Survivor's Manual for CIS 3200

JEFFREY SAITO (HITMAN7128)

Last Update: December 21, 2024

> How do you optimally encode a text file? How do you find shortest paths in a map? How do you design a communication network? How do you route data in a network? What are the limits of efficient computation? This course gives a comprehensive introduction to design and analysis of algorithms, and answers along the way to these and many other interesting computational questions. You will learn about problem-solving; advanced data structures such as universal hashing and red-black trees; advanced design and analysis techniques such as dynamic programming and amortized analysis; graph algorithms such as minimum spanning trees and network flows; NP-completeness theory; and approximation algorithms.
>
> — *Description of CIS 3200 in course catalog*

All exams in CIS 3200 are completely closed note.

Contents

§1 Introduction

First off, it helps to understand the many goals in this class because it naturally leads in to how to tackle it.

- **Communication of algorithms:** You have to make sure you include all relevant parts of your algorithm. It helps to understand why each part is necessary so it's easier for you to remember to write it on the test. I will try my best to explicitly state what is necessary for specific types of problems. However, the onus is ultimately on *you* to ask the professor and TAs what is sufficient justification, since that can change depending on the professor. Bear in mind, for HWs and tests, they don't care if you "meant" to write something that you didn't actually write.
- **Formalizing reasoning:** The professor does not like proofs that rely on the words "and so on." An example would be "This statement being true for $n = 1$ implies it holds for $n = 2$, and the statement being true for $n = 2$ implies it holds for $n = 3$, *and so on*." It's not a proper proof of a pattern indefinitely holding because for all we know, the pattern could break at some point. The *proper* way to go about the proof is to use induction because that's the formal mathematical method of showing the pattern holds indefinitely. There will be other learning opportunities for writing airtight proofs, but that's the primary one.

- **Getting your hands dirty in unfamiliar situations:** Sometimes a problem may *look* intimidating. However, once you actually try out some examples to understand what is going on, it may not be so bad.
- **Problem solving on the fly:** This is what trips a lot of people up on the midterm, where there's time pressure compared to a HW. Luckily, by extracting intuition from HW problems and through pattern recognition, it will become easier to identify what you should try out (and sometimes, you'll figure out what to do immediately).
- **Modifying existing algorithms:** It's not uncommon to have a HW or midterm problem, where they take an algorithm seen in class and then throw a wrench in it or add a twist to it. For these type of problems, you'll have to understand why the original algorithm works the way it does in order to figure out what changes need to be made. You'll also have to make it clear why the changes allow the algorithm to still work as intended by identifying what parts of the proof can be preserved and what needs to be re-addressed.

§2 Midterm 1

Statistics from Fall 2024 iteration (out of 100 points):

- Maximum: 95
- Mean: 60.6
- Median: 61
- Standard Deviation: 18.06

§2.1 List of Content

Insertion Sort

- The inductive idea is to nibble at the sorting problem one step at a time: if $A[1..k]$ is sorted, we should try to extend that to $A[1..(k+1)]$.
- The mini-goal of getting $A[k+1]$ to the right spot is achieved through swaps. Applying the mini-goal for each element gets us a sorted array.
- POC relies on a loop invariant to show we can get from $A[1..k]$ sorted to $A[1..(k+1)]$ sorted.
- Runs in $\Theta(n^2)$ worst case.

Loop Invariants

- A true statement depending on loop index variable to show a property holds throughout the execution. Has 3 parts (with k being the looping variable):
- Base Case: Showing the statement holds even before the loop starts
- Maintenance: Showing that if the statement holds before the loop for $k = i$, it holds before the loop for $k = i + 1$
- Termination: Showing that we get the desired property at the end

Merge Sort

- We take a bigger leap of faith here with Merge Sort by using Divide and Conquer.
- We can assume by Strong Induction that the algorithm will sort both halves of the array, and then it's a matter of combining the sorted halves into one big sorted array to complete the induction.
- Merge is a sub-algorithm that can turn two sorted arrays into one big sorted array.
- Has runtime $T(n) = 2T(\frac{n}{2}) + O(n)$, which is $O(n \log n)$ by Master Theorem.

Quick Sort

- First, we pick a pivot and partition the array about it (requires $O(n)$ time).
- Then, both halves about the pivot are conquered.
- The runtime is $T(n) = O(n) + T(|A_1|) + T(|A_2|)$, where A_1 is the subarray left of the pivot and A_2 is the subarray right of the pivot.
- The base case is when $n = 0$ or 1, in which we do nothing.
- If the pivots constantly end up on extreme ends, then $T(n) = O(n^2)$.
- However, we can get the median in $O(n)$ time in which the recurrence becomes $T(n) = O(n) + 2T(\frac{n}{2})$, which is $O(n \log n)$.

Sorting Lower Bounds

- The comparison model is limited to two actions: determining if $i \leq j$ (for some inputs i and j) and shuffling memory around.
- Due to the binary nature of the comparison model, we create a decision tree modeling the sequence of actions that can happen based on the input.
- There are multiple parallels that can be drawn between the algorithm and decision tree:
  - Execution of algorithm $\longleftrightarrow$ root to leaf path
  - Execution time $\longleftrightarrow$ path length
  - Worst case runtime complexity $\longleftrightarrow$ depth of tree
- We must have at least 1 leaf for every possible output. But as a binary tree, if there at least L leaves, the depth of the tree is at least $\log_2(L)$.
- In the sorting problem, if we use the comparison model, there are $n!$ distinct outputs (one for each permutation of the array), so the depth is at least $\log_2(n!) = \Omega(n \log n)$.
- Thus, any sorting algorithm using the comparison model (like Merge Sort and Quick Sort) runs in $\Omega(n \log n)$.
- This can be generalized to say any algorithm using a binary model runs in $\Omega(\log L)$, where L is the number of distinct outputs.

Count Sort

- Given an array of integers and the range of the array, it will sort it without using the comparison model.
- With one pass of the array, the number of times each integer appears is recorded. Then, it is read out.
- Takes $O(n + k)$ time, where n is the length of the array and k is the length of the range of the array.
- It is a stable algorithm because we can preserve the relative order of objects with the same keys as we pass through the array.

Radix Sort

- A sorting algorithm on arrays of positive integers.
- We write each element in base b, for some integer $b > 1$. Then, we use Count Sort with the 1st LSD (least significant digit) as the key. Then, we use it again with the 2nd LSD as the key. We keep repeating this on the LSDs from right to left until we sort using the MSD (most significant digit) as the key.

- Proof of correctness relies on Count Sort being stable: Suppose $x < y$ and let x_i be the i^{th} digit from the right in x (define y_i similarly). Then, $x < y$ implies $\exists j$ s.t. $x_j < y_j$ and $x_\ell = y_\ell \ \forall \ell > j$. So when sorting by the j^{th} LSD, regardless of what happened before, x gets placed before y by $x_j < y_j$ and it will remain that way for future Count Sorts, since $x_\ell = y_\ell \ \forall \ell > j$ and Count Sort being stable.
- If n is the length of the array and k is the range of the array, runtime is $O(\log_b k) \cdot O(n+b)$. This is because $\log_b k$ is the number of digits of each entry in the array when converting to base-b, and $O(n+b)$ comes from one Count Sort use. If k is a polynomial in n, i.e. $k = n^{O(1)}$, then when $b = n$, this becomes $O(n)$.

