdot, k)$. This is:

$$n \sum_{t\in V} C \cdot (1 + \text{indeg}(t)) = \mathcal{O}(n^2 + mn)$$

However, this isn't the best we can do. Instead, the actual Bellman Ford runs yet faster on sparse graphs (in a textbook).

Algorithm 5.3 (Bellman-Ford)

```
function BFBook(()$G$, $s$)
```

```
$T[1 \ldots n] \leftarrow \infty$ for all values
$T[s] \leftarrow 0$
for $k = 1$ to $n - 1$ do
    for $e = (u, v) \in E$ do
        $T[v] \leftarrow \min(T[v], T[u] + w(u, v))$
return $T$
```

Runtime Analysis Outer loop runs in n, inner in m; this gives us $\mathcal{O}(mn)$.
Why are these equivalent? Well, first, note that this is just a different ordering of filling in the DP table T. This is because instead of using a specific node every time, we still look at all edges. Now, the question is, why is our replacement overwriting our DP table instead of making a second column? First we make the following claims (easy to check with induction):

- At all points in time, for all v, $T[v]$ is the length of some path from $s \to v$ (counting the empty path as a path with length ∞).
- For all k and v, after going through the outer loop k times, $T[v] \leq f(v, k)$.

Thus this means after all the iterations, $\delta(v) \leq T[v]$ (because all paths are at least the length of the shortest path), also $T[v] \leq f(v, n-1) = \delta(v)$ by definition. Therefore, we must have $T[v] = \delta(v)$, as we claim.

Now, we have our finished algorithm; where is the negative cycle detection we've been promised. Assume WLOG everyone is reachable from s. We show the following:

Theorem 5.1
There exists a negative cycle if and only if $f(v, n) < f(v, n-1)$ for some v in the recurrence we defined above.

Proof
The if condition is straightforward. Suppose that there was some vertex v where $f(v, n) < f(v, n-1)$ but there was no negative cycle. Because any path with at least n edges has a cycle, this means that we could splice out the cycle (which must be positive by assumption) and make a path with $n-1$ or less edges, which would mean $f(v, n) \geq f(v, n-1)$, which is a contradiction.
The only if is a bit trickier. We proceed by contrapositive. Suppose that for all v, $f(v, n) \geq f(v, n-1)$. Now, consider some cycle $C = v_1 \to v_2 \to \cdots \to v_r \to v_1$. Well, we know that for all i,

$$\begin{aligned} f(v_{i+1}, n-1) \leq f(v_{i+1}, n) &\leq f(v_1, n-1) + w(v_i, v_{i+1}) \\ \sum_{i=1}^{r} f(v_{i+1}, n-1) - f(v_{i+1}, n) &\leq \sum_{i=1}^{r} w(v_i, v_{i+1}) \\ 0 &\leq \sum_{i=1}^{r} w(v_i, v_{i+1}) \end{aligned}$$

Thus the sum of the weights of C is non-negative, meaning we cannot have a negative cycle, which is exactly the contrapositive of our only if claim.

5.2.2 Floyd-Warshall Algorithm

Now we consider the problem of all pairs shortest paths. As input, we are given directed graph G. We should return the 2d array d such that for all i,j, $d[i][j]$ = total length of the shortest $i \to j$ path.

One way to do this would be to run Bellman-Ford n times, once for each vertex. This gives us a total runtime of $\mathcal{O}(n^2 m) = \mathcal{O}(n^4)$ if the graph is dense. We can do better. Assume without loss of generality that G is the complete graph (if $e \notin E$, just pretend $w(e) = \infty$). We do the following:

Algorithm 5.4 (Floyd-Warshall Algorithm)
Our recurrence relation is as follows. $f(i, j, k)$ is the length of the shortest path from i to j, only using vertices from $\{1, \dots, k\}$. We want to find $f(\cdot, \cdot, n)$.

$$f(i, j, k) = \begin{cases} w(i, j) & k = 0, i \neq j \\ 0 & k = 0, i = j \\ \min\{f(i, j, k-1), f(i, k, k-1) + f(k, j, k-1)\} & \text{otherwise} \end{cases}$$

The algorithm is ommitted for brevity. We can run a similar set of optimizations we did for Bellman-Ford.
Memory Analysis The original memory is $\mathcal{O}(n^3)$, which can be made $\mathcal{O}(n^2)$ using bottom-up DP.
Runtime Analysis We triple for loop over k, i, j and then update the table in constant time every time. $\mathcal{O}(n^3)$ runtime.

5.2.3 More DP Examples

We consider the problem of longest increasing subsequence. We are given an input array A of length n. We want to find the longest subsequence (that is, non-contiguous elements in the same order as in the original array) that is increasing.

We again have to figure out a recurrence relation. At each index i, we have the choice of either taking that element into our subsequence or not. Furthermore, sometimes we cannot; in particular, if the last element of the subsequence is too large, taking i may cause it to stop increasing. Thus, we have the following recurrence relation, letting $f(i, last)$ be the length of the longest increasing subsequence of $A[i \dots n]$ such that all values we use are $\leq A[last]$. We want to find:

$$\max_{1 \leq i \leq n} \{f(i, i+1) + 1\}$$

Instead, we can use a sentinel trick to make this a bit easier. Prepend $-\infty$ to A. Now, $f(1, 2)$ is what we're solving for.

Now, for the actual recurrence relation:

$$f(last, i) = \begin{cases} 0 & i = n + 1 \\ f(last, i+1) & A[i] \leq A[last] \\ \max\{f(last, i+1), f(i, i+1)\} + 1 & \text{otherwise} \end{cases}$$

which can be done in $\mathcal{O}(n)$ with bottom-up dp, with runtime $\mathcal{O}(n^2)$.

5.3 Lecture 13

5.3.1 Knapsack Problem

We consider the knapsack problem. Given as input an array $A[1 \dots n]$, of items. Each item is a pair (w, v) (weight and value). We also have a knapsack (bag) that can hold at most W weight.

Example 5.1
There is a natural greedy approach to solving this.

$$\begin{aligned} A &= [(10.001, 10.002), (10, 10), (10, 10)] \\ W &= 20 \end{aligned}$$

We greedily take the most "bang for your buck," biggest ratio of value to weight. Greedy will take $(10.001, 10.002)$, while optimal is $[(10, 10), (10, 10)]$.
You can prove that you get at least half of the optimal solution with the greedy approach.

Algorithm 5.5 (Knapsack DP)
First, we again think about recursive brute force, and then form a recurrence relation. Define $f(i, C)$ is the maximum value that you can make with the last i items and capacity C. We want $f(1, W)$. The recurrence relation is:

$$f(i, C) = \begin{cases} 0 & i = n + 1 \\ f(i + 1, C) & w_i > C \\ \max\{f(i + 1, C), v_i + f(i + 1, C - w_i)\} & \text{otherwise} \end{cases}$$

Note that i ranges from 1 to n and C ranges from 0 to W, furthermore we only need the answers from $i + 1$ to get i. Thus, runtime is $\mathcal{O}(nW)$, memory is $\mathcal{O}(W)$ bottom-up.

This is pseudo-polynomial time. Why is that? Call b the amount of bits in the input of your algorithm. We define an algorithm that runs in polynomial time as one where the runtime is a polynomial in terms of b.

Note that the size of the input is $\Theta(n \log W)$ in bits (since every weight is at most W). This means that nW is polynomial in n and exponential in $\log W$!

5.3.2 Traveling Salesman Problem

Now consider the following problem. There are n locations with distances $D[i][j]$ = distance from i to j, which form a metric space (nonnegative and satisfy triangle inequality). We want to traverse this complete graph, visiting all verteices, starting at 1 while minimizing total travel.

The naive brute force way is to try every ordering of visiting the vertices, this is $\mathcal{O}(n!)$.

Algorithm 5.6 (TSP)
Define $f(i, S)$ as the min travel time to visit all locations in $S \subseteq \{1, 2, \dots, n\}$ when starting at 1. We want $f(1, \{2, 3, \dots, n\})$.

$$f(i, S) = \begin{cases} 0 & S = \emptyset \\ \min_{j \in S} f(j, S \setminus \{j\} + D[i][j]) & \text{otherwise} \end{cases}$$

Runtime Analysis We must calculate for $\mathcal{O}(n2^n)$ entries in each set, with each maximum taking at most linear time, giving us: $\mathcal{O}(n^2 2^n)$. **Memory** Top-down memoization is $\mathcal{O}(n2^n)$, but with bottom-up, it's something like

$$\max_{1 \le k \le n} n\binom{n}{k} = n\binom{n}{n/2} = \mathcal{O}(\sqrt{n}2^n)$$

The example of chain matrix multiplication was also discussed, but I have ommitted it here; DPV does a pretty good job of explaining it.

6 Linear Programming

6.1 Lecture 13, Continued

6.1.1 Linear Programs

Linear programming is maximizing or minimizing some linear function subject to some linear inequality constraints. In the language of linear algebra this is:

Definition 6.1 (Linear Program)
A linear program is an optimization problem which can be framed as:

$$\max_x c^T x : Ax \leq b$$

where the A, b, c are given, and the inequalities are elementwise.

This may seem abstract, so let us give an example of a linear program.

Example 6.1
I run a factory that sells foo for \$4/oz and bar for \$5/oz. In my warehouse, I have

- W = 10 oz water
- B = 6 oz butter
- G = 6 oz goo

Also, to make our products, we need to following ingredients:

- In order to make 1 oz of foo, it takes 2 oz water, 1 oz butter, 4 oz goo.
- In order to make 1 oz of bar, it takes 3 oz butter, 2 oz goo.

Call x the oz of foo I make, and call y the oz of bar I make. The objective is thus:

$$\max_{x,y} 4x + 5y$$

subject to:

$$2x + 0y \leq 10, 1x + 3y \leq 6, 4x + 2y \leq 6, -x \leq 0, -y \leq 0$$

Then define:

$$c = \begin{bmatrix} 4 \\ 5 \end{bmatrix}, b = \begin{bmatrix} 10 \\ 6 \\ 6 \\ 0 \\ 0 \end{bmatrix}, A = \begin{bmatrix} 2 & 0 \\ 1 & 3 \\ 4 & 2 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}$$

Note that this satisfies the definition of LP we gave above.

We briefly discuss convexity in the context of LP, as well as some other properties.

Definition 6.2 (Feasibility)
A linear program is feasible if there is at least one vector x that satisfies all the constraints.

Definition 6.3 (Bounded)
A linear program is bounded if its solution is not $+\infty$ or $-\infty$.

Definition 6.4 (Convexity)
A region is convex if for any two points in the region, the line between them is also in the region.

Theorem 6.1
The feasible region in any LP is always convex. Furthermore, the solution to an LP always lies on a vertex of this convex polytope.

Furthermore, note that all of these constraints are "halfspaces" i.e. they are on one side of a bunch of lines. This defines a convex polygon (or polytope in higher dimensional space) that we have to satisfy (a feasible region). We have to maximize in the c direction. One of the optimal solutions of this is always a vertex.

Even though the amount of vertices may be exponential, there exist polynomial-algorithms to solve LPs. We are not too concerned about their details or implementations; we are recommended to take 270. Instead, we just need to understand how to formulate our problems as LPs.

We briefly discuss duality. Remember in our example (without the nonnegative constraints) we had the following equations.

$$\begin{aligned} 2x + 0y &\leq 10 \\ 1x + 3y &\leq 6 \\ 4x + 2y &\leq 6 \end{aligned}$$

Suppose I wanted to prove to you that the maximum amount you could make was no more than 5 dollars. Multiply all the equations by some multipliers $z_1, z_2, z_3 \geq 0$ and add them:

$$(2z_1 + z_2 + 4z_3)x + (3z_2 + 2z_3)y \leq z^T b$$

What if $2z_1 + z_2 + 4z_3 = 4$ and $(3z_2 + 2z_3) = 5$. Then we have recovered our original c.

$$c^x \leq z^T b$$

this is the dual linear program (upper bound on our revenue).

Definition 6.5 (Duality)
Suppose our original "Primal" LP was:

$$p^* = \max_x c^T x : Ax \leq b$$

the dual LP is:

$$q^* = \min_y b^T y : A^T y = c, y \geq 0$$

Note that each y is an upper bound to $c^T x$; the dual tries to find the "tightest upper bound."

Theorem 6.2 (Weak/Strong Duality)
Weak duality tells us $p^* \leq q^*$.
Strong duality tells us that if the primal LP is feasible and bounded, so is the dual and $p^* = q^*$. Furthermore, if the primal is unbounded, then the dual is infeasible.

Also, the dual of the dual is the primal.

6.2 Lecture 14

We briefly list algorithms used to solve LPs (but not detail them). They include:

- Simplex (Dantzig '47), a greedy algorithm. Runs in exponential time worst-case; actually converges quickly in practice.
- Ellipsoid (Khachiyan '79), first poly-time algorithm for LP. For m constraints, n variables and L precision, this takes poly(m, n) $\cdot L$ time. Not as usable.
- Interior Point (Karmarkar '84) Poly-time, but usable.

Maximizing an LP is basically just as hard as finding a feasible x (suppose we could, we could just binary search to find an optimal point). Furthermore, we can write LPs in the "primal" (α) form in the "dual" (β) form.

Example 6.2
Suppose we start with the following example:

$$\min c^T x : Ax \leq b$$

Let us try to rewrite it in the dual form. First, we write $x = x^+ - x^-$ under the constraint $x^+, x^- \geq 0$. Now we have non-negativity. Now how do we get inequalities to turn into equalities? We do the following:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_j$$

We introduce "slack" variables s_j such that:

$$\sum_{j=1}^{n} a_{ij} x_j + s_j = b_j, s_j \geq 0$$

So our entire LP becomes:

$$\max -c^T (x^+ - x^-) : A(x^+ - x^-) + s = b, x^+, x^-, s \geq 0$$

Now let's go the other way. Suppose we start with:

$$\max b^T y : A^T y = b, y \geq 0$$

Then, note that the condition of equality is just equivalent to two inequality constraints. Thus we can write,

$$\min -b^T y : A^T y \leq b, A^T y \leq -b, -y \leq 0$$

6.2.1 Maximum Flow

Imagine you have some directed graph of "pipes" which each have a certain "capacity." There is a single source vertex s and sink vertex t that all the "water" comes from and leaves. What is the maximum rate of water flow this network can support from the source to the sink?

The above formulation is termed the $s - t$ formulation of the max flow problem.

The natural algorithm is the following greedy algorithm; Keep finding $s \rightarrow t$ paths iteratively (perhaps using DFS) and push flow amount that saturates the bottleneck (shortest) edge. This unfortunately does not find the correct solution. It will find a flow, but it might not be optimal.

This is actually a linear program! Our algorithms above are enough to solve it. Consider some flow $f \in \mathbb{R}^m$ (assign a number to each edge). There are certain constraints:

- $\forall e \in E, f_e \geq 0$ (Nonnegativity)
- $\forall e \in E, f_e \leq u_e$ (Capacity constraint)
- $\forall v \in V, \sum_{e=(v,\cdot)} f_e - \sum_{e=(\cdot,v)} f_e$ (Conservation of flow)
- We wish to maximize the flows: $\max \sum_{e=(s,\cdot)} f_e = \max |f|$.

The first correct algorithm that solved this in better than exponential time was Ford-Fulkerson in the 50s.

Algorithm 6.1 (Ford-Fulkerson)
This algorithm runs in pseudo-polynomial. It's polynomial in the max capacity U, number of vertices n, and number of vertices m.
For all edges $e = (x, y)$, we will pretend $(y, x) \in E$ with $u_{(y,x)} = 0$ (if it isn't present already). Furthermore, if there is f_e flow on e, we will pretend $-f_e$ flow on the reversed edge (i.e. if we push flow f_e, then the reversed edge will have capacity $0 + f_e$). This allows us to "undo" the decision at some point.
Now, we follow the normal greedy algorithm. Iteratively try to find an $s \longrightarrow t$ path in the "residual graph" until you can't (this path is called an "augmenting path").
Runtime Analysis In the original paper, they gave an example graph G with $|V| = 10$ and $|E| = 48$ such that the algoritihm doesn't terminate. However, it hinged on using irrational numbers as capacities. Let us suppose they are integers from 1 to U; we can then make an argument for termination. Note that this means that all the flows are going to always be integral; this means any capacity in the residual graph is an integer. This means when we find an augmenting path, we send at least 1 flow from s to t; this means at most we'll have to do a DFS the max-flow amount of times ($|f^*|$). This yields runtime:

$$\mathcal{O}((m+n)|f^*|) = \mathcal{O}((m+n)(n-1)U) = \mathcal{O}(n(m+n)U)$$

since in the worst case, there are $n - 1$ nodes coming out of s with capacity U each.
Using BFS instead of DFS leads to so some savings and an even better $\mathcal{O}(m^2 n)$. There are more improvements; we'll leave those for Wikipedia.
Proof of Correctness We again start with some prerequisite lemmas.

- Any $s \longrightarrow t$ flow can be decomposed into $\leq m$ cycles and $s \longrightarrow t$ paths.

  Proof By induction on m. The base case (with nonzero flow) is a one-edge graph with s to t ($m = 1$). So it decomposes into a single-edge path from $s \longrightarrow t$.

  We consider the inductive step where $m > 1$. Consider some flow f from s to t. Make a graph H where each edge exists if there is nonzero flow across that edge. Now let us do a DFS in H from s. There are a few possible cases.

  - If we reach t in this DFS, we found a path from $s \longrightarrow t$ in this graph. We look at the minimum flow edge along that path, and extract that path with that smallest value from f. Now, this graph has at least one fewer edge, so by inductive hypothesis, this decomposes into $m - 1$ paths and cycles; adding the original path back gives us all m paths and cycles.
  - If we find a cycle, then we can do the same thing as above; just extract the cycle.
  - We get stuck at some vertex $v \neq t$, this would mean there is no conservation of flow (since you can't "get out" of this vertex).

- Suppose f is a non-max flow in G. Then $f^* - f$ is a feasible flow in G_f.

These are enough to show the correctness of Ford-Fulkerson (every augmenting path gives a better flow which is feasible).

6.3 Lecture 15

Another proof of correctness of Ford-Fulkerson comes from the following results (which take some setup).

Definition 6.6 (Vertex Cut)
A cut of a graph G is a partition of V into two non-empty sets S and $V - S$.

Definition 6.7 ($s - t$ Cut, Capacity of a Cut)
An s-t cut is a cut where $s \in S$ and $t \in V - S$. We define the capacity of such a cut as:

$$u(S) = \sum_{e \in S \times (V - S)} u_e$$

Definition 6.8 (Net Flow)
Given an s-t cut S, we define the net flow $f(S)$ by:

$$f(S) = \sum_{e \in S \times (V - S)} f_e - \sum_{e \in (V - S) \times S} f_e$$

This means maximizing our flow means maximizing $f(\{s\})$. Furthermore, for all $s - t$ cuts, $f(S) = |f|$. Let's show this:

Proof
We induct on $|S|$. The base case is $|S| = 1 \implies S = \{s\}$. Note $f(\{s\}) = |f|$ by definition.

The inductive step has $|S| > 1$. There must be at least two vertices in S, s and v. Suppose we looked at v being in $V - S$ instead. Then there are some edges that go from the other elements of S to/from v. There may also be edges from/to vertices in T. For the first two cases, call the flow A and B; for the latter two cases, call the flow C and D. Note that by conservation of flow:

$$A + D = B + C \implies A - B = C - D$$

Furthermore:

$$f(S - \{v\}) = f(\text{stuff not with } v) + A - B$$

By the inductive hypothesis, the left-hand side is the flow, i.e. $|f|$. However,

$$f(S) = f(\text{stuff}) + C - D$$

So by our conservation equations, $f(S) = f(S - \{v\}) = |f|$.

Theorem 6.3 (Max-flow, Min-cut Theorem)
Consider a flow f and a cut S. We have

$$\max_f |f| = \min_S u(S)$$

This means the maximum flow is the same as the minimum cut capacity.

Proof
We will show $|f^*| \leq u(S^*)$ and $|f^*| \geq u(S^*)$.

To show the first claim, we will show for all flows f and cuts S, $|f| \leq u(S)$.

$$\begin{aligned}|f| &= f(S)\\ &= \sum_{a\in S, b\in(V-S)} f_{ab} - f_{ba}\\ &\leq \sum_{a\in S, b\in(V-S)} f_{ab}\\ &\leq \sum_{a\in S, b\in(V-S)} u_{ab}\\ &= u(S)\end{aligned}$$

as required.
To show the second claim, we will show that there exists a flow f and cut S such that $|f| = u(S)$, because $u(S) \geq u(S^*)$ and $|f| \leq |f^*|$. Let f be a max flow, and define

$$S = \{v : s \text{ has a path to } v \text{ in } G_f$$

We know

$$|f| = f(S) = \sum_{a\in S, b\in V-S} f_{ab} - \sum_{a\in S, b\in V-S} f_{ba}$$

However, every outgoing edge must be fully exhausted; otherwise it would be a part of S (reachable in G_f). Furthermore, any incoming edge to S cannot have nonzero capacity, because otherwise in the residual graph there would be an outgoing edge with nonzero capacity, meaning that more would be reachable than just S.s
Thus, we have:

$$|f| = \sum_{a\in S, b\in V-S} u_{ab} = u(S)$$

Thus we have shown both directions, so we are done.

The min-cut problem is very close to the dual of the max-flow problem. As a reminder we have the following constraints:

- $\forall e \in E, f_e \geq 0$ (Nonnegativity)
- $\forall e \in E, f_e \leq u_e$ (Capacity constraint)
- $\forall v \in V, \sum_{e=(v,\cdot)} f_e - \sum_{e=(\cdot,v)} f_e = 0$ (Conservation of flow)
- We wish to maximize the flows: $\max \sum_{e=(s,\cdot)} f_e = \max |f|$.

To massage them into the "primal" form, we have to convert all the constraints to $\leq$ constraints, which would be a lot of work. However, we use the same trick where we multiply by y_i multipliers on each side. Here is the more general derivation:

We start with the basic form of the constraints of the primal:

$$\begin{aligned}Ax \leq b &\iff \sum a_{1j}x_j \leq b_1, \sum a_{2j}x_j \leq b_2, \ldots\\ &\iff y_1 \sum a_{1j}x_j \leq b_1 y_1, \sum a_{2j}x_j \leq b_2 y_2, \ldots \text{for some } y_i \geq 0\\ &\iff x_1(a_{11}y_1 + a_{21}y_2 + \ldots) + x_2(a_{12}y_1 + a_{22}y_2 + \ldots) + \cdots \leq b^T y\end{aligned}$$

Where we want $A^T y = c$. Variables in the primal are constraints in the dual; constraints in the primal are variables in the dual.

Going back to the max flow problem, we try to reformulate the dual. Note that if the ith constraint was an equals instead of inequality, then no need for $y_i \geq 0$ constraint. Furthermore, for all the non-negativity constraints, flip them to be $-x_i \leq 0$ and multiply by some other dual variables to get: $-w_i x_i \leq 0$ and $w_i \geq 0$, so we need

$$a_{1i} y_1 + a_{2i} y_2 + \cdots - w_1 = c_i$$

but since we have the free parameter of w's, we can just write this constraint as:

$$A^T y \geq c$$

Let us now try writing the dual using the things we learned. Call u the capacities of the edges (which were $\leq$ constraints originally).

$$\min u^T y : A^T [yz] = c, y \geq 0$$

We claim when the y's and z's are integers this is just the min-cut problem.

Define:

$$y_{ab} = \begin{cases} 1 & a \in S, b \in V - S \\ 0 & \text{otherwise} \end{cases}$$

$$z_a = \begin{cases} 1 & a \in S \\ 0 & \text{otherwise} \end{cases}$$

We claim all the constraints are satisfied, with casework on a and b (omitted here).

6.4 Lecture 16

6.4.1 Games

Definition 6.9 (Game)
A game involves $p > 2$ players taking an action (in secret) from a set of possible actions. Once all the actions are taken, an outcome is revealed; each player gains some utility (score) for each possible outcome.

We first note the most well-studied game: the Prisoner's Dilemma.

Example 6.3
There are two apprehended suspects who the police are sure committed a crime and have a lesser charge, but they need a confession to get a greater charge. The police give the following deal to the suspects:

- If you stay silent: you get 1 year in prison.
- You tell on your partner: you get 0 years in prison, your partner gets 3 years.
- If both partners snitch: both get 2 years.

	snitch	silent
snitch	(-2, -2)	(0, -3)
silent	(-3, 0)	(-1, -1)

Thus, any rational prisoner will tell on their partner, since they only gain from their decision (hold the row constant; moving left is always better for the column player.)
We can think of this as an input to some algorithm, which might spit out the optimal strategy for the people.

Definition 6.10 (Zero-Sum Games)
A zero-sum game is a game where the payoffs for the 2 players sum to 0 for any outcome.

A good example of a zero-sum game is rock, paper, scissors. We can tabulate here in the following "payoff matrix."

	rock	paper	scissors
rock	(0, 0)	(-1, 1)	(1, -1)
paper	(1, -1)	(0, 0)	(-1, 1)
scissors	(-1, 1)	(1, -1)	(0, 0)

We can just write this as the 1st number then, since all cells sum to 0:

0	-1	1
1	0	-1
-1	1	0

The row player thus wants to maximize the score and the column player wishes to minimize the score.

Now we discuss strategies of playing the game.

Definition 6.11 (Strategies)
There are two main types of strategies.

1. Pure Strategies: Always picking some particular action.
2. Mixed Strategies: Some probability distribution over pure strategies.