Selection

- We can get the k^{th} smallest element in an array of n elements, for any $k \in [n]$, in $O(n)$ time.
- Divide the elements of the array into groups of 5. Then, find the median of each group. Recursively get the median of the set of medians and call this M. Partition the array about M and let A_L and A_R be the left and right halves, respectively, about M.
- As it turns out, we are guaranteed that $|A_L| \geq \frac{3n}{10} - 1$ and $|A_R| \geq \frac{3n}{10} - 4$. This limits how big the resulting subproblem is.
- Then, if M is the i^{th} smallest element, we recurse on the appropriate half to find the k^{th} smallest element:
  - If $i = k$, then we output M
  - If $k \leq i - 1$, then output SELECT(A_L, k)
  - If $k \geq i + 1$, output SELECT$(A_R, k - i)$
- Then, we have $T(n) = O(\frac{n}{5}) + T(\frac{n}{5}) + O(n) + T(\frac{7n}{10} + 4)$:
  - $O(\frac{n}{5})$ is from dividing into groups and get the median from each group
  - $T(\frac{n}{5})$ is from recursively getting M from the set of medians
  - $O(n)$ is from partitioning the array about M
  - $T(\frac{7n}{10} + 4)$ is the worst case of the resulting subproblem
- We can use induction to show $T(n) \leq Cn$ for a large enough constant C.

Master Theorem

- Suppose we have the recurrence $T(n) = a \cdot T(\frac{n}{b}) + \Theta(n^k)$
  - If $k > \log_b(a)$, then $T(n) = \Theta(n^k)$ (indicating the combine step is the bottleneck)
  - If $k = \log_b(a)$, then $T(n) = \Theta(n^k \log n)$ (indicating there's a balance between combine vs the subproblems)
  - If $k < \log_b(a)$, then $T(n) = \Theta(n^{\log_b a})$ (indicating the divide and conquer part is the bottleneck)
- Proof is algebra heavy, but basically, it considers the tree of subproblems as we break the big problem down until we reach base cases. Then, we sum the work at each level based on the combine step's runtime.
- Use it for Divide and Conquer runtime analysis

Recurrences

- Recurrence relations are equations that rely on previous terms to solve.
- One example is $T(n) = T(\frac{n}{3}) + T(n-1) + \log(n)$, where we need to know $T(\frac{n}{3})$ and $T(n-1)$ (two terms prior to $T(n)$) to compute $T(n)$ itself.

- Some but not ALL recurrences can be solved by Master Theorem. If it can be solved via Master Theorem, you can just cite it.
- For a recurrence that can't be solved via Master Theorem, it helps to expand or create a tree diagram. This can make it easier to guess the pattern.
- Be warned that the expansion method or tree method on its own does NOT constitute a rigorous proof.
- You have to use induction to rigorously prove a recurrence. There are two ways to go about this:
  - Suppose you guess $T(n) = O(f(n))$ for some function $f(n)$. For the base case, you can assume $T(n)$ is a constant for sufficiently small n. Then, you need to pick a constant c s.t. $T(n) \leq c \cdot f(n)$ for all positive integers n, where trying to go about the inductive step should help guide what c should be.
  - The other way is to combine expansion and induction. Have a claim about what the formula looks like the i^{th} time you expand. Prove the base case and relate the i^{th} expansion to the $(i+1)^{th}$ expansion (for an arbitrary i) to prove the induction step.

Divide and Conquer

- An algorithm paradigm with 3 steps:
  - Divide: divide the problem into subproblems
  - Conquer: recursively solve each subproblem
  - Combine: combine the solved subproblems to solve the big problem

Convex Hull

- Given a set S of n points, we seek to find CH(S), defined to be the tightest convex polygon including enclosing all n points.
- We note a segment (a, b) (between points a and b) lies on CH(S) if and only if all other points are on the same side of (a, b).
- If we recursively solve both halves, the cost is $2T(\frac{n}{2})$ and then we just need to combine.
- We connect the two completed halves by considering all lines between points that cross the cut. The one with the highest intersection point with the cut should be added. Same with the lowest intersection point.
- The naive way to do the combine step is draw all possible lines, which takes $O(n^2)$ time. Then, the runtime is $T(n) = 2T(\frac{n}{2}) + O(n^2)$, which is $O(n^2)$ by Master Theorem, where we're in the case that the combine step is the bottleneck.
- So we should improve the combine step: As it turns out, the highest intersection point can be found by fixing the point on the left and varying the one on the right until we find a local maximum. Repeat but with the roles of the sides switched.
- This new combine step takes $O(n)$ time.
- So the new runtime is $T(n) = 2T(\frac{n}{2}) + O(n)$ which is $O(n \log n)$, an improvement.

Matrix Multiplication

- Our goal is to multiply two $n \times n$ matrices A and B together.
- We can divide A and B into 4 pieces each of size $n/2 \times n/2$. Then, we recurse on 8 products involving $n/2 \times n/2$ matrices.

- Runtime is $T(n) = 8T(\frac{n}{2}) + O(n^2)$, where the n^2 comes from assembling the computed submatrices. By Master Theorem, this is $O(n^3)$, where the subproblems are the bottleneck. So we should try to see if we can do it in fewer subproblems
- It turns out we can get it done with 7 subproblems instead of 8, so it becomes $O(n^{\log_2 7})$, which is an improvement.

Dynamic Programming

- A technique where solving a bigger problem is reliant on solving a smaller version of said problem.
- Memoization is used to avoid redundant re-computation.
- Requires base cases, just like recursion. They often involve edge cases that "don't make sense" like picking from an empty set or picking activities from $[(n+1)..n]$ (a nonsensical array range).
- Runtime is usually in the form (number of subproblems) $\times$ (time to combine subproblems). There are rare exceptions though.
- Dynamic Programming relies on a high level question to help us search through the solution space. Properties of a good high level question:
  - All possible cases are covered by at least one of the answers to the question
  - There are finitely many possible answers
  - For each answer, the subproblem generated is a smaller version of the original (or a related problem)

Weighted Activity Selection

- Problem details:
  - Inputs: n activities that come with: a start time s_i, an end time t_i, and a reward r_i
  - Output: $S \subseteq [n]$ s.t. if $i, j \in S$ (and $i \neq j$), then $[s_i, t_i] \cap [s_j, t_j] = \emptyset$
  - Objective function: Maximize $\sum_{i \in S} r_i$
- Essentially, we want to select activities such that we maximize our reward and no activities overlap in time.
- High Level Question: Do we want to pick activity i? If so, pick it and get the reward, eliminating the activities that overlap with it. If not, skip it and move on to $i+1$.
- First, let $G(i)$ denote the max reward for activities $[i..n]$. Let $j^*(i)$ denote the first activity after activity i that doesn't overlap with it (set it to $n+1$ if no activities after activity i don't overlap with it). Our output is $G(1)$.
- Then, $G(i) = 0$ if $i > n$ (for our base cases) and $G(i) = \max\{r_i + G(j^*(i)), G(i+1)\}$.
  - The former case is when we choose activity i, where we get reward r_i but can't choose anything until $j^*(i)$ (the next activity not overlapping with it), so the rest is $G(j^*(i))$.
  - The latter case is where we skip activity i, so we handle $[(i+1)..n]$, but we get no reward from skipping activity i.
- We can sort the activities in $O(n \log n)$ time and then precompute j^* for each i in $O(n \log n)$ time too (binary search to find the first activity after t_i)
- We have n subproblems and with j^* precomputed, the combine step is $O(1)$. The runtime for the dynamic programming part is $n \times O(1) = O(n)$.