Returning back to rock-paper-scissors, suppose player 1 plays mixed strategy:

$$x = (1/3, 1/3, 1/3)$$

Then suppose player 2 plays some other mixed strategy:

$$y = (y_1, y_2, y_3)$$

Then, the expected payoff (using the reduced payoff matrix U with one entry) is:

$$\mathbb{E}[\text{payoff}] = \sum_{j=1}^{3} \sum_{i=1}^{3} \frac{1}{3} U(i, j) y_j = \sum_{j=1}^{3} y_j(-1/3 + 1/3 + 0) = 0$$

Thus, we have the following.

Definition 6.12

- There exists a strategy x for the row player which guarantees $\mathbb{E}[\text{payoff}] \geq 0$.
- Similarly, there exists a strategy y for the column player which guarantees $\mathbb{E}[\text{payoff}] \leq 0$.

Thus, we say that the pair of strategies is in equilibrium. This means the value of this game is 0 (the value that can be guaranteed by both parties).

Let us look at more examples:

Example 6.4

	L	R
T	5	-3
B	-1	1

Let us try to find the equilibrium. What if the row player tries mixed strategy $x = (1/2, 1/2)$? Let's try pure strategies for the column player:

- L means the expected payoff is:
$$5\frac{1}{2} - 1\frac{1}{2} = 2$$
- R means the expected payoff is:
$$-3\frac{1}{2} + 1\frac{1}{2} = -1$$

Now, lets have the column player try mixed strategy $y = (1/2, 1/2)$. Let's again try pure strategies:

- T means the expected payoff is:
$$5\frac{1}{2} - 3\frac{1}{2} = 1$$
- B means the expected payoff is:
$$-1\frac{1}{2} + 1\frac{1}{2} = 0$$

Thus, these strategies are not in equilibrium, because -1 is the minimization of the expected payoff (what the column player seeks to do) and 1 is the maximization of the expected payoff (what the row player seeks to do). Here is a better strategy: $x = (1/5, 4/5)$. Pure strats for column player:

- L means the expected payoff is
$$5\frac{1}{5} - 1\frac{4}{5} = \frac{1}{5}$$

- R means the expected payoff is

$$-3\frac{1}{5}+1\frac{5}{5}=\frac{1}{5}$$

$y=(2/5,3/5)$ also guarantees at most $\frac{1}{5}$ payoff.
How can we find this magical pair of strategies that cause a payoff? Let us be more general.
If $x=(x_1,x_2)$, the strategy L gives payoff $5x_1-1x_2$ and R gives payoff $-3x_1+1x_2$. Thus, we want to find:

$$\begin{aligned}&\max_x \min\{5x_1-x_2,-3x_1+x_2\}:x_1+x_2=1\\&=\max_{x,z} z: z\le 5x_1-x_2, z\le -3x_1+x_2, x_1+x_2=1, x_1,x_2\ge 0\end{aligned}$$

Where we introduce slack variable z, which will be the min at optimum.
Note that we could write a similar LP for the column player. Get $y=(y_1,y_2)$, the strategy T gives payoff $5y_1-3y_2$ and the strategy B gives payoff $-y_1+y_2$. Thus we want to find:

$$\begin{aligned}&\min_y \max\{5y_1-3y_2,-y_1+y_2\}\\&=\min_{y,w} w: w\ge 5y_1-3y_2, w\ge -y_1+y_2, y_1+y_2=1, y_1,y_2\ge 0\end{aligned}$$

Taking Jelani's word for it, these two LPs are duals! This means that they have the same optimum; thus they both yield the equilibrium strategy.

We talk about some more duality facts that can help us see this connection:

- $x_i\ge 0$ in primal; the corresponding constraint with any x_i in dual is inequality (instead of =)
- $Ax=b$ in primal; the corresponding dual variable does not have a nonnegativity constraint.

There are a few more unanswered questions. Firstly, let's generalize what we've seen in the specific game example.

Suppose our utility matrix is $U\in\mathbb{R}^{m\times n}$

The row player seeks to once again:

$$\max_x \min_y \sum_{j=1}^{n}\left(\sum_{i=1}^{m} x_i U(i,j)\right) y_j$$

note that the RHS is also $y^T Ux$. Note that this is just a weighted sum of y_j, where the y_j's all sum to 1. To minimize, we just put all the probability on the smallest column. However, this is just a pure strategy. Thus, we have shown that the best response to a fixed mixed strategy is a pure strategy.

This is also just an LP, so we can find them in polynomial time via linear programming. By duality, we have:

Theorem 6.4 (Minimax Theorem)
For any zero-sum game U, we have

$$\max_x \min_y \sum_{j=1}^{n}\left(\sum_{i=1}^{m} x_i U(i,j)\right) y_j=\min_y \max_x \sum_{i=1}^{m}\left(\sum_{j=1}^{n} y_j U(i,j)\right) x_i$$

6.5 Lecture 17

6.5.1 Online Decision-Making

We illustrate what making an "online decision" is with an example. Basically, new information is being added incrementally, and every step that we get new information, we must make some sort of decision.

Example 6.5 (Rain/Umbrella)
Suppose there are some experts $1, 2, 3, \dots, n$ that predict the rain over T days. We say a 1 if the weather prediction was wrong and 0 if the weather prediction was right.

experts	day 1	loss tally	day 2	tally	
1	1	1	1	2	...
2	1	1	0	1	...
$\vdots$					

How can we figure out on some day t whether it's going to rain or not? Which "expert" do we trust?

We define some notation.

- n experts, T days.
- We follow expert $i_t \in [n]$ on day $t \in \{1, \dots, T\}$.
- Expert i incurs loss of $\ell_i^{(t)}$ on day t.
- We define $\ell_i^{(t)} \in [0, 1]$, (not just a binary decision).
- Our total loss is $L = \sum_{t=1}^{T} \ell_{i_t}^{(t)}$.

So we want to minimize L.

Ideally, we want to compare our loss L to the minimum loss of $\sum_t \min_{i \in [n]} \ell_i^{(t)}$. However, this is not a realistic goal. If each day it rains or not randomly with probability $p = \frac{1}{2}$, no matter what determinism is used, $\mathbb{E}[L] = \frac{T}{2}$ (but the minimum loss is 0; suppose one expert always says it doesn't rain and one says it always does).

Instead, we will settle for minimizing:

$$R = L - \min_{i \in [n]} \sum_{t=1}^{T} \ell_i^{(t)} = L - L^*$$

where R is termed as "regret." We assume that the "adversary" that reveals the weather is strong enough to know our algorithm for decision-making, but cannot know the output of random calls.

We give a few inefficient strategies before stating optimum.

1. For all t, $i_t = 1$. Here, clearly, $R \geq T$.
2. Follow the majority opinion of experts. Again, we can have $R \geq T$.
3. Each day, listen to a uniformly random expert. Now, our regret (in expectation) is a little bit better.

$$\mathbb{E}[R] = \left(1 - \frac{1}{n}\right) T$$

To prove the upper-bound:

$$\begin{aligned}
\mathbb{E}\left[L\right] &= \sum_{t=1}^{T}\sum_{i=1}^{n}\frac{1}{n}\ell_i^{(t)} \\
&= \sum_{t=1}^{T}\min_i \ell_i^{(t)} + \frac{1}{n}\sum_{j\neq i}\ell_j^{(}t) \\
&\leq L^* + \frac{n-1}{n}T \\
\mathbb{E}\left[R\right] = \mathbb{E}\left[L\right] - \mathbb{E}\left[L^*\right] &= \left(1-\frac{1}{n}\right)T
\end{aligned}$$

4. Follow the expert who has the lowest loss so far.

$$\mathbb{E}\left[R\right] \geq \left(1-\frac{1}{n}\right)T$$

The proof is by adversarial example.

experts	$\ell^{(1)}$	tally	$\ell^{(2)}$	tally	$\ell^{(3)}$	tally
1	0	0	1	1	0	1
2	$\frac{1}{3}$	$\frac{1}{3}$	0	$\frac{1}{3}$	1	$\frac{4}{3}$
3	$\frac{2}{3}$	$\frac{2}{3}$	0	$\frac{2}{3}$	0	$\frac{2}{3}$

Note that we get maximum loss every day except for the first day, meaning our loss is $T-1$. However, the best expert, has loss $L^* = \frac{T}{n}$, so $R \approx \left(1-\frac{1}{n}\right)T$

Now we introduce the best strategy. We need both look at lowest loss, while also being randomized so our adversary can't just pick the one we're picking. This probability distribution needs to depend on the history of losses, and needs to be updated due to the online nature of the problem.

Algorithm 6.2 (Hedge/Multiplicative Weights Algorithm)
Each day will have a probability distribution $x^{(t)}$ on the experts. This means that every day, our expected loss on day t,

$$L_t = \sum_{i=1}^{n} x_i^{(t)}\ell_i^{(t)}$$

And we redefine L in terms of expectation, i.e.

$$L = \sum_{t=1}^{T} L_t, R = L - L^*$$

We also define weights for each expert and timestep, i.e. $w_i^{(t)}$.
The algorithm to evolve this distribution is as follows:

- We choose a parameter $\epsilon \in (0, \frac{1}{2}]$.
- We have a uniform prior on all the experts, i.e. $w_i^{(1)} = 1$ for all $i \in [n]$
- Define $W^t = \sum_{j=1}^{n} w_j^{(t)}$
- $x_i^{(t)} = \frac{w_i^{(t)}}{W^t}$
- $w_i^{(t+1)} = w_i^{(t)}(1-\epsilon)^{\ell_i^{(t)}}$

We can interpret the last line as a "Bayesian update."

We now claim how good the hedge algorithm is.

Theorem 6.5
Choosing parameter ϵ, then hedge achieves,

$$\mathbb{E}\left[R\right] \leq \epsilon T + \frac{\ln n}{\epsilon}$$

which is minimized by choosing $\epsilon = \sqrt{\frac{\ln n}{T}}$, which means

$$\mathbb{E}\left[R\right] \leq 2\sqrt{T \ln n}$$

Proof
We have two pre-requisite lemmas.

- $W_{T+1} \geq (1-\epsilon)^{L^*}$

$$\begin{aligned} W^{(T+1)} &= \sum_{i=1}^{n} w_i^{(T+1)} \\ &\geq w_{i^*}^{(T+1)} \\ &\geq \prod_{t=1}^{T} (1-\epsilon)^{\ell_{i^*}^{(t)}} \\ &= (1-\epsilon)^{\sum_t \ell_{i^*}^{(t)}} \\ &= (1-\epsilon)^{L^*} \end{aligned}$$

- $W_{T+1} \leq n \prod_{t=1}^{T} (1-\epsilon L_t)$ We need to show $W^{(t+1)} \leq W^{(t)}(1-\epsilon L_t)$. If this were true, then by induction:

$$W^{(T+1)} \leq W^{(1)} \prod_{t} (1-\epsilon L_t) = n \prod_{t=1}^{T} (1-\epsilon L_t)$$

Note that:

$$\begin{aligned} W^{(t+1)} &= \sum_{i=1}^{n} w_i^{(t+1)} \\ &= \sum_{i=1}^{n} w_i^{(t)} (1-\epsilon)^{\ell_i^{(t)}} \\ &\leq \sum_{i=1}^{n} w_i^{(t)} (1-\epsilon \ell_i^{(t)}) \\ &= \sum_{i=1}^{n} x_i^{(t)} W^{(t)} (1-\epsilon \ell_i^{(t)}) \\ &= W^{(t)} \sum_{i=1}^{n} x_i^{(t)} (1-\epsilon \ell_i^{(t)}) \\ &= W^{(t)} \left(\sum_{i=1}^{n} x_i^{(t)} - x_i^{(t)} \epsilon \ell_i^{(t)} \right) \\ &= W^{(t)} (1-\epsilon L_t) \end{aligned}$$

So now we have that:

$$\begin{aligned}
(1-\epsilon)^{L^*} &\le n\prod_{t=1}^{T}(1-\epsilon L_t) \\
L^*\ln(1-\epsilon) &\le \ln n + \sum_{t=1}^{T}\ln(1-\epsilon L_t) \\
L^*\ln(1-\epsilon) &\le \ln n + \sum_{t=1}^{T}\ln(1-\epsilon L_t) \\
L^*(-\epsilon-\epsilon^2) &\le \ln n - \epsilon\sum_{t=1}^{T}L_t \\
L^*(-1-\epsilon) &\le \frac{\ln n}{\epsilon} - L \\
L - L^* &\le \frac{\ln n}{\epsilon} + \epsilon L^* \\
R &\le \frac{\ln n}{\epsilon} + \epsilon T
\end{aligned}$$

where we noted $\forall z \in [0, \frac{1}{2}], -z - z^2 \le \ln(1-z) \le -z$

There is also a connection to zero-sum game. Using weak duality and the fact that in the hedge algorithm,

$$\lim_{t\to\infty} R = \frac{2\sqrt{T\ln n}}{T} \to 0$$

we can prove the minimax theorem (instead of invoking strong duality). Furthermore, we can find the equilibrium by running the hedge algorithm on the original game, just treating payoffs as negative losses.

Algorithm 6.3 (Zero-sum Game Hedging)
The row player starts with $x^{(1)} = (\frac{1}{m}, \frac{1}{m}, \dots, \frac{1}{m})$ and the column player starts with $y^{(1)} = (\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n})$, and we do the updates in both according to losses: $\ell^{(t)} = -Ay^{(t)}$ and $\ell(t) = A^{\perp}x^{(t)}$ (i.e. the payoffs). As $t \to \infty$, we converge to the equilibrium, optimal strategy for both players.

7 Reductions

7.1 Lecture 18

7.1.1 Cook Reductions

We define algorithmic reduction, which is very useful for solving an extended set of problems.

Definition 7.1 (Reduction)
Given problems A, B, we say $A \rightarrow B$ or "A reduces to B" if the existence of an efficient algorithm for B implies there exists an efficient algorithm for A.

Given a reduction $A \rightarrow B$, there is good and bad we can prove from it.

- If we know how to solve B efficiently, then we can solve A efficiently.
- If we know A has no efficient algorithm, then neither does B.

There are many types of reductions. We discuss one of them:

Definition 7.2 (Cook Reduction)
Given a polynomial Time Alg_B that solves B, we should be able to call it (possibly multiple times) to create a solution to A that also runs in polynomial time.

In general our roadmap is:

$$a \longrightarrow \boxed{Pre} \longrightarrow b \longrightarrow \boxed{Alg_B} \longrightarrow o \longrightarrow \boxed{Post} \longrightarrow o_A$$

We turn a, an instance of problem A into b, and instance of B in polynomial time with a preprocessing algorithm. Then, we run it through Alg_B, then run the output through a postprocessing algorithm to get a solution for A.

7.1.2 Maximum Bipartite Matching

We consider the problem of maximum Bipartite maching. As an input we get an undirected Bipartite graph with two classes L and R where all the edges go between the two classes. We define

Definition 7.3 (Matching)
A bipartite matching $M \subseteq E$ is a subset M such that no two edges in M share a common vertex.

Thus, we try to find the M such that $|M|$ is as large as possible. To solve this problem, it actually reduces to (integral) max flow!

Algorithm 7.1 (Max Bipartite Matching Reduction)
For the preprocessing, suppose the instance a looks like on the left. We turn it into the directed one on the right:

$$a \longrightarrow \boxed{Pre} \longrightarrow b \longrightarrow \boxed{Alg_B} \longrightarrow o \longrightarrow \boxed{Post} \longrightarrow a$$

We constrain all the flows to be integer and run maximum flow on the graph (there is always an integral max flow). Now, we claim we can extract a matching from any integral flow (there is a bijective correspondence). In particular:

- matching M maps to f_M s.t. $|f_M| = |M|$
- flow f maps to M_f s.t. $|f| = |M_f|$

Knowing this, we know that the best matching will map to some flow which must be the max one, and from any flow we can extract a matching, so therefore the max flow must yield a max matching. We prove this claim.

Proof
Suppose $M = \{e_1, \ldots, e_R\}$. Then

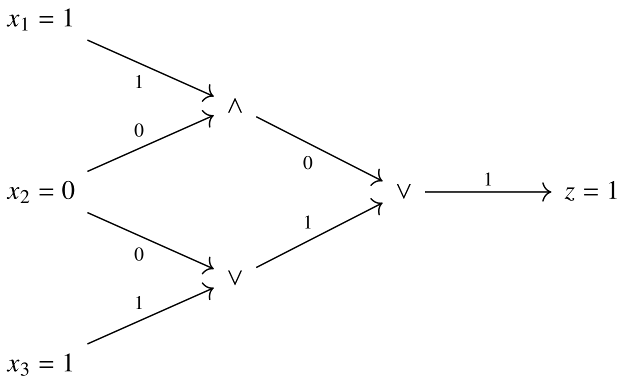
$$(f_M)_e = \begin{cases} 1 & \text{if } e = (s{,}a) \text{ and } a \text{ is matched in } M \\ 0 & \text{if } e = (s{,}a) \text{ and } a \text{ is not matched in } M \\ 1 & \text{if } e = (b{,}t) \text{ and } b \text{ is matched in } M \\ 0 & \text{if } e = (b{,}t) \text{ and } b \text{ is not matched in } M \\ 1 & \text{if } e \in M \\ 0 & \text{if } e \notin M \end{cases}$$

this can be seen graphically to have the same value as the matching ($|M|$ endpoints on each class that connect to S or T and the other class, flowing 1 each).
Now, consider some flow f. We know from our previous analyses that f decomposes into a collection of cycles and s-t paths. We know there are no cycles, so there is only the latter. Thus, all the paths must be flowing exactly one unit of flow. One of these paths looks like $s \longrightarrow a \longrightarrow b \longrightarrow t$. To construct M_f, place all the (a, b) in this form into the matching.

Runtime The runtime of this is as follows. Note that constructing the flow network from the original graph and reading off the flows takes linear time. Thus, we only care about the Ford-Fulkerson runtime, $\mathcal{O}(mn)$.

7.1.3 Circuit Value Problem

We are given a Boolean circuit C with n input bits $x_1, \ldots, x_n \in \{0, 1\}$ and one output bit z. We want to compute z. Such an input is a DAG, and it may look like this:



Thus to solve the problem, it can intuitively be seen that the following algorithm suffices, given a circuit C,

1. Topologically sort C.
2. Simulate the circuit from left to right to compute values at intermediary gates in this order.

This is linear-time algorithm. However, the circuit value problem is pretty powerful–it is what runs in our computers! In fact, any problem that can be solved in polynomial time (in $A \in \mathcal{P}$) has an "efficient, log-space" reduction to CVP (we define $\mathcal{P}$ at length in the next lecture; log-space is covered in courses like CS172). Here is a brief proof sketch.

Proof (Sketch)
We know there exists a poly-time algorithm Alg_A for A. Given an instance a for A, $Alg_A(a)$ uses $\leq T = \text{poly}(n)$ memory and time. Then, make T circuits, one for each time step. The outputs of C_t are inputs for C_{t+1}, where each change in memory state at timestep t is tracked by circuit C_t (a computer implements this as just boolean gates).

Furthermore, note that reductions are transitive, $A \longrightarrow B \longrightarrow C$ implies $A \longrightarrow C$. Now, we will show that CVP reduces to Linear Programming, which implies every poly-time problem reduces to LP.

Theorem 7.1
CVP reduces to LP.

Proof
Have one variable per gate and per input bit (treat them as constant-value gates). Let z_g is a variable that will store the value of gate g. The constraints we use are as follows:

- $\forall g, 0 \leq z_g \leq 1$
- $\forall$input gates where "$x_j = b$", $z_j = b$
- If we have g_1 and g_2 feeding into AND gate g, we add constraints

$$z_g \leq z_{g_1}$$
$$z_g \leq z_{g_2}$$
$$z_g \geq z_{g_1} + z_{g_2} - 1$$

- Similar constructions can be made for NOT and OR gates.
- We do not care about the objective, only feasibility, i.e. z (the output) will be set by the LP, which is correct.
- Note that all constraints force integrality, so there is no issue there.

7.1.4 Matrix Inversion/Multiplication

We will show a two-sided reduction between n-by-n Matrix Multiplication (MM) and Matrix Inversion (MI).

Algorithm 7.2 (Matrix Inversion/Multiplication Reduction)
First, we reduce MM $\longrightarrow$ MI. We want AB. Preprocess

$$M = \begin{bmatrix} I & -A & 0 \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}$$

Then, if we compute the inverse, it is

$$M^{-1} = \begin{bmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}$$

Then, in postprocessing, spit out the top right block of M^{-1}. We see that both preprocessing and postprocessing take linear $\mathcal{O}(n^2)$ time (super efficient!).

Now, we reduce MM → MI. By Gaussian Elimination, we can write any matrix as the product of a lower triangular matrix and an upper triangular one:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} I & 0 \\ CA^{-1} & I \end{bmatrix} \begin{bmatrix} A & B \\ 0 & D - CA^{-1}B \end{bmatrix}$$

Calling $S = D - CA^{-1}B$ and $Y = CA^{-1}$, then:

$$M^{-1} = \begin{bmatrix} A^{-1} & -A^{-1}BS^{-1} \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -Y & I \end{bmatrix} = \begin{bmatrix} A^{-1} - ZY & Z \\ -S^{-1}Y & S^{-1} \end{bmatrix}$$

by calling $Z = -A^{-1}BS^{-1}$. Therefore, in order to invert an n-by-n matrix, it suffices to invert 2 $n/2$-by-$n/2$ matrices plus $\mathcal{O}(1)$ MMs and Matrix additions/subtractions. Suppose MM ran in time $\mathcal{O}(n^{\omega})$. This means we have

$$T(n) \leq 2T(\frac{n}{2}) + \backslash^{\omega} = \mathcal{O}(n^{\omega})$$

Again, we have a linear-time reduction!

In fact, it turns out most problems in linear algebra reduce to matrix multiplication. It is kind of the central problem in linear algebra algorithms.

8 "Hard Problems" and NP-Completeness

8.1 Lecture 19

8.1.1 Search Problems

As a reminder note that if $A \to B$ in the Cook sense, then it implies two things:

- If B is "easy", then A is "easy".
- If A is "hard", then B is "hard".

i.e. this means that A is at least as hard as B.

Definition 8.1 (Boolean Problems)
Let $R \subset \{0,1\}^* \times \{0,1\}^*$. Consider some element $(x, w) \in R$. Then we call x the "instance" and w the "witness".
We can assign to each (x, w) "instance/solution" a True or False.

Armed with this R, we will look at two problems:

1. Decide(R): Given x, does there exist a w such that $(x, w) \in R$?
2. Search(R): Given x, return one w such that $(x, w) \in R$ (or detect if no such w exists).

Clearly, Decide(R) → Search(R) (if you have an oracle that can solve search, clearly it can solve decide). It turns out search can also be reduced to decide; first run the oracle on R, then run the oracle on the subset $Q = \{(x, w) : w \text{has 0 as its first character}\}$, then choose another decision (i.e. using binary search).

Example 8.1 (Max Flow)
If G is the graph, along with source s and sink t. We define $x = \langle G, s, t\rangle$ and $w = f^*$ (or really their binary encodings).
Decide(R): for any x, the answer is yes (any graph has a max flow.
Search(R): algorithm to find f^*.

Now, the question we ask is: does there always exist an algorithm to solve Decide(R)? The answer is no, this would be solving the famous halting problem. In particular, it corresponds to the instance $x = \langle P\rangle$. $(x, w) \in R_{halt}$ if and only if P halts when it runs.

8.1.2 P/NP

We consider problems R with an efficient verfier, V_R which is an algorithm where:

$$V_R(x, w) = \begin{cases} 1 & (x, w) \in R \\ 0 & (x, w) \notin R \end{cases}$$

By efficient, we have that V_R runs in time poly($|x|$) (Polynomial in the length of x).

Note 8.1
Any such R can be decided (or even searched) in $\leq \exp(\text{poly}(|x|))$ time. This is because such a V_R exists, we can brute force search over all w and try to find one such that $V_R(x, w) = 1$.

Now, we define the complexity classes $\mathcal{P}$ and $\mathcal{NP}$.

Definition 8.2 ($\mathcal{P}$ and $\mathcal{NP}$)

- $\mathcal{P}$ means Polynomial-time. A language $L \subseteq \{0,1\}^*$, $L \in \mathcal{P}$ if there exists an algorithm solving $x \in L$ in time poly($|x|$) (decided in polynomial time).
- $\mathcal{NP}$ means Nondeterministic Polynomial-time. Nondeterministic refers to a Nondeterministic Turing Machine, where branching can happen in parallel with unlimited parallelism. A problem $R \in \mathcal{NP}$ if there exists a poly($|x|$)-time verifier V_R such that for all x:

  If there exists poly($|x|$)-sized w such that $(x, w) \in R$, then $V_R(x, w)$ is True, otherwise $V_R(x, w)$ is False.

Intuitively, NP means that you can verify solutions efficiently. Note that:

Theorem 8.1 (P in NP)
$\mathcal{P} \subseteq \mathcal{NP}$

Proof
Consider some $L \in P$. Then, define $R = L \times \{0,1\}^*$. There is an algorithm A to solve if something is in L in polynomial time. The question $(x, w) \in R$ is the same as solving if $x \in L$, this means that we can run A as a verifier on just the x argument, meaning that $R \in \mathcal{NP}$.