k-center Problem

- Problem details:
  - Input: n towns along the real line with placements $t_1, t_2, \cdots, t_n$, and an integer k
  - Output: A placement for k stations $s_1, s_2, \cdots, s_k$ along the real line
  - Objective function: Minimize $\max_{i\in[n]}\{\min_{j\in[k]}|t_i - s_j|\}$
- Essentially, we want to minimize the worst distance any town has to travel to get to the nearest station.
- High Level Question: How many towns should the first station serve? If we decide s_1 serves the first r towns, it is optimal to place s_1 at the midpoint between t_1 and t_r to minimize the worst case distance any of the first r towns have to travel to, to reach s_1.
- Define $G(a, b)$ to be the minimum worst case distance in serving towns $t_a, \cdots, t_n$ using b stations. Our output is $G(1, k)$.
- Base Cases are $G(i, 0) = \infty$ when $i \leq n$ and $G(i, 0) = 0$ if $i > n$. Also, $G(i, j) = 0$ if $i > n$.
- Otherwise, $G(i, j) = \min_{i\leq m\leq n}\left\{\max\left\{\frac{|t_i - t_m|}{2}, G(m+1, j-1)\right\}\right\}$
- $\frac{|t_i - t_m|}{2}$ comes from serving towns t_i to t_m and $G(m+1, j-1)$ comes from serving the rest after t_m with 1 less station. Then, we minimize over all choices of m (where the first station serves towns t_i to t_m)
- There are nk subproblems, since $a \in [1..n]$ and $b \in [1..k]$. Then, time to combine is $O(n)$ because we consider values of m from i to n (but i can be 1 worst case, meaning we find the minimum of up to $O(n)$ terms). So the runtime is $nk \times O(n) = O(n^2k)$

k-median Problem

- A twist on the k-center problem, where instead, we minimize the sum of the distances from each town to its nearest station.
- The new objective function is to minimize $\sum_{i=1}^{n}\left(\min_{j\in[1..k]}|t_i - s_j|\right)$.
- We reuse the same high level question. But when serving towns t_1 through t_r, we place the station at the median of t_1 through t_r for optimality. So let $M(i, m)$ be the median of $\{t_i, \cdots, t_m\}$.
- $G(i, j) = \min_{i\leq m\leq n}\left\{\sum_{\ell=i}^{m}|t_\ell - M(i, m)| + G(m+1, j-1)\right\}$ with base cases $G(i, 0) = \infty$ for $i \leq n$, $G(i, 0) = 0$ for $i > n$, and $G(n+1, j) = 0$
- We still have nk subproblems, but the combine step is $O(n^2)$: not only do we have to find the minimum over n different values (worst case), but for each value, we have to sum up to n terms. So the runtime is $nk \times O(n^2) = O(n^3k)$

Optimal BSTs

- Problem details:
  - Inputs: n keys in sorted order, each key having a probability p_i associated with it
  - Output: A BST T of the n keys
  - Objective Function: Minimizing the expected lookup time in the BST, that is, minimizing the sum $\sum_{i=1}^{n}\text{depth}_T(i)\cdot p_i$
- High Level Question: Which node should be the root? If we choose node r, then the left subtree of r is a BST of nodes $[1..(r-1)]$ by the properties of a BST, and the right subtree of r is a BST of nodes $[(r+1)..n]$. Finding those subtrees is a smaller version of the original problem!
- Define $G(a, b)$ to be the minimum cost for a BST with keys $[a..b]$. Output is $G(1, n)$.

- Base case is $G(a,b) = 0$ when $a > b$ (the empty set, so no additional cost)
- Otherwise, $G(a,b) = \min_{a \le i \le b}\{G(a,i-1) + G(i+1,b) + \sum_{k=a}^{b} p_k\}$
- To optimize runtime, we should pre-compute $\sum_{k=a}^{b} p_k$ for each $a, b \in [n]$. There are n^2 pairs (a,b) and we sum $b-a$ numbers, which is at most n. So, pre-computing each of those sums is $O(n^3)$.
- Now for the dynamic programming runtime. We have n^2 subproblems, since $a, b \in [n]$. Then, we take the minimum over $b-a$ numbers (at most n numbers), and each number is $O(1)$ time to retrieve (with the p_k sums pre-computed). So the combine step is $O(n)$. Thus, the runtime is $n^2 \times O(n) = O(n^3)$.

Edit Distance

- Problem details:
  - Input: Two strings: x and y
  - Output: The minimum number of operations needed to transform x to y. Acceptable operations are an insertion, deletion, or substitution.
- To make it easier to get the output, we can consider "alignment:" place bots ($\perp$) throughout both x and y. For example, if $x =$ HOUSTON, then we could do HOU $\perp$ S $\perp$ TON.
- Suppose we did place some bots throughout x and y and line up the columns of x and y. Then, the edit distance between the two strings is the number of columns with different characters (including when one character is a bot and the other is not).
- Three possibilities for a column: $x_i - y_j$, $x_i - \perp$, $\perp - y_j$ (we ignore double bots because those are redundant columns).
- Let $G(i,j)$ be the minimum cost for aligning $x[i..n]$ with $y[j..n]$. Output is $G(1,1)$.
- Base cases are $G(i, n+1) = n - i + 1$ and $G(n+1, j) = n - j + 1$ (when we've exhausted one of the two strings and then have to go through the rest of the other string).
- Then, $G(i,j) = \min\{\mathbb{1}\{x_i \neq y_j\} + G(i+1, j+1), 1 + G(i+1, j), 1 + G(i, j+1)\}$. The three terms in the min come from the 3 cases on the column.
- There are n^2 subproblems and $O(1)$ time to combine, so the runtime is $n^2 \times O(1) = O(n^2)$.

§2.2 Test Specific Strategies

- Make sure you communicate your solution with as little ambiguity as possible (such as defining the appropriate variables and functions). It helps to remember what components constitute a complete solution for the type of problem you're working on. Writing up the HW solutions should help with your communication.
- They're very strict on grading for the midterm and will take off points if it misses a crucial element necessary for rigor. As said in the Introduction section, if you didn't actually write it down, they won't bother with regrade requests that are "I meant to say [blank]."
- Heavy emphasis on DP (from what was said in the review session). Remember to say "use memoization" so that you can justify the runtime not involving redundant re-computations.
- Use the objective function as inspiration for the G function. For example, when trying to maximize/minimize cost, we would say "let $G(a,b)$ be the maximum/minimum cost in range $[a..b]$ that fits the constraint."
- If the DP problem involves you choosing things optionally from a set, a good high level question is "Should the first thing in the set be chosen?" Because naturally, choosing it creates restrictions on what other things from the set can be chosen (smaller subproblem), and forgoing is the same problem with one less element in the set (another smaller subproblem).