We also have a few related notions:

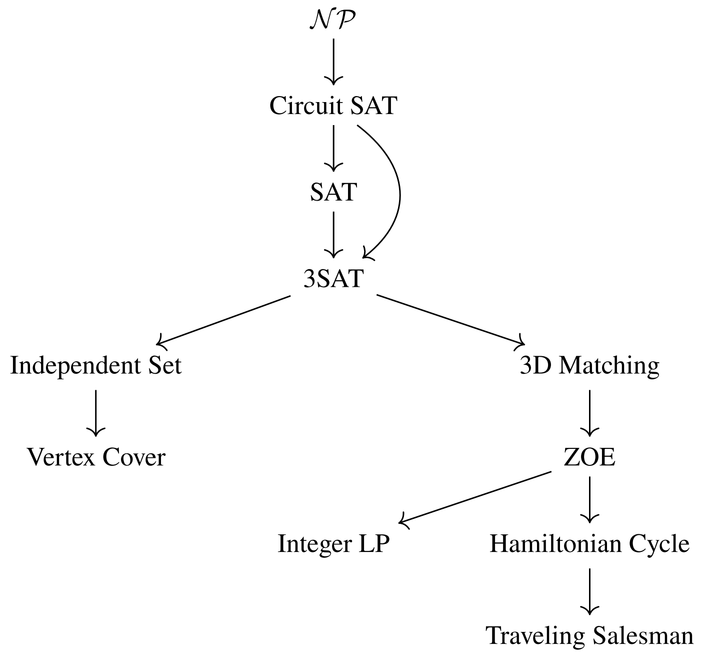
Definition 8.3 (NP-Hard)
A problem R is NP-hard if $\forall B \in \mathcal{NP}$, then $B \rightarrow R$. (i.e. it's at least as hard as everything in NP, but it could be outside NP).

Definition 8.4 (NP-Complete)
A problem R is NP-complete if $R \in \mathcal{NP}$ and R is NP-hard. (i.e. the hardest problems in NP).

However, how do you show that something is NP-hard? You need to show there's a reduction from every NP problem to it. It's a little bit easier if we start with a problem that every NP problem reduces to, and then show it reduces to another problem (we can then conclude that other problem is also NP-hard).

8.1.3 CSAT, SAT, 3SAT, Independent Set

Here is our roadmap of reductions, with forward arrows if a problem reduces to another.



We define the circuit SAT problem as follows.

Definition 8.5 (Circuit SAT (CSAT))
Define:

$$R = \{(x, w)\}, \langle C\rangle, (x, w) \in R \text{ if } C(w) = 1$$

where C is a circuit with AND, OR, NOT gates.

Let us show CSAT is NP-complete.

Proof
First, clearly it's in NP: an efficient verifier is V_{CSAT}; given C, w, the verifier just needs to evaluate the input on C.

Secondly, we will show CSAT is NP-hard. We will use a handwavy argument; CS172 has more rigorous arguments. Suppose $B \in \mathcal{NP}$. This means there exists an efficient verifier V_B. Preprocess (V_B, x) (a program and instance) to create a circuit C_B such that $C_B(w) = V_B(x, w)$. Then, feed C_B to the CSAT algo. We have reduced B to CSAT.

Definition 8.6 (SAT)
The input is a "formula" in Conjunctive Normal Form (CNF). It looks something like this:

$$\phi(x) = (x_1 \vee \overline{x_7}) \wedge (x_8) \wedge (x_2 \vee x_1 \vee x_3 \vee \overline{x_4}) \wedge \ldots$$

The SAT question asks if there is a setting of x that makes $\phi(x)$ true? The witness w is this assignment of x.

Now we claim SAT is NP-complete.

Proof
First, clearly it's in NP: an efficient verifier is just evaluating the formula after plugging in the w (the values of x_i's).

Secondly, we show it is NP-hard. Note that reductions are transitive; if we show CSAT reduces to SAT, then all NP problems reduce to SAT (so it is NP hard). In a CSAT Circuit, assign to every wire a variable. For every True, the variable z coming out of it is the clause (z); for False it is $(\overline{z})$. For every OR gate, if there x and y coming in and z coming out, $(z \vee \overline{x}) \wedge (\overline{z} \vee \overline{y}) \wedge (\overline{z} \vee x \vee y)$. You can check the truth table matches up with OR. There are similar decompositions for NOT and AND (omitted for brevity). Furthermore, the output gate O needs to be True. Thus, CSAT reduces to clauses in SAT.

Another claim is that SAT reduces to 3SAT (the version of the problem where every clause has size 3). This is not here (not sure why lol)

We now define the Independent Set Problem.

Definition 8.7 (Independent Set)
An Independent Set is a subset $S \subseteq V$ such that no two elements in S are neighbors.

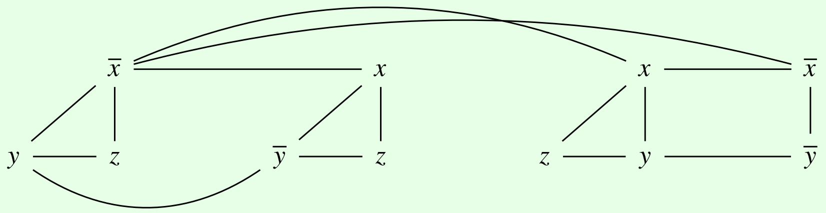
Definition 8.8 (Independent Set Problem (IS))
Given an undirected graph G and an integer k, does there exist an Independent set of size $\geq k$?

Let us show the reduction from 3SAT to ISET (by example).

Example 8.2
Let this be our 3SAT instance:

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

Then, we define the IS instance as follows:



(Suppose all negations were neighbors). Now, solving 3SAT reduces to checking IS on this graph with k = the number of variables. Doing this means that we've reduced 3SAT to IS (and this can be done generally). Thus IS is NP-complete.

8.2 Lecture 20

8.2.1 Vertex Cover, Clique

We now consider the Vertex Cover problem. First we define what a vertex cover is.

Definition 8.9 (Vertex Cover)
A vertex cover is a subset of vertices $S \subseteq V$ where $\forall e \in E, \exists v \in S$ such that e has v as one of its endpoints.

Now, the Boolean Vertex Cover problem is focused on:

$$\textsc{VertexCover} = \{\langle G, k \rangle : \exists \text{ vertex cover of } G \text{ of size} \leq k$$

The reduction then follows from the following claim:

Theorem 8.2
S is a vertex cover if and only if $V \setminus S$ is an Independent set.

Proof
First, assume S is a vertex cover. Suppose $V \setminus S$ was not an independent set. This would mean two vertices $a, b \in V \setminus S$ are neighbors. However, this would mean that (a, b) is not covered by S, which is a contradiction, since S is a vertex cover. Thus, by contradiction, $V \setminus S$ must be an independent set.
The other direction is similar.

This means that the reduction can be summarized as the following. Given an input $\langle G, k \rangle$ to independent, we will reduce to vertex cover with input $\langle G, n - k \rangle$. This will find if there exists a vertex cover of size $\leq n - k$, which if there is, can be converted to an independent of size $\geq n - (n - k) = k$. Thus, the reduction was successful.

Now, we consider the clique problem.

Definition 8.10 (Clique)
A clique is a set of vertices that have all possible edges between them in the graph.

Now, the Boolean Clique problem asks:

$$\text{CLIQUE} = \{\langle G, k \rangle : G \text{ has a clique of size} \geq k$$

One can make a similar observation. Note that any independent set in G is the same as any clique in the complementary graph (i.e. with the edges not present added and the ones not present removed).

8.2.2 3D Matching

We now look at a generalization of max bipartite matching. In 3D matching, we use triples. Given n boxes, cars, and drivers,

$$b_1, \ldots, b_n, c_1, \ldots, c_n, d_1, \ldots, d_n$$

Let $T = B \times C \times D$. The 3D Matching problem asks, does there exist a subset of T which covers each box, car, and driver exactly once.

Consider a special case of 3SAT. Consider some instance ϕ. It is true if ϕ is satisfiable 3SAT instance, and each literal in ϕ occurs ≤ 2 times. We will show 3SAT reduces to this (3SATL2) and then we reduce to 3D matching.

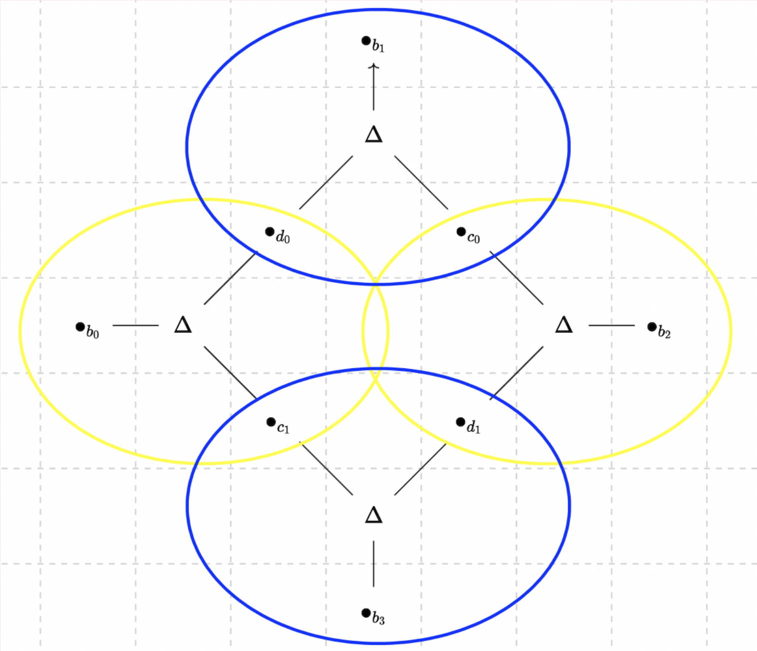
Algorithm 8.1 (Reduction 3SAT to 3SATL2)
Say ϕ is a 3SAT instance on n variables $x_1, \ldots, x_n$. First, we can simplify ϕ by removing any variable $x_1, \ldots, x_n$ that only appears positive or negative. Suppose some literal x that appears $t > 2$ times (positive or negative). Then, we create t new variables $y_1, \ldots y_t$ and replace the ith occurrence of x with y_i. Now, to enforce these variables to be equal, we just add the clauses:

$$(y_1 \vee \overline{y}_2) \wedge (y_2 \vee \overline{y}_3) \wedge \cdots \wedge (x_n \vee \overline{x}_1)$$

Note that now, each of the y_i appears at most twice (counting positive and negative separately), so we have reduced to an equisatisfiable form.

Now, we show the 3SATL2 → 3DM.

Algorithm 8.2
Given ϕ on variables $x_1, \ldots, x_n$ with clauses $R_1, \ldots, R_m$. For each x_i, draw the following diagram.



Each triangle is a triple and each dot is a box, car, or driver. If we want to cover all cars and drivers, must either take blue triples (call this $x_i = False$) or yellow triples (call this $x_i = True$). Now, repeat that "picture" n times. Furthermore, we need to add more boxes, cars, drivers, and triples to ensure that the clauses are also followed. Consider some clause R, for example,

$$R = (x \vee \overline{y} \vee z)$$

Then, we introduce a new car c_R, driver d_R, and 3 new triples (one per literal in the clause). For our example we use:

- x - add triple (b_{x_0}, c_R, d_R) or $b_{x_2}, c_R, d_R)$
- $\overline{y}$ - add triple (b_{y_1}, c_R, d_R) or $(b_{\overline{y}_3}, c_R, d_R)$
- z - add triple (b_{z_0}, c_R, d_R) or (b_{z_2}, c_R, d_R)

this is why we said we needed 3SATL2, because our diagram can only suppose 4 boxes per variable (each variable can only participate in 2 clauses positively and 2 negatively). However, you might have some leftover boxes, if each variable participates in less than 2 clauses. Note that there are $4n$ boxes introduced (from the diagrams), and we matched $2n + m$ of the cars/drivers. Thus, there are $4n - (2n + m) = B$ boxes left. Thus, we can introduce B new cars and drivers who are willing to take any box. We then match the last B boxes with these "dummy" cars. Now, we have turned our 3SATL2 instance into a 3DM instance on $4n$ cars, boxes, and drivers. Thus, we have completed an efficient reduction.

8.2.3 ZOE, Integral LP

We now turn to the problem of systems of zero-one equations. We are given $A \in \{0,1\}^{m\times n}$. The ZOE problem asks if there exists $x \in \{0,1\}^n$ such that $Ax = \mathbf{1}$.

We state that there is a reduction from 3DM to ZOE.

Algorithm 8.3
Take our instance $b_1, \ldots, b_n, c_1, \ldots, c_n, d_1, \ldots, d_n$ where there are m triples among the items (3-cliques). Define

$$A_{ij} = \begin{cases} 1 & \text{if } i \text{ row participates in } j\text{th triple} \\ 0 & \text{otherwise} \end{cases}$$

Note that $A \in \{0,1\}^{3n\times m}$. Then, finding a solution to this ZOE instance leads to what follows. Consider some box b which is at row i. After doing the matrix multiplication, we have $\sum_{T \in \{\text{triples containing b}\}} x_T = 1$, i.e. exactly one of these triples must exist. This is exactly what 3DM wants, so solving the ZOE instance finds us a matching such that exactly one triple exists for each item.