- Another good way to devise a high level question is to see what you have control over and what you don't. Generally, if you see that one variable has the most control over the others, that's the variable that should be considered for the high level question.
- Knowing why the algorithms presented in-class work the way they do may be vital if they decide to create a question like "Here's Merge Sort but with a twist!"
- The decision trees concept can help get a lower bound on an algorithm implementing a binary model, so remember the parallels between a decision tree and an algorithm's execution.
- Divide and Conquer, like in CIS 1210, is still incredibly tricky. A good idea is to go like: "Suppose I am magically able to solve these subproblems. What do I want to do with them?"
- You may be able to reverse engineer a Divide and Conquer algorithm. For example, if you're given a runtime of $O(n^{\log_2 3})$, realize that it looks like $b = 2$ and $a = 3$ from the Master Theorem, so you can deduce the recurrence should be something like $T(n) = 3T(n/2) + O(n)$. Then, simply interpreting the recurrence (3 subproblems of size $n/2$ with $O(n)$ time to combine) could help you reverse engineer the solution.
- Remember that $O(\log n)$ suggests a binary search like algorithm, while $O(n \log n)$ suggests the recurrence $T(n) = 2T(n/2) + O(n)$.
- Unlike CIS 1210, they will **NOT** provide Master Theorem for you on the test, so you will have to memorize it. Luckily, the interpretation of the 3 cases (seeing if the bottleneck is the subproblems, the combine step, or a tie between the two) should help make it easier for the memorization.
- Here are tools you have at your disposal based on the allotted runtime:
  - You can use Select in $O(n)$
  - You can sort an array in $O(n)$ with Radix Sort, provided the range of the array is at most a polynomial in n
  - You can sort any array in $O(n \log n)$ time
- Review the solutions of the HWs and independent HWs. Don't merely know the solution, but also the thought process and tip-offs in coming up with that solution.

§3 Midterm 2

Statistics from Fall 2024 iteration (out of 100 points):

- Maximum: 100
- Mean: 61.37
- Median: 65
- Standard Deviation: 17.48

§3.1 List of Content

Greedy Algorithms

- One flaw of DP is that it explores every potential answer to the problem in the solution space and doesn't rule out blatantly poor choices.
- A greedy algorithm is much faster by only exploring only 1 potential answer to a high level question.
- However, it is harder to prove that the output from a greedy algorithm is actually the best that can be done, and thus, it's not as broadly applicable as DP.

Exchange Argument

- An argument where you take a solution (that is not necessarily optimal) and transform it in some way to get a solution that is just as good, if not, better than the original solution.
- This is helpful because it shows an optimal solution might have a specific structure, or that you can rule out solutions that can be improved upon.
- This argument useful for proving the correctness of a Greedy algorithm G by creating a lemma along the lines of this: "For any nonnegative integers k, if there exists an optimal solution OPT that agrees with G on its first k choices, then there exists an optimal solution $\widetilde{OPT}$ s.t:
  - $\widetilde{OPT}$ is feasible (satisfies problem constraints)
  - obj($\widetilde{OPT}$) is at least as good as obj(OPT), if not better (where obj is the objective cost function in the problem)
  - $\widetilde{OPT}$ agrees with G on its first $k+1$ choices
- Through such an Exchange Argument, you can reason that it follows inductively that Greedy matches the optimal solution.
- You also have to show at the end that the length of OPT doesn't exceed the length of G's output (i.e. they have the exact same length).

Unweighted Activity Selection

- This is the same problem as the weighted activity selection, but each activity has reward 1. (We still have to ensure no two selected activities overlap)
- Greedy algorithm G: pick the activity with the first end time, eliminating other overlapping activities in the process. Then, again, pick the activity left with the first end time, eliminating other activities. Keep doing this until there are no activities left.
- Now, we consider a lemma for our proof of correctness. So we suppose there exists an optimal solution OPT that agrees with G on the first k choices. We need to show we can pick $\widetilde{OPT}$ that satisfies the 3 conditions, regardless of the $(k+1)^{th}$ choice of OPT.
- What we can do is, construct $\widetilde{OPT}$ by having it copy G on the first $k+1$ activities, then copy OPT on the rest of the activities.
  - Feasibility: We know G's choices satisfies the constraints. Since OPT is an optimal solution by definition, it satisfies the constraints. So the only issue with feasibility is making sure in $\widetilde{OPT}$, the $(k+1)^{th}$ and $(k+2)^{th}$ choices don't create an issue.
  - We know the $(k+1)^{th}$ in $\widetilde{OPT}$ (same as G) is the first possible finish time, so it finishes before (or at the same time as) the $(k+1)^{th}$ in OPT, which finishes before the $(k+2)^{th}$ in OPT starts. So no feasibility issue here.
  - Objective Function Comparison: Clearly, by the construction, both OPT and $\widetilde{OPT}$ have the same number of activities, so their obj costs are the same.
  - Agrees with first $k+1$ choices: Again, clear from the construction.
- So we know G and OPT agree on the first k choices, for any k. But the last part is showing OPT can't have more choices than G. We can argue this because G always makes choices if it can, but when it stops, that signifies there are no choices left, and thus, nothing more that can be put in OPT.

Huffman

- Problem details:
  - Input: An alphabet of n characters with each character having a frequency p_i associated with it
  - Output: A binary tree T with n character nodes

  - Objective function: $\text{obj}(T) = \sum_{i=1}^{n}(p_i \cdot \text{depth}_T(i))$
  - Constraints: there should be no ambiguity, meaning two different messages should never have the same encoding (basically, we always want to get back the original message after decoding)

- No ambiguity is equivalent to no two encoding strings being prefixes of one another. In binary tree language, this is equivalent to each character node being at a leaf (if a character node were at an internal node, it blocks the path to another leaf, violating the prefix free part).
- Every internal node should have 2 children. An internal node having only 1 child is unoptimal as it protracts that branch for no reason, which is unoptimal for the objective function.
- An Exchange Argument can be done to show the nodes with the two lowest frequencies should be at the bottom level: if they aren't, we can switch them to the bottom and show there is a decrease in the objective function as a result of the switch.
- The algorithm essentially takes all n nodes and sorts them by the frequencies $p_1 \geq p_2 \geq \cdots \geq p_n$. It repeatedly takes the nodes N_1 and N_2 with the two lowest frequencies, merges them into a node with frequency being the sum of the frequencies of N_1 and N_2 and puts it back in the collection of nodes. Rinse and repeat until there's only 1 node left.
- Time is $O(n \log n)$: we sort by frequency first which is $O(n \log n)$. Then, we have n steps, where we take the two smallest nodes, add them, and insert it properly back into the sorted list of nodes (through binary search with is $O(\log n)$). So this is $O(n \log n)$ work.

Job Selection Problem

- Problem details:
  - Input: n jobs to complete with deadlines $d_1, d_2, \cdots, d_n$ (all integers). Each job takes 1 unit of time. Success at job j means starting it before $d_j - 1$. Job j has reward r_j.
  - Output: A subset of the jobs
  - Objective function: Sum of the rewards of the jobs chose
- Let D be the last deadline among $d_1, d_2, \cdots, d_n$. Furthermore, define a dummy job, call it job 0 s.t. $d_0 = \infty$ and $r_0 = 0$. This is used when we don't have any jobs available at a given period of time.
- Greedy Algorithm G: Work backwards from time D. Among the jobs available at time $D-1$, pick the one with the largest reward for time $D-1$. Then, do the same but for $D-2$. Continue until we stop after time 1.
- Now, we consider that lemma that goes along with an exchange argument, where we have OPT agree with G on the first k choices. Then, we have to construct $\widetilde{OPT}$ that is feasible, $\widetilde{OPT}$ is at least as good for the objective function as OPT is, and that $\widetilde{OPT}$ agrees with G on the first $k+1$ choices. (Remember that we consider the first choice to be the job at $D-1$, the second choice to be the job at $D-2$, and so on)
- Let i_m denote the m^{th} choice G makes and let j_m denote the m^{th} choice OPT makes. Have OPT and $\widetilde{OPT}$ copy G on the first k choices.
  - Case 1: $j_{k+1} = i_{k+1}$. Then, have $OPT = \widetilde{OPT}$, which obviously satisfies the 3 conditions.
  - Case 2: $j_{k+1} \neq i_{k+1}$ but OPT does i_{k+1}. Then, have $\widetilde{OPT}$ switch i_{k+1} and j_{k+1} but copy OPT on the rest. It's feasible because G was able to do i_{k+1} for the $(k+1)^{th}$ choice and pushing j_{k+1} to a later choice means doing it earlier, which is fine. Objective function is the same on both OPT and $\widetilde{OPT}$, since they do the same jobs. And $\widetilde{OPT}$ agrees with G on first $k+1$ choices.
  - Case 3: $j_{k+1} \neq i_{k+1}$ but OPT doesn't do i_{k+1}. Then, have $\widetilde{OPT}$ do i_{k+1} for the $(k+1)^{th}$ choice and then copy OPT on the rest of the choices. It's feasible because G was able to do i_{k+1}. Objective function is better for $\widetilde{OPT}$ than OPT, since G choosing i_{k+1} instead of j_{k+1} means i_{k+1} was better. And $\widetilde{OPT}$ agrees with G on first $k+1$ choices.