Now, we discuss integral linear programming, which is linear programming with added integrality constraints (require some variables to be integers). The decision problem writes:

$$\begin{aligned} \exists x?\ c^T x &\geq k \\ Ax &\leq b \\ \forall i \in S, x_i &\in \mathbb{Z} \end{aligned}$$

Reducing ZOE to Integer LP is not too hard. We make an Integer LP with the following objective and constraints.

- $k = -\infty$ (we only care about feasibility)
- Constraints: $Ax = \mathbf{1}$
- $\forall i, 0 \leq x_i \leq 1$
- S is the set of all variables.

8.3 Lecture 21

8.3.1 Coping with NP-Hardness

There are a few exact methods to solve NP-hard problems.

- Faster exponential-time algorithms.
- Backtracking (doing a brute-force search, but ending a recursive branch early if you know the assignments created will be unfeasible)
- Branch and bound: cut off subtrees of a recursive brute-force search early if you can prove that any completion of a current partial solution cannot be optimal.

Branch and bound can be used for approximation as well, where we replace "be optimal" with "be 1% worse than optimal.

There are also heuristic methods to solve these problems.

- Local search: start with a random solution, then iteratively make "local" adjustments greedily to make it better until you cannot anymore.
- Simulated annealing: the same as local search, but allow yourself with a (small) probability to make local adjustments that increase cost.

8.3.2 Approximation Algorithms for Vertex Cover

Finally, there are also approximation algorithms. These are heuristics with provable guarantees of closeness to OPT. For example, for minimization problems, we want to find a solution of cost $\leq \alpha \cdot OPT$ for an approximation ratio $\alpha \in [1, \infty)$. For maximization problems, you want the same thing, except solution of cost $\geq \beta \cdot OPT$ for an approximation ratio $\beta \in (0, 1]$.

Example 8.3
For example, take the vertex cover problem. This is finding $S \subseteq V$ such that each $e \in E$ is incident upon at least one $v \in S$. We wish to minimize $|S|$.
The first observation is that there's a simple reduction from vertex cover to set cover. Vertices become sets of their edges and edges are universe elements. We then use our old greedy approximation for set cover. Therefore, our original greedy algorithm for vertex cover is a $\ln m$ approximation, which is at most $2 \ln n$. Is this approximation tight? It IS tight for set cover (we showed this previously). However, this does not mean our set cover can be realized by some instance of vertex cover, so the approximation may not be tight.
It turns out it is. Remember that the greedy algorithm is something like take the highest degree vertex first, then the next highest, etc. You can show with the following construction that greedy gives $OPT \cdot \ln n$. Have a bipartite graph with one class L having n vertices. Then, in the other class R, $\left\lfloor \frac{n}{i} \right\rfloor$ vertices of degree i. We claim OPT is at most n; just take everything in L. However, greedy will either take everything in R first, or take something in L which has the same degree as something in R (effectively replacing it). Either way, $|R|$ vertices will be taken. But $|R| \approx n \ln n$ (harmonic series) so greedy is tight.

Here is a better algorithm for approximating vertex cover.

Algorithm 8.4 (Greedier Vertex Cover)

```
function VC(()$V, E$)
    $S \rightarrow \emptyset$
    while $\exists \in E$ not covered by $S$ do
        Add both endpoints of $e$ to $S$
    return $S$
```

This is really good. In fact:

Theorem 8.3
Under this algorithm,

$$|S| \leq 2 \cdot OPT$$

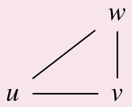
Proof
Look at edges e not covered while loop iterations which forced us to add to S. Call the set of such edges M. We claim M is a maximal matching. It is clearly a matching, as no edges can have a vertex in common (because otherwise after choosing one of the edges to trigger the while loop, we would not choose the other). Now, we need to show that M is maximal; i.e. we cannot add to it to make it bigger. Suppose there was a way to add an edge to increase the size of the matching; this would mean that there would be an edge that we can add that has no vertex in common with any other edge; however, since S at the end is a vertex cover, this is impossible, thus M is a maximal matching.
For all vertex covers S and all matchings M, $|S| \geq |M|$, (which would mean $OPT \geq |M|$). The reason is the following: suppose you have some matching M; then we have to take at least one vertex for each edge in M (since they share no common vertices).
By the definition of the algorithm, $S \leq 2|M|$ (since in each iteration we add both endpoints). Now the inequalities fall into place:

$$|S| \leq 2|M| \leq 2OPT$$

Thus, the S we find is at most a 2-approximation.

So, this is a great approximation. Can we do better? No one knows. But we can do the same (in a cool way).

Algorithm 8.5 (ILP Vertex Cover)
Take the following graph; the construction we show works generally.



We write the following integer linear program:

$$\begin{aligned}
&\min x_u + x_v + x_w \text{ subject to} \\
&x_u + x_v \geq 1 \\
&x_u + x_v \geq 1 \\
&x_u + x_w \geq 1 \\
&0 \leq x_u, x_v, x_w \geq 1 \\
&x \text{ must be integral}
\end{aligned}$$

Now, what we do is do an LP Relaxation (take out the integrality constraint). Then, we shove this into an LP solver to solve in polynomial time. However, this is doesn't work perfectly (for example, OPT here is: $x_u = x_v = x_w = \frac{1}{2}$). To turn this into an integer solution, "round" the fracational solution to an integral solution. "Rounding" in different cases can be done in different ways; in this case, let's round in the standard numerical sense.
Correctness Concerns The worry is that by rounding down, you may not have a vertex cover. However, for each edge (u, v), note that at least one of the variables must be $\geq \frac{1}{2}$ in order for the constraint to be satisfied. This means that every edge will be covered, since at least one of these will be rounded up to 1.
Cost Analysis We claim that $OPT(\text{LP}) \leq OPT(\text{ILP})$ since the feasible set in the LP is a superset of the one in the ILP. Furthermore, the vertex cover returned has size at most $2 \cdot OPT(\text{LP})$, because the objective function

can only at most double. Thus, the size of the vertex cover is at most $2 \cdot OPT(\text{ILP}) \leq 2 \cdot OPT(\text{VC})$. Thus, algorithm is a 2-approximation.

8.3.3 Approximations for TSP

As a reminder, the Traveling Salesman Problem (TSP) is the following.

Given n cities with distances d_{ij}, the goal is to start at city 1, visit every other city once, and return to 1, such that this tour is as short as possible. We consider the metric TSP, i.e. the distances works as a metric (non-negative, $d_{ii} = 0$, triangle inequality holds).

First, we think about how to build a tour. Any tour C is a cycle from 1 back to itself. Thus, C with the last edge removed is a path (which is a special case of a spanning tree). This means that $OPT \geq$ cost of a spanning tree $\geq$ cost(MST). This gives rise to the following idea.

Algorithm 8.6 (Metric TSP 2-approximation)
We do the following.

- Compute MST T^*.
- Traverse T^* with a DFS pre-order tour (i.e. traverse the vertices in pre-order).

Thus, from our property before, the cost of a full DFS tour (going from parents to children back to parent) is $2 \cdot \text{cost(MST)} \leq 2 \cdot OPT$. However, the cost of a pre-order tour is even less by the triangle inequality. Thus the cost of the pre-order tour $\leq 2 \cdot OPT$, so this is a valid 2-approximation.

Now, we think about the general case of TSP, without the triangle inequality.

Theorem 8.4 (General TSP)
For any poly-time computable function $f(n)$, there is no $f(n)$-approximation of general TSP (no triangle inequality) unless $\mathcal{P} = \mathcal{NP}$.

9 Randomized Algorithms

9.1 Lecture 22

9.1.1 Randomized Algorithms

There are two types of randomized algorithms we will study. We can think of randomized algorithms as taking in two inputs x (the usual input) and r (some randomization source).

Definition 9.1 (Las Vegas Random Algorithm)
A Las Vegas Random Algorithm is always correct. However, its runtime is a random variable that is small in expectation. An example of this is QuickSort. We put an upper bound on $\mathbb{E}_r T(\mathcal{A}(x, r))$ for all x (i.e. worst-case x).

Definition 9.2 (Monte Carlo Random Algorithm)
Monte Carlo Random Algorithm is always efficient. However, correctness is a random variable that is nearly correct in expectation. An example of this is polling a sample of the population to estimate the incidence in the entire population. We put a lower bound on $\mathbb{P}_r(\mathcal{A}(x, r)$ is correct).

9.1.2 Probability Basics

Definition 9.3 (Random Variable)
A random variable X is a function that maps outcomes in randomness (some "coin flip") to real numbers.

Definition 9.4 (Expectation)
The expectation of a random variable X with support $\mathcal{X}$ is defined as:

$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} x \mathbb{P}[X = x]$$

Theorem 9.1 (Markov's Inequality)
For some nonnegative random variable X, we have

$$\mathbb{P}[X > k] \leq \frac{\mathbb{E}[X]}{k}$$

Markov's in particular very good for bounding the runtime of Las Vegas algorithms (even though it's considered fairly "crude").

9.1.3 Las Vegas

The first random algorithm we look at is for comparison-based sorting, where without loss of generality all array elements are distinct. This is a Las Vegas randomized algorithm.

Algorithm 9.1 (QuickSort)
The quicksort algorithm works as follows. Suppose our array is A.

1. Randomly choose an index p (the pivot) in the array A.
2. Compare every element in A to $A[p]$. If it's smaller, put it in the new array L; if it's bigger, put it in the new array R.
3. Recursively call QuickSort on L and R. Then, the answer is sorted L, pivot, sorted R.

(Probabilistic) Runtime Analysis First, note that runtime is proportional to the number of comparisons. Also, note that you will never compare the same two elements twice (since we pull out the pivot from the recursion). This means, we can bound the runtime as:

$$T(n) = C \cdot \sum_{i<j} X_{ij}$$

where $X_{ij} = \mathbf{1}\{\text{if } i\text{th smallest element is compared with } j\text{th smallest element}\}$. By linearity, this means that:

$$\begin{aligned}\mathbb{E}[T(n)] &= C \cdot \mathbb{E}\left[\sum_{i<j} X_{ij}\right] \\ &= C \cdot \sum_{i<j} \mathbb{E}\left[X_{ij}\right] \\ &= C \cdot \sum_{i<j} \mathbb{P}\left[i\text{th and } j\text{th elements are ever compared}\right]\end{aligned}$$

This probability can be computed with the following logic. Note that we choose whether or not to compare i or j when we pick a pivot between i and j, inclusive (otherwise i and j both end up in the same subarray and the decision is pushed to a recursive call). If the pivot is in this region, we only compare the two elements if i or j is the pivot, which has probability $\frac{2}{j-i+1}$. Then, we compute:

$$\begin{aligned}\mathbb{E}[T(n)] &= 2C \sum_{i<j} \frac{1}{j-i+1} && (1)\\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j-i+1} && (2)\\ &= \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) + \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1}\right) + \ldots && (3)\\ &= (n-1)\frac{1}{2} + (n-2)\frac{1}{3} + \cdots + (1)\frac{1}{n} && (4)\\ &\leq n(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) && (5)\\ &\leq n \ln n = \mathcal{O}(n \log n) && (6)\end{aligned}$$

If you use Markov's inequality, then we have:

$$\mathbb{P}\left[T(n) > dn \log n\right] < \frac{c \cdot n \log n}{d \cdot n \log n} = \frac{c}{d}$$

which is shows that differing by a different enough constant is already fairly unlikely (concentration about the mean).

9.1.4 Monte Carlo

Next, we take a look at some Monte Carlo Random Algorithms.

Consider $A, B \in \mathcal{R}^{n\times n}$. Suppose something could multiply matrices and give us an answer. We now seek to verify the result of a matrix multiplication $C = A \times B$ (in faster than the $\mathcal{O}(n^3)$ time needed to just multiply A and B).

Algorithm 9.2 (Frievald's Algorithm)
Pick vectors $x_1, \ldots, x_T \in \{0,1\}^n$ independently, uniformly at random. for $i = 1$ to T:

- If $ABx_i \neq Cx_i$, then return False

Otherwise, return true.
Note that the time to run this algorithm is $\mathcal{O}(Tn^2)$ because each matrix-vector multiplication takes n^2 time.
(Probabilistic) Correctness Let us denote $D = AB - C$. We essentially want to know if $D = 0$. Our claim is that $D \neq 0$, then $\mathbb{P}[Dx = 0] \leq \frac{1}{2}$. This means the probability of never catching an error (incorrectness) is at most $(\frac{1}{2})^T$. Suppose you want a bound on incorrectness to be p, set $T = \left\lceil \log_2 \frac{1}{p} \right\rceil$. Note that if $AB = C$, we will always return true, so we only need the other case. Let's prove our probability claim.
If $D \neq 0$, then there exists an entry $D_{ij} \neq 0$, thus there exists a column d_j that is not the zero vector.
Note that if $Dx = 0$, then

$$Dx = \sum_{k=1}^{n} x_k d_k = x_j d_j + \sum_{k\neq j} x_k d_k = 0$$

Let $x\Big|_j$ mean "flip the jth bit of x". If $x_j = 0$, then $\sum_{k\neq j} x_k d_k$ must have been 0. This means $Dx\Big|_j \neq 0$, because it is d_j. If $x_j = 1$, then $\sum_{k\neq j} x_k d_k$ must have been $-d_j$. This means $Dx\Big|_j \neq 0$, because it is d_j.
Now, let us pair $\{0,1\}^n$ into 2^{n-1} pairs, where each x is paired with $x\Big|_j$. This means that the number of x such that $Dx = 0$ (since only at most one of the pairs can be 0) is at most 2^{n-1}. Thus, $\mathbb{P}[Dx = 0] \leq \frac{2^{n-1}}{2^n} = \frac{1}{2}$.