- Thus, this shows the lemma and then it follows inductively that an optimal solution OPT agrees with G on its first n choices. G and OPT have to be the same length because when G stops, that means no more choices are left.

BFS

- We can define the distance between two vertices s and v through the δ function, signifying the length of the shortest path of edges from s to v. If v is not reachable from s, we say $\delta(s,v) = \infty$.
- BFS is the distance problem but in its simplest form, particularly when there are no edge weights. Thus, a path length is measured by how many edges it uses.
- A simple claim is that in any unweighted graph, if $(u,v) \in E$, then $\delta(s,v) \leq \delta(s,u) + 1$. (essentially a triangle inequality argument)
- BFS uses a 3 color system: namely white for nodes that are undiscovered, gray for nodes in the queue, and black for nodes that are no longer in the queue. We start with all nodes being white.
- We initialize an array L to signify layers.
- First we place a source node s into a queue Q and mark it gray. We set $L[s] = 0$. Then, while Q is non-empty:
  - We dequeue Q to get a node v.
  - For all $(v,w) \in E$ s.t. w is white, we insert w into Q. Then, we set $L[w] = L[v] + 1$ and set w's color to gray.
  - We mark v as black to say we're done with it.
- A node v in layer d satisfies $\delta(s,v) = d$ for any d. This can be shown via induction on the layers by saying all of layer d is in the queue before anything in layer $d+1$. The base case is the start when only s is in the queue (and s is the only node that can have distance 0 from s), so we have all of layer 0 in the queue. Then, assuming we have all of layer d in the queue already, when we discover new neighbors of the nodes in layer d, we increase the layer number, so we have all of layer d before anything in layer $d+1$.
- So when BFS dequeues v and discovers a white node w, we have that $\delta(s,w) \leq \delta(s,v) + 1$ from the claim earlier. But equality has to occur, since w was undiscovered and that cannot happen if it was in the same layer (or a prior layer) as v.
- Runs in $O(n+m)$ time, where n is the number of nodes and m is the number of edges.

DFS

- Another graph traversal that uses the same coloring system as BFS. However, instead of a queue, it uses a stack.
- In this way, we create a more ambitious graph search than BFS that aggressively tries to go out as far as it can until it reaches a dead end. Then, it recursively backtracks to explore routes that it missed.
- Also runs in $O(n+m)$ time.

MST

- Problem details:
  - Input: Graph $G = (V,E)$ that is undirected, weighted, and connected, along with a weight function $w : E \to \mathbb{R}$
  - Output: A spanning tree T, which is a subgraph of G that is a tree and connected. Furthermore, T must have the same vertex set as G. The goal is to find such a tree T that minimizes $\sum_{e \in T} w(e)$.

- Define a cut of a graph to be a subset $S \subseteq V$. Furthermore the edges that are "cut" are given by the set $\{(u,v) \in E \mid u \in S, v \notin S\}$ (essentially the edges with endpoints on opposite sides of the cut).
- Cut Lemma: For any cut $S \subseteq V$, the minimum weight edge crossing the cut is in the MST T.
- The proof of above is essentially an exchange argument: we define e^* so that it is the minimum edge weight crossing the cut but we don't use it in T. If we added e^* to T, we would create a cycle (since a tree is maximally acyclic). Then, if we were to trace the cycle around, there is forced to be another edge e crossing the cut that was used in the original T. But since e crosses the cut and e^* is of minimum weight crossing the cut, removing e and adding e^* is better for the cost. And this exchange also preserves the fact that T is a tree. Essentially, the argument is "Why bother using e when e^*, the lightest edge crossing the cut, does the same job for us but better?"
- This leads to Prim's algorithm that, at a high-level, repeatedly proposes cuts in order to add more edges to the MST:
  - We maintain a set S, which only starts with an arbitrary node s. We also have a min heap H of edges crossing the cut sorted by edge weight. We start by adding all edges with s as an endpoint.
  - Then, we repeat the following while H is non-empty:
    - We extract an edge (x,y) from H, where we label x and y s.t. $x \in S$ and $y \notin S$ (we can easily switch the labeling if need be). Then, add y to S. Furthermore, check all neighbors z of y.
    - If $z \in S$, delete (y,z) from H, as it no longer crosses the cut. If $z \notin S$, add (y,z) to H as the addition of y to S means that edge now crosses the cut.
  - Runtime: Each vertex v eventually has its own iteration where it gets added to S. We iterate through v's neighbors (which means $\deg(v)$ iterations), where we do a heap operation (one such operation is $O(\log m)$, since H stores edges). So v's iteration is $\deg(v) \cdot O(\log m)$. When summing over all $v \in V$, the $\deg(v)$ becomes $O(m)$ by Handshaking Lemma, so the total runtime is $O(m \log m)$.
- Alternatively, we can use Kruskal's algorithm:
  - We first sort edges by weight in increasing order. Then, we maintain connected components, where we start with each vertex being its own CC.
  - We then repeatedly process the edges in order: if the edge connects 2 CCs, we include it and merge the CCs. Otherwise, we ignore the edge and move on to the next one.
  - Runtime: We assume we can determine whether two CCs are the same in $O(\log n)$ and we can merge two CCs in $O(\log n)$ as well via Union-Find. Then, it's $O(m \log m)$ to sort the edges via Merge Sort. Then, we have m iterations, where we need to verify CCs and possibly merge them, which is at most $O(\log n)$ per iteration. So the total runtime is $O(m \log m) + m \cdot O(\log n) = O(m \log m)$ (where $O(\log n) = O(\log m)$ from $m \leq n^2$).

Union-Find

- A data structure that maintains a disjoint partition $C_1, C_2, \cdots, C_\ell$ of a set V.
- Each set is a tree with a root node representing said set. Essentially, for every node in the set, there will be a path of nodes that eventually reach the root.
- There's no downside to having multiple nodes point to the same node, since we're traversing the tree to the root rather than descending away from the root.
- For the Find(u) operation, we simply return the C_k s.t. $u \in C_k$ by traversing nodes from u until we reach a root node.
- We can take the union of sets C_i, C_j by finding the roots of both, call them r_i, r_j, respectively. We change the next node of r_i to be r_j if C_j is a taller tree than C_i (or vice versa if C_i is the taller tree). If there is a tie, either works. This merges the trees.

- We can show by induction that the height of any tree C_i is at most $\log_2(|C_i|)+1$, particularly using the fact that the tree only increases in height when we take the union of two trees that are the same height.
- Through the induction, Find and Union become $O(\log n)$ (since $|V| = n$).
- Path Compression: Whenever we use the Find(u) method, for every node along the path from u to its root, change its pointer to the root. This speeds up future lookups.

Dijkstra

- We return to the shortest path problem, but now, we have weights assigned to each edge. For now, edge weights will be nonnegative.
- Problem details:
  - Input: graph $G = (V, E)$, source node $s \in V$, weight function $w : E \to \mathbb{R}_{\geq 0}$
  - Output: The values of $\delta(s, v)$, $\forall v \in V$. Here, we define $\delta(s, v)$ to be the minimum of $\sum_{e \in p} w(e)$ over all paths p from s to v.
- We need to maintain an array D, where $D(v)$ (for some $v \in V$) represents our guess for the best distance.
- This is how we expand S: if $v \notin S$ is a vertex s.t. $D(u)+w(u, v)$ is minimized over all $u \in S$ and $(u, v) \in E$, then v can be added to S with $D(v) = D(u) + w(u, v)$.
- The idea is we can get from s to v by using a path p that goes from s to u in $D(u)$ distance and then uses (u, v) for a total cost of $D(u) + w(u, v)$. Since $s \in S$ but $v \notin S$, any other path p' from s to v has to leave the cut S at some point. If p' leaves S at $(x, y) \in E$, where $x \in S$ and $y \notin S$, that incurs a cost of $D(x) + w(x, y)$, which is already no better than $D(u) + w(u, v)$, since u and v are supposed to minimize that quantity. Then, the rest of the route from y to v is nonnegative, since edge weights are nonnegative, so the cost of this new path doesn't improve.
- With this in mind, we can maintain a heap H with vertices $v \notin S$ for which $\exists (u, v) \in E$ s.t. $u \in S$. The key for v will be $\min\{D(u) + w(u, v) \mid u \in S, (u, v) \in E\}$.
- To maintain H, at each step we extract the min v and add it to S (to signify we found the true value of $D(v)$, as discussed in the minimization idea). Then, we need to add new vertices that should go in H, so we check v's neighbors. If $z \notin S$ is a neighbor of v:
  - If z is not in H, we add z to H with key $D(v) + w(v, z)$.
  - Otherwise, we decrease the key of z to $D(v) + w(v, z)$ if this value is smaller than the current key of z.
- Runtime: Heap size is n because it stores vertices. We do at most n extractions from the heap, one for each vertex. So this is $O(n \log n)$. We have as many chances to decrease a key as we do checking the neighbors of some vertex, which is $O(m)$ chances by Handshaking Lemma. So this is $O(m \log n)$. Total is $O((n + m) \log n)$.

Bellman-Ford

- Problem details:
  - Input: graph $G = (V, E)$, source node $s \in V$, weight function $w : E \to \mathbb{R}$. There could be negative edge weights, but no negative weight cycles.
  - Output: The values of $\delta(s, v)$, $\forall v \in V$.
- We only care about simple paths (no repeated vertices): if a vertex is repeated, then we have a cycle, which is of nonnegative weight. So we can remove the cycle and improve (or not change) the path weight. Thus, the optimal path should be simple, implying it has at most $n - 1$ edges.

- We approach with Dynamic Programming with the following top-level question: What was the last edge on the shortest simple path from s to v which uses at most k edges?
- Define $G(v,k)$ to be the length of the shortest path from s to v using at most k edges, where $k \in [0..(n-1)]$.
- Base cases are $G(s,k) = 0$, $\forall k \in [0..(n-1)]$ (because no negative cycles) and $G(v,0) = \infty$, $\forall v \in V \setminus \{s\}$.
- Then, we have $G(v,k) = \min(\{G(v,k-1)\} \cup \{G(u,k-1) + w(u,v) \mid u \in V \text{ s.t. } (u,v) \in E\})$. Basically, we either forgo the k^{th} edge or we use the k^{th} edge by checking what edges go into v and examining their respective costs.
- We promised there are no negative weight cycles, but Bellman-Ford can innately detect them anyway. In fact, there are no negative weight cycles $\iff$ the n^{th} row of the table is the same as the $(n-1)^{th}$.
  - The $\Rightarrow$ direction follows from the fact that we only care about simple paths when there are no negative weight cycles, so allowing an n^{th} edge will not change anything.
  - The $\Leftarrow$ direction takes an arbitrary cycle $(c_1,c_2),(c_2,c_3),\cdots,(c_{\ell-1},c_\ell),(c_\ell,c_1)$. From the n^{th} row being the same as the previous, we require for an arbitrary $i \in [\ell]$,

    $$G(c_i, n-1) = G(c_i, n) \leq G(c_{i-1}, n-1) + w(c_{i-1}, c_i),$$

    where the inequality comes from the min in how we defined G. Then, by summing the inequalities, we ultimately arrive at $0 \leq \sum_{i=1}^{\ell} w(c_{i-1}, c_i)$.
- Runtime: To compute the subproblem $G(v,k)$, we take the minimum over $\text{in}(v)$ choices, so the subproblem is $O(\text{in}(v))$ work. Summing that over all $v \in V$ and $k \in [0..n]$ (to account for all choices of the input), the $O(\text{in}(v))$ becomes $O(m)$ when summed over all $v \in V$ by Handshaking Lemma. Then, summing over the $O(n)$ different values of k gives an $O(nm)$ runtime.

Floyd-Warshall

- The previous shortest paths problems were all from a single source node s. But now, we will try to get $\delta(u,v)$, $\forall u,v \in V$. Edge weights can be negative but no negative weight cycles.
- Arbitrarily number the vertices $1,2,\cdots,n$. For $u,v \in V$ and $k \in [n]$, define $G(u,v,k)$ to be the shortest path from u to v s.t. all internal nodes (non-endpoints) of the path are from $\{1,2,\cdots,k\}$.
- High level question: Does the shortest path from u to v with all internal nodes from $\{1,2,\cdots,k+1\}$ actually use $k+1$?
  - NO: Then, $G(u,v,k+1) = G(u,v,k)$ by discarding $k+1$.
  - YES: $G(u,v,k+1) = G(u,k+1,k) + G(k+1,v,k)$ because the path breaks into two parts at $k+1$ where the remaining internal nodes in both parts has to use vertices from $\{1,2,\cdots,k\}$.
- Base Cases:

$$G(u,v,0) = \begin{cases} 0, & \text{if } u = v \\ w(u,v), & \text{if } (u,v) \in E \\ \infty, & \text{otherwise} \end{cases}$$

- Runtime: There are n^3 subproblems and combine takes $O(1)$ time. Thus, the runtime is $O(n^3)$.

Johnson

- Another stab at the All Pairs, Shortest Paths problem with a faster algorithm.
- Some intuition: what if we could transform the graph in a way s.t. all weights are nonnegative, while preserving the shortest paths? That way, we can run Dijkstra's n times (one for each vertex).
- Algorithm:

  - Run Bellman-Ford at an arbitrary source s. This outputs $D(u)$ $\forall u \in V$ (length of shortest path from s to u).
  - Define a new weight function $\widetilde{w} : E \to \mathbb{R}$ s.t. $\widetilde{w}(a,b) = w(a,b) + D(a) - D(b)$, $\forall (a,b) \in E$.
  - Then, run Dijkstra's at each of the n vertices to get shortest paths from every possible source.