Lastly, we visit the problem of the the global min-cut. Suppose you have a weighted, undirected graph G. This means to find a cut (non-empty partition of V) such that the total weight crossing the cut is minimized. This problem can be solved by $n-1$ calls of max-flow/min-cut. We consider the unweighted case to look at a simple random algorithm.

Algorithm 9.3 (Karger's Contraction Algorithm)
First, we define contraction. Vertices u and v over an edge (u, v) are contracted by "gluing" the nodes together into a meta-node, and keeping all the edges to other vertices (this may make the graph a multi-graph).
Now the main runtime of the algorithm looks like:

1. Pick an edge uniformly at random and contract along that edge.
2. Keep doing this until there are two meta-vertices left; these meta-vertices are the cut.

We run this r times (where r is sufficiently large–see below).
Note the runtime is is $\mathcal{O}(rn^2)$.
(Probabilistic) Correctness We first have the following claim. Fix a particular min-cut $S, V \setminus S$. Then,

$$\mathbb{P}[\text{contraction iteration returns } S] \geq \frac{1}{\binom{n}{2}}$$

To see this, let k denote the number of edges crossing $(S, V\setminus S)$. Note that the algorithm outputs S if and only if it never contracts an edge crossing the cut. Consider the 1st round of contraction. The probability of contracting an edge crossing the cut is $\frac{k}{m}$.
Look at all cuts of the form $\{u\}, V \setminus \{u\}$. We know the amount of edges crossing the cut must be at least k, but the number of edges crossing is $\deg(u)$. We know $\sum_{u\in V} \deg(u) = 2m \geq nk$. We must have $m \geq \frac{nk}{2}$.
Thus, the probability is at most $\frac{k}{\frac{nk}{2}} = \frac{2}{n}$. This means the probability of not contracting an edge across the cut is at least $1 - \frac{2}{n}$. The next time around, the probability of not messing up is at least $1 - \frac{2}{n-1}$, and so forth. This

means the probability of returning the cut is:

$$\begin{aligned}\mathbb{P}\left[\text{contraction iteration returns } S\right] &\geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\cdots \\ &= \frac{n-2}{n}\cdot\frac{n-3}{n-1}\cdot\frac{n-4}{n-2}\cdot\frac{n-5}{n-3}\cdots\frac{2}{4}\cdot\frac{1}{3} \\ &= \frac{2}{n(n-1)} \\ &= \frac{1}{\binom{n}{2}}\end{aligned}$$

By this claim, the chance of failing every time is at most:

$$\mathbb{P}\left[\text{fail to find mincut}\right] \leq \left(1-\frac{1}{\binom{n}{2}}\right)^r \leq e^{-\frac{r}{\binom{n}{2}}}$$

If we want this to be at most some confidence p, then we choose $r = \left\lceil \binom{n}{2} \ln \frac{1}{p} \right\rceil$. Note that a corollary of this claim is that there are at most $\binom{n}{2}$ mincuts (a tight example is the n-cycle).

9.2 Lecture 23

9.2.1 Hashing

We consider the dictionary problem. We wish to maintain a database of (key, value) pairs were the keys are integers in $\{0, \dots, U-1\}$. Suppose there are n distinct keys in the dictionary. There are two different formulations of this idea.

1. Static: The database is given up front (no insertions/deletions). Must support query(k).
2. Dynamic: Supports insertions, deletions, and queries.

There are many different solutions to this data structure problem.

1. Store database in an array of size U. This works for both static and dynamic. $\mathcal{O}(1)$ insertion, deletion, and query, but $\mathcal{O}(U)$ memory.
2. Store database in a balanced binary search tree. $\mathcal{O}(n)$ memory, $\mathcal{O}(\log n)$ time per insertion/deletion (deterministic).
3. In the static case, you can store data in an array sorted by key. To query, use binary search. This approach has similar asymptotics to the tree approach.

It turns out there is a randomized way to do this even faster.

9.2.2 Dynamic Hashing

First we consider the dynamic case.

Algorithm 9.4 (Hashing with Chaining)
First, we choose a random "hash" function $h : \{0, \dots, U-1\} \to \{0, \dots, m-1\}$. (That is, at initialization the function mapping the elements of U is made in a random way).
Then, we store an array of m elements, where $A[i]$ contains a doubly-linked list storing all (k, v) such that $h(k) = i$. To insert (k, v), calculate $h(k)$ and put it at the front of the doubly-linked list at $A[h(k)]$. To delete, go to that linked list and then remove it from the doubly-linked list. Finally, to query, go to $A[h(k)]$ and go through the entire linked list and search for the element.
Runtime Analysis We claim $\mathbb{E}\left[\text{time for an op}\right] = \mathcal{O}(T_h + \frac{n}{m}) = \mathcal{O}(1)$ if $T_h = \mathcal{O}(1)$, $m \geq n$ where T_h is the time to run the hash function.

Proof
Call $T(x)$ the time to query x (insertion and deletion are very similar). Define indicators $Z_i = \mathbf{1}\{h(k_i) = h(x)\}$ where k_i is the ith key.

$$\begin{aligned}
\mathbb{E}\left[T(x)\right] &\leq \mathbb{E}\left[T_h + C \cdot \text{Size of linked list containing } x\right] \\
&\leq T_h + C \cdot \sum_{i=1}^{n} \mathbb{E}\left[Z_i\right] \\
&\leq T_h + Cn \cdot \mathbb{P}\left[Z_i = 1\right] \\
&\leq T_h + C\frac{n}{m}
\end{aligned}$$

Thus we have the claim.

However, there is a problem. Naively picking and storing a random h takes U memory (we have to decide the output for all inputs!). Note that in our analysis all that mattered was that the probability of a collision was $\frac{1}{m}$. Let us see how to solve this.

We use an idea called universal hashing or k-wise independent hash families.

Definition 9.5 (k-wise Independent Hash Family)
A set $\mathcal{H}$ of functions mapping $\{0, \dots, U-1\} \to \{0, \dots, m-1\}$ is k-wise independent if for all choices of inputs $x_1 \neq x_2 \neq \cdots \neq x_k$ and all choices of outputs $y_1, y_2, \dots, y_k$,

$$\mathbb{P}_{h \in \mathcal{H}}((h(x_1) = y_1) \wedge (h(x_2) = y_2) \wedge \cdots \wedge (h(x_k) = y_k)) = \frac{1}{m^k}$$

Definition 9.6 (Universal Hash Family)
A set $\mathcal{H}$ of functions mapping $\{0, \dots, U-1\} \to \{0, \dots, m-1\}$ is universal if for all choices of inputs $x_1 \neq x_2$,

$$\mathbb{P}_{h \in \mathcal{H}}(h(x_1) = h(x_2)) = \frac{1}{m}$$

i.e. collision probability is $\frac{1}{m}$.

Note that 2-wise independence implies universality. Thus, from our definition, our proof only needed h to be drawn from a universal hash family. Now we ask "how do we get a universal hash family"?

Example 9.1
Suppose $\mathcal{H}$ is the set of all functions mapping from $\{0, \dots, U-1\}$ to $\{0, \dots, m-1\}$. This is k-wise independent for all k. Specifying any of these functions $h \in \mathcal{H}$ takes $\log_2 |\mathcal{H}|$ bits (as a 'seed'). In this case, $|\mathcal{H} = m^U|$, so it takes $\Theta(U \log m)$ bits.

Let us see an even better hash family that has very short seed length.

Example 9.2
Fix $U = m = p$ which is a prime. Consider the family

$$\mathcal{H} = \{(ax + b) \mod p : a, b \in \{0, \dots, p-1\}\}$$

Then note that $|\mathcal{H}| = p^2$, which means $\log |\mathcal{H}| = \mathcal{O}(\log p)$. You can show that $\mathcal{H}$ is 2-wise independent; the sketch of the proof is that $h(x_1) = y_1 \wedge h(x_2) = y_2$ is uniform over y_1, y_2 since you need two points to specify a line. However, it is not 3-wise independent, because once you specify the line, it is known what $h(x_3)$ is.

9.2.3 Static (Perfect) Hashing

Now we turn to the static dictionary, where we have constant query time guaranteed, instead of in expectation.

Definition 9.7 (Perfect Hashing)
A hash function h is a perfect hash function if for all $x_1 \neq k_2$ in the database, then $h(k_1) \neq h(k_2)$.

The idea is with the "inverse" birthday paradox; if you have way less than $\sqrt{365}$ people in a room, probably no two people have the same birthday. This means we want m, the size of the hash table, to be roughly $m \geq \Omega(n^2)$, i.e. where *n* is the size of the database. Let's do a more careful analysis:

$$\begin{aligned}\mathbb{P}\left[\text{at least one collision}\right] &= \mathbb{P}\left[\text{number of collisions} \geq 1\right] \\ &\leq \frac{\mathbb{E}\left[\text{number of collisions}\right]}{1} \\ &= \binom{n}{2}\mathbb{P}\left[a \text{ and } b \text{ collide}\right] \\ &= \frac{\binom{n}{2}}{m} \\ &\leq \frac{1}{2}\end{aligned}$$

if we pick $m = n^2$. If we have a collision, we redo the process again (probability of infinite collisions is 0). Keep doing this over and over again. This means the expected number of times we have to redo the process is twice.

9.3 Lecture 24

9.3.1 Streaming Algorithms

Definition 9.8 (Streaming Algorithms)
Streaming Algorithms are algorithms that make one pass over some set of data. Equivalently, these are dynamic data structures (supports updating the data). In particular, we are concerned about minimizing memory consumption when doing these things (want $o(n)$ memory).

One use case may be trying to figure out if you work at an internet company and are trying to find out if there is a DOS attack. Clearly, you can't store every single packet (this is a lot of data), so you want some kind of online algorithm that can sit at the router and make assertions from the data it sees flying by.

Here are some types of problems streaming addresses.

- Counting problems (below)
- Graph problems on dynamically changing graphs (you can find a spanning forst in $\mathcal{O}(n \log^3 n)$ where n is the amount of nodes).
- Linear algebra problems (low-rank approximation, regression)
- Geometric problems

Here is our first problem. We want to build a counter N subject to 3 operations.

1. init(): $N \leftarrow 0$
2. incr(): $N \leftarrow N + 1$
3. query(): **return** N

The trivial algorithm is a single counter with N represented in binary. This takes $\Theta(\log N)$ bits. Suppose we want to do better and are given an upper bound on the counter (it will never exceed T). If we use s bits of memory, the amount of states the program can have is 2^s. Since we need at least T distinct states, s can be no smaller than $\log T$.

9.3.2 Approximate Counter

Instead, we need to settle for some kind of approximate counting scheme. Suppose Jelani gave us a counter value that was $N = 500$ digits long. We'd need $500 \log_2 10$ bits of memory. We approximate by storing $5 - 0 - 0$, i.e. the $\lceil \log_{10} N \rceil$ (we know the true counter is between 10^{500} and $10^{501} - 1$). Alternatively we can say we want the number to 1% error, i.e. store $\lceil \log_{1.01} N \rceil$.

To see how good this is, this means that because $\log(1 + \epsilon) = \Theta(\epsilon)$,

$$\log_{1.01} N = \frac{\log_{10} N}{\log_{10} 1.01} \approx 100 \log_{10} N$$

Thus for a general ϵ, we only need to store a value that is roughly $\Theta\left(\frac{\log N}{\epsilon}\right)$ to know the answer within an error of ϵ. This means we can hope to use about $\log \log N + \log \frac{1}{\epsilon}$ bits instead of $\log N$ bits.

Now, let's figure out how to change this algorithm to be a bona-fide data structure; supporting increments. Suppose randomly incremented instead. The first idea of the algorithm is not to store N in memory, instead store X. Then the operations work as follows:

- init(): $X \leftarrow 0$

- incr(): $X \leftarrow X + 1$ with probability p
- query(): **return** $\frac{X}{p}$

The problem with this is that in generally you only same $\log \frac{1}{p}$ bits of memory; we want this to increase with N. Instead, we use the following function.

- init(): $X \leftarrow 0$
- incr(): $X \leftarrow X + 1$ with probability $\frac{1}{2^X}$
- query(): **return** $2^X - 1$

Let us show why this works to reduce memory.