- First, we need to prove that $\widetilde{w} \geq 0$ always so that Dijkstra's is even allowed. We know $D(a) + w(a,b)$ is some path from s to b, so it's at least as big as the shortest path, meaning it's $\geq D(b)$. So

$$\widetilde{w}(a,b) = w(a,b) + D(a) - D(b) \geq D(b) - D(b) = 0.$$

- Now, we show shortest paths are preserved by the transformation: Let p be any path from a to b so we have $p = \{(v_0, v_1), (v_1, v_2), \cdots, (v_\ell, v_{\ell+1})\}$, where $v_0 = a$ and $v_{\ell+1} = b$. Then,

$$\widetilde{w}(p) = \sum_{i=0}^{\ell} \widetilde{w}(v_i, v_{i+1}) = \sum_{i=0}^{\ell} [w(v_i, v_{i+1}) + D(v_i) - D(v_{i+1})] = \sum_{i=0}^{\ell} [w(v_i, v_{i+1})] + \sum_{i=0}^{\ell} [D(v_i) - D(v_{i+1})]$$

$$= w(p) + D(a) - D(b).$$

  So all paths from a to b have their weight increased by $D(a) - D(b)$ (a fixed constant when a and b are fixed). So all paths from a to b are equally affected, meaning the shortest path stays the same.

- However, we have to transform the path weights back to the original weights, so the shortest path from a to b (in the original graph) is $\min_{p: a \to b} [\widetilde{w}(p)] + D(b) - D(a)$.

- Runtime: Running Bellman-Ford once is $O(nm)$ and then transforming the edges from w to $\widetilde{w}$ is $O(m)$. Then, Dijkstra's n times is $O(n(n+m)\log n)$. So the runtime is $O(n(n+m)\log n)$. This is faster than Floyd-Warshall.

Network Flows

- Problem details:
  - Input: Directed graph $G = (V, E)$, source $s \in V$, sink $t \in V$, capacity function $c : E \to \mathbb{R}_{\geq 0}$.
  - Output: A flow function $f : E \to \mathbb{R}_{\geq 0}$ satisfying two main properties:
    * Capacity constraint: $0 \leq f(e) \leq c(e)$, $\forall e \in E$.
    * Conservation of flow: $\forall v \in V \setminus \{s, t\}$, the incoming flow given by $\sum_{(u,v) \in E} f(u, v)$ is equal to the outgoing flow given by $\sum_{(v,w) \in E} f(v, w)$.
  - Objective: Maximize the flow in the graph, which is given by $|f| = \sum_{(v,t) \in E} f(v, t)$. Equivalently, maximize $|f| = \sum_{(s,v) \in E} f(s, v)$.

- An analogy is to think of this as a pipe network in trying to route water from s to t. The pipes are edges with various widths and we can't overload their capacities (hence, the capacity constraint). We also can't add water into the system anywhere besides s, and we can't remove water from anywhere in the system besides t (hence, the conservation of flow).

- The network flow problem has applications in max bipartite matching and vertex capacities.

- Note that if we find an (s,t)-path p where every edge on p has some capacity left, we can push more flow along p. We can find such a path via BFS. In particular, if we increase f by some fixed quantity for each $e \in p$, it satisfies the flow function properties, as long as none of the edges exceed capacity. So the most the "fixed quantity" can be is $\min\{c(e) - f(e) \mid e \in p\}$.

- However, if we mindlessly and greedily pushing flow whenever we can, we may make mistakes. When an edge becomes full capacity, it will block off (s,t)-paths. So we need a way to undo bad decisions.

Ford-Fulkerson

- To help encode the concepts of "capacities left" and "changing our mind" in the same entity, we introduce the residual graph G_f.
- Once we have G, c, and f fixed, we can define its residual graph $G_f = (V, \widetilde{E})$ (so the vertices are the same, but the edge set is new) and a new capacity function $\widetilde{c_f} : \widetilde{E} \to \mathbb{R}_{\geq 0}$.
- For each $(u, v) \in E$, create two corresponding edges to (u, v) in $\widetilde{E}$:
  - Type 1 edges: $(u, v) \in E \implies (u, v) \in \widetilde{E}$ with $\widetilde{c_f}(u, v) = c(u, v) - f(u, v)$. These are forward edges with weight equal to how much we can increase the flow by before we overload the edge's capacity.
  - Type 2 edges: $(u, v) \in E \implies (v, u) \in \widetilde{E}$ with $\widetilde{c_f}(u, v) = f(u, v)$. These are back edges with weight equal to how much we can change our mind on the flow on (u, v) (which is of course, limited by how much flow we sent on (u, v) in the first place).
- Suppose we find an (s, t)-path p in G_f with minimum edge weight Δ. Then, if $(u, v) \in p$ is a type 1 edge in $\widetilde{E}$, increase $f(u, v)$ by Δ in G. Otherwise, if $(u, v) \in p$ is a type 2 edge, decrease $f(v, u)$ (the reverse) by Δ in G. One can show this transformation not only satisfies the necessary properties for a flow function but also increases $|f|$ by Δ.
- This idea leads to the Ford-Fulkerson algorithm, where we start at $f(e) = 0$, $\forall e \in E$ (which is always a feasible flow). Then, we repeat the following:
  - Create G_f and find an (s, t)-path p not using anything edges of weight 0. Let Δ be the minimum edge weight in p.
  - Push Δ extra flow into the graph.
  - Otherwise, if no such path p exists, terminate.
- Runtime: If we assume we have integer flows and an integer capacity we function, each iteration improves the flow by at least 1 (if it doesn't improve at all, we terminate). So if $|f^*|$ is the true max flow, we have at most $|f^*|$ iterations. Furthermore, each iteration involves creating G_f and running BFS, which is $O(m)$ (the m term dominates n, since the graph is going to be connected; disconnected nodes are useless in a flow network). So the runtime is $O(|f^*|m)$.
- Proof of correctness is involved and will be covered in the Min-Cut section.

Min-Cut

- Consider an (s, t)-cut, meaning a subset $S \subset V$ of the vertices s.t. $s \in S$ but $t \notin S$.
- Consider the sum of the capacities of the edges crossing this cut (in the direction leading out of the cut), that is, $\sum_{u \in S, v \notin S} c(u, v)$. This can be referred to as the capacity of S, denoted as $\text{cap}(S)$.
- The cut's capacity is essentially a "bottleneck" limiting how much flow can pass through a "zone" in the graph. In fact, for any flow function f and cut S, we can show $|f| \leq \text{cap}(S)$.
- Note that the conservation of flow leads to

$$|f| = \sum_{\substack{(u,v) \in E, \\ u \in S, v \notin S}} f(u, v) - \sum_{\substack{(w,u) \in E, \\ w \notin S, u \in S}} f(w, u)$$

  by initially summing the outgoing flow minus the incoming flow over all vertices in S and having many terms cancel. The second sum is ≤ 0, since all the f terms are nonnegative. In the first sum, we can bound each term by $f(u, v) \leq c(u, v)$ by what we know about feasible flows, so we have

$$|f| = \sum_{\substack{(u,v) \in E, \\ u \in S, v \notin S}} f(u, v) - \sum_{\substack{(w,u) \in E, \\ w \notin S, u \in S}} f(w, u) \leq \sum_{\substack{(u,v) \in E, \\ u \in S, v \notin S}} c(u, v) - \sum_{\substack{(w,u) \in E, \\ w \notin S, u \in S}} 0 = \text{cap}(S).$$