Theorem 9.2
Let X_n be the value of X after first n increments. We claim $\mathbb{E}\left[2^{X_n}\right] = n + 1$.

Proof
To see this, proceed by induction. Note that for a base case $n = 0$, then $X_0 = 0$, so $\mathbb{E}\left[2^0\right] = 1 = 0 + 1$. Now consider the inductive step.

$$\begin{aligned}
\mathbb{E}\left[2^{X_{n+1}}\right] &= \sum_{j=0}^{\infty} \mathbb{P}\left[X_n = j\right] \mathbb{E}\left[2^{X_{n+1}} \mid X_n = j\right] \\
&= \sum_{j=0}^{\infty} \mathbb{P}\left[X_n = j\right] \left(\frac{1}{2^j} 2^{j+1} + \left(1 - \frac{1}{2^j}\right) 2^j\right) \\
&= \sum_{j=0}^{\infty} \mathbb{P}\left[X_n = j\right] (2^j + 1) \\
&= \sum_{j=0}^{\infty} \mathbb{P}\left[X_n = j\right] (1) + \sum_{j=0}^{\infty} \mathbb{P}\left[X_n = j\right] 2^j \\
&= 1 + \mathbb{E}\left[2^{X_n}\right] \\
&= n + 2
\end{aligned}$$

because $\mathbb{E}\left[2^{X_n}\right] = n + 1$ by inductive hypothesis. Thus the claim is true by induction.
We still have an issue. The error between X and N is large when N is small.

9.3.3 Reducing Randomized Error, Morris' Algorithm

To do further analysis, we cover other probabilistic tools.

Definition 9.9 (Variance)
For a random variable X, its variance is defined as:

$$\mathrm{Var}\,(X) = \mathbb{E}\left[(X - \mathbb{E}\left[X\right])^2\right] = \mathbb{E}\left[X^2\right] - (\mathbb{E}\left[X\right])^2$$

Theorem 9.3 (Chebyshev's Inequality)
Consider a random variable X and some $\lambda > 0$. Then we have,

$$\mathbb{P}\left[|X - \mathbb{E}\left[X\right]| > \lambda\right] \leq \frac{\mathrm{Var}\left(X\right)}{\lambda^2}$$

We have an unbiased estimator Z for N ($Z = 2^X - 1$). Let us bound the error probability to p, through Chebyshev's

$$\mathbb{P}\left[|Z - N| > \epsilon N\right] \leq \frac{\mathrm{Var}\left(Z\right)}{\epsilon^2 N^2} \leq p$$

so to keep error probability low, we want the variance to be small. Since the expectation of Z is fixed as N, we want $\mathbb{E}\left[Z^2\right]$. The algorithm so far does not even depend on ϵ, so we can definitely not control the variance below this threshold. In fact, it is possible to show by induction that

$$\mathbb{E}\left[2^{2X_n}\right] = \frac{3}{2}n^2 + \frac{3}{2}n + 1 \implies \mathrm{Var}\left(Z\right) = \frac{1}{2}n^2 - \frac{1}{2}n - 1$$

There are two ways to reduce the variance down.

1. Run R copies of the algorithm in parallel, then us $Z = \frac{1}{R}\sum_{i=1}^{R} Z_i$ as an estimator. In fact, $R = \frac{1}{\epsilon^2 p}$ works.
2. Change the base of exponentiation in the increment probability. Note that 0.5^X has low memory but high variance; 1.0^X has high memory but low variance. In fact, one can show that if the we increment with probability $\frac{1}{(1+a)^X}$ and use estimator $Z = \frac{(1+a)^X - 1}{a}$ one can show $\mathbb{E}\left[Z\right] = N$ and $\mathrm{Var}\left(Z\right) \leq \frac{aN(N-1)}{2}$. The choice $a = 2\epsilon^2 p$ works. Furthermore, one can show the memory is close to

$$\mathcal{O}\left(\log\log N + \log\frac{1}{a}\right) = \mathcal{O}\left(\log\log N + \log\frac{1}{\epsilon} + \log\frac{1}{p}\right)$$

   which is very close to our theoretical best (without the last term).

The second approach is termed Morris' algorithm. It turns out, from cutting-edge research done by Professor Nelson himself found that the best bound is:

$$\mathcal{O}\left(\log\log N + \log\frac{1}{\epsilon} + \log\log p\right)$$

unless your p and ϵ are so small that you're better off using a $\log N$ deterministic algorithm instead.

9.3.4 Unique Counting

We now consider the unique counting problem. Now, we want to consider distinct users that increment the counter. We see a stream of m items (possibly with duplicates) coming from some universe $\{1, \ldots, n\}$. Want to count the number of distinct elements in this stream.

Theorem 9.4
There exists an algorithm for unique counting using $\mathcal{O}\left(\frac{1}{\epsilon^2} + \log n\right)$ bits of memory with success probability $1 - \epsilon$.

Here is an idealized algorithm for this approach.

- init(): $X \leftarrow 1$, Pick a random hash function $h : [n] \rightarrow [0, 1]$.
- update(i): $X \leftarrow \min(X, h(i))$

- query(): **return** $\frac{1}{X} - 1$

You can run this many times to reduce variance again. However, there are a few issues; firstly you cannot store real numbers, secondly you have to store *h* efficiently. These problems can be fixed by picking *h* from a 2-wise independent hash family.

10 Lower Bounds and Alternate Models of Computation

10.1 Lecture 25

10.1.1 Lower Bounds

So far, we have tried to minimize computational resources to solve various problems. In particular, we have cared about upper bounds; given some problem P, we can define some function $f_P(n)$ as the minimum number of resources (number of timesteps, number of bits of memory, etc) to solve P on worst-case inputs of size n.

Thus, we can think of an algorithm as an upper bound of $f_P(n)$, i.e. there might exist a more efficient algorithm, but the theoretically $f_P(n)$ must be at least the runtime/memory of that algorithm. How can we prove a lower bound on $f_P(n)$?

If the measure of complexity is the number of comparisons, we know that $f_{\text{sorting}}(n) = \Omega(n \log n)$ and $f_{\text{sorting}}(n) = \mathcal{O}(n \log n)$ because merge-sort exists. Thus, we have a very tight bound.

However, for non-comparison based algorithms, we want general time lower bounds. Turns out, this is pretty hard. For all we know, all NP-complete problems we mentioned prior may be able to be solved in linear time. There are a few problems where $\omega(n)$ complexity are known through the time hierarchy theorem (which we will not discuss in depth). Here's what such a problem looks like.

Example 10.1
Take some problem that looks like the following.

- Input is the source code to a program P and a string x.
- If we run P on input x, would it terminate in at most $|x|^3$ steps?

Clearly this takes $\Theta(|x|^3)$ on worst-case input if we were to just simulate it (where the program doesn't terminate at all; we cannot know this apriori due to the halting problem). It turns out due to this "time hierarchy theorem," it turns out we can put a lower bound on it that looks like $\Omega(n^{3-\epsilon})$ steps.

It turns out it's much easier to prove lower bounds for alternate models of computation. Here are a few we will study.

- Comparison Model - Sorting example discussed above
- Circuit Model
- Cell Probe Model
- Communication Model

10.1.2 Circuit Model

Suppose one has a circuit of n input bits, with AND, OR, NOT gates. Consider the problem of integer multiplication. Suppose we have a collection $\{M_{2n}\}_{n=1}^{\infty}$ which are functions that multiply two bit strings of size n (i.e. two n-bit numbers).

$$M_{2n} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{2n}$$

The question is designing a family of Boolean circuits $\{C_{2n}\}$ each solving M_{2n} so as to minimize some notion of complexity.

We mentioned prior that one can always write a circuit that computes the output of an algorithm where the size of the circuit is proportional to the runtime of the algorithm. Thus, a trick to prove runtime lower bounds on algorithms would be to prove circuit size lower bounds.

Suppose we measure size s of a circuit by the number of wires. It turns out that

Theorem 10.1
The number of circuits of size s is $2^{\mathcal{O}(s \log s)}$.

However, the number of functions on n bits mapping to an output bit is is exactly 2^{2^n} (2^n choices for each input, 2 choices for each output). Thus if s is polynomial in n, this means that $2^{\mathcal{O}(s \log s)}$ is very small to 2^{2^n} that almost every function cannot be computed by poly-sized circuits. We could try to find one of these an exponential lower bound for the circuit size of certain problems. However, coming up with a problem that takes exponential size to solve IS REALLY hard.

Note 10.1
The highest known lower bound for any explicit function is size $\geq (3 + \frac{1}{86})n$.

If we restrict depth, this becomes a little bit better.

Example 10.2
Consider the parity problem, where we want to find the parity of some bitstring (the xor of all its bits); it turns out any $\mathcal{O}(1)$ depth circuit family for PARITY requires size $\geq \exp(\Omega(n))$.

10.1.3 Cell Probe Model

We analyze the cell probe model (for data structures). It looks like the following. Suppose you have some memory, where each word in the memory takes up w bits and a finite memory of size S. Now, suppose a data structure algorithm receives a query call; it's going to read/write some of the cells of memory somehow. In a read call, we read all w bits of some block; in a write call, we write all w bits of some block. After some finite amount of rounds of communication between the algorithm and memory, the query call will be resolved. Thus, we measure the cost of the computation, the query time, (in lower bound) the number of rounds of communication.

It turns out there is a lot more headway made in this area.

Example 10.3

- We can use the cell probe model to prove that amortized complexity for union find is $\Omega(\alpha(m,n))$; meaning that the disjoint forest data structure is optimal (since it runs in $\mathcal{O}(\alpha(m,n))$).
- Consider the dynamic partial sum problem. We have n items, where a query$(j) = \sum_{i=1}^{j} x_i$ and update(j, δ) does $x_j \leftarrow \delta$. It turns out $\min(t_{update}, t_{query}) = \Omega(\log n)$, but note that we can solve this with a balance binary search tree (store the sum of a node's subtree in the node); this already has $\mathcal{O}(\log n)$ updates and queries.

How were these bounds proven? They generally come from communication lower bounds (like information theory).

10.1.4 Communication Model

Suppose there are two entities Alice and Bob who want to together compute $f(x, y)$, where Alice gets input $x \in X$ and Bob gets input $y \in Y$. They send messages $m_1, m_2, \ldots$ until one of them figures out the answer.

Some goals are to minimize the number of rounds as well as minimize the total bits communicated. There are also variants to the problem.

- Deterministic
- Randomized

  - Public coin (both know the same random string, without having to communicate it)
  - Private coin (each has their own random string, unknownst to the other)

Example 10.4
Consider the EQ (Equality) problem. $X = Y = \{0, 1\}^n$ and we wish to determine whether x and y are equal bit strings. We consider the amount of bits of communication one needs to use.

- In the deterministic setting, it turns out you need at least n communication, so the problem is $\Theta(n)$.
- In the randomized public coin settings, it turns out you have $\Theta(1)$ bits. Thus, we
- In the randomized private coin setting, it turns out you have $\Theta(\log n)$. Alice will look at the first n^2 primes and choose a random prime p amongst those and send it to Bob. Then, we compute $x \mod p$ and set that. Then, Bob can compute $y \mod p$, and return yes if $(x - y)$ is 0. Since there are about $\frac{\log x}{\log \log x}$ prime factors of x the chance that x and y share the prime factor p is very small.

10.1.5 Lower Bounds for Streaming

It turns out one can prove memory lower bouhds for streaming algorithms via communication complexity lower bounds.

Theorem 10.2 (Lower Bound for Unique Counting)
Any space-S deterministic streaming algorithm for exact unique counting means there exists a deterministic communication protocol for EQ with total communication $\leq S$.

Proof
Suppose such an algorithm $\mathcal{A}$ exists. Then we can have the following protocol to solve EQ. Alice creates a stream containing all i such that $X_i = 1$. Then, Alice runs $\mathcal{A}$ on the stream. Then, Alice sends all S bits of memory used by the algorithm to Bob. By the correctness of $\mathcal{A}$, Bob must be able to use $\mathcal{A}$ to query the memory and find the number of 1s in Alice's input, suppose this number is t. Bob streams an index of 1 into $\mathcal{A}$. If the answer is $t + 1$, then Alice's bit at position 1 is 0; otherwise it is a 1. Repeating this over all indices allows Bob to reconstruct Alice's entire input. Then, Bob can simply check bit-by-bit if this matches his bit-string. This protocol only used S bits of communication, proving the claim.

But any communication algorithm needs at least n bits to be communicated; thus any exact deterministic streaming algorithm for unique counting must use at least n bits of memory (hash maps are optimal).

Here are two other results that arise. It's possible to show:

Theorem 10.3 (Relaxing Unique Counting)

- Any approximate deterministic algorithm for unique counting requires $\Omega(n)$ bits of memory.
- Any exact randomized algorithm for unique counting requires $\Omega(n)$ bits of memory.

Thus, the only way to get an efficient solution for this problem is using a randomized algorithm and approximation.