- If we can find a flow f^* and (s,t)-cut S^* for which $|f^*| = \text{cap}(S^*)$, then we cannot increase $|f|$ any further (since the flow always has to be less than the capacity of every cut). That would show maximality of a flow.
- Reuse the formula we derived before:

$$|f| = \sum_{\substack{(u,v)\in E,\\ u\in S, v\notin S}} f(u,v) - \sum_{\substack{(w,u)\in E,\\ w\notin S, u\in S}} f(w,u).$$

  But now, set S to be the BFS tree when Ford-Fulkerson gets stuck (meaning no (s,t)-path from s to t using non-zero edges). This means anything in S is reachable from s and anything NOT in S is NOT reachable from s.
- Consider the $(u,v) \in E$ s.t. $u \in S$ and $v \notin S$. The fact that u is reachable but not v and $(u,v) \in E$ would have to mean $\widetilde{c_f}(u,v) = 0$ to have BFS ignore that edge. Then, $f(u,v) = c(u,v)$.
- Next, consider the $(w,u) \in E$ s.t. $w \notin S$ and $u \in S$. The corresponding back edge $(u,w) \in \widetilde{E}$ would have to have weight 0 to prevent w from being reachable from u (and thus s). That would mean $\widetilde{c_f}(u,w) = 0 \implies f(w,u) = 0$.
- The above sum becomes

$$|f| = \sum_{\substack{(u,v)\in E,\\ u\in S, v\notin S}} c(u,v) - \sum_{\substack{(w,u)\in E,\\ w\notin S, u\in S}} 0,$$

  which is equivalent to $|f| = \text{cap}(S)$.
- So thus, when we obtain a G_f for which we get stuck, that means we have reached the maximum flow. This proves the correctness of Ford-Fulkerson.

§3.2 Test Specific Strategies

- Greedy algorithms are *arguably* the trickiest topic for this midterm. They rely on a rigid principle for making one choice at a time, and not only do you have to feel convinced it's optimal, you have to prove it too. That means you have to be able to shoot down faulty Greedy strategies via counterexamples. Even if you think you have the correct one, the proof has a lot of parts.
- Luckily, when I took the course, the professor said he will mitigate it by: either giving you the Greedy strategy outright and asking you to prove it, or the Greedy strategy will be very obvious.
- When going from OPT to $\widetilde{OPT}$, you have to be careful about $\widetilde{OPT}$ mindlessly copying G on the first $k+1$ choices and then copying OPT on the rest. The problem is $G[k+1]$ could also show up in OPT but later than the $(k+1)^{th}$ spot, resulting in the entry for $G[k+1]$ appearing twice in $\widetilde{OPT}$. This can result in an infeasible $\widetilde{OPT}$, since the problem may not allow for duplicate entries in a solution. This tricked a lot of people on the midterm.
- The general way you should construct $\widetilde{OPT}$ is have it copy G on the first k choices (same as OPT on the first k choices). That is mandatory. Then, you divide up into cases:
  - If $OPT[k+1] = G[k+1]$, then have $\widetilde{OPT}$ copy OPT entirely (even beyond the k^{th} choice).
  - If $OPT[k+1] \neq G[k+1]$ but $G[k+1]$ appears in OPT later than the $(k+1)^{th}$ choice (meaning $OPT[j] = G[k+1]$ for some $j > k+1$), then switch $OPT[j]$ and $OPT[k+1]$. *Then*, copy the rest of OPT to $\widetilde{OPT}$.
  - If $OPT[k+1] \neq G[k+1]$ but $G[k+1]$ doesn't appear in OPT at all, have $\widetilde{OPT}[k+1] = G[k+1]$. Then, have $\widetilde{OPT}$ copy the rest of OPT.
- Not all Greedy proofs need these 3 cases, but many do, especially if duplicate choices in the same solution are not allowed. Of course, for each of the 3 cases, you need to show the 3 things ($\widetilde{OPT}$ is feasible, $\widetilde{OPT}$ is at least as optimal as OPT, and $\widetilde{OPT}$ agrees with G on the first $k+1$ choices). You'll also need to justify why G and OPT are the same length (usually based on the fact that G stops when it can no longer make choices).

- You can always assume G and OPT are feasible, so "This choice is feasible, since Greedy did it" counts as sufficient justification.
- The *hardest* topic would probably be flow networks, or moreso, applications of that problem. They require clever interpretations of "flow" like in the bipartite matching example.
- Like in the bipartite matching example, flows where the flow on each edge is either 0 or 1 has a combinatorial interpretation. A 1 means you select something, while a 0 means you don't (like how a 1 in bipartite matching means we select this connection and 0 means we don't).
- Remember the max flow problem is equivalent to the min-cut problem. So if the problem tasks you with finding some minimum bottleneck, it may be easier to think of it as a min-cut problem while still using a max-flow algorithm.
- With the trickiest and hardest topic covered, the other algorithms are fairly rudimentary in how you should go about studying them: make sure you understand the moving parts and why they work, since you may be asked to modify such an algorithm. Also understand where the runtime comes from.
- Some examples of important moving parts are the Cut Lemma in Prim's algorithm (which allows a greedy strategy for the algorithm to choose the edges) and the high level questions for Bellman-Ford and Floyd-Warshall.
- If you see a runtime of $O((n+m)\log n)$, think back to Dijkstra's and Prim's, where you propose cuts, keep track of a heap, and greedily chose the best edge crossing a cut.

§4 Final

Statistics from Fall 2024 iteration (out of 100 points):

- Maximum: 93
- Mean: 48.77
- Median: 49.5
- Standard Deviation: 17.4

§4.1 List of Content

The final is cumulative, so the content from Midterm 1 and Midterm 2 is fair game.

Capacity-Scaling

- The issue with Ford-Fulkerson is that its runtime depended on the output (namely $|f^*|$). It was slow because we only made progress by at least 1 flow unit per iteration, so we need to be more ambitious with improving the flow to improve the runtime.
- Luckily, we can reuse the work from the POC of Ford-Fulkerson to say that whenever there's no (s,t)-path in G_f (using edges with non-zero weight), we know we reached the optimal flow.
- Suppose the max flow problem is the same as the original, but with one change: we have that $c : E \to \{1, 2, \cdots, C\}$ for some fixed positive integer C. Then, we can get a max flow algorithm in $O(m^2 \log C)$ time as follows:
  - Start with $f = 0$ everywhere. Initialize $\Delta = C$ (the max possible capacity of any edge).
  - Repeat this until we get stuck: Let $G_f(\Delta)$ be the residual graph containing only edges $(u,v) \in \widetilde{E}$ s.t. $\widetilde{c_f}(u,v) \geq \frac{\Delta}{2}$. If we find such a path, increase flow by $\geq \frac{\Delta}{2}$.
  - When we get stuck, we either terminate if $\Delta < 1$, or we update Δ to $\frac{\Delta}{2}$ if $\Delta \geq 1$.

- The POC follows from the algorithm stopping when we don't have an (s,t)-path at all (since $\Delta < 1$ allows all edges from G_f to be used in $G_f(\Delta)$ besides 0 weight edges). The runtime is where we see improvements.
- The main lemma is that when $G_f(2\Delta)$ has no (s,t)-path, then $\exists$ cut S s.t. $s \in S$, $t \notin S$, and $\text{cap}(S) < |f| + m\Delta$. To prove it, we reuse a formula from the min-cut:

$$|f| = \sum_{\substack{(u,v)\in E,\\ u\in S, v\notin S}} f(u,v) - \sum_{\substack{(w,u)\in E,\\ w\notin S, u\in S}} f(w,u).$$

- Like before, we set S to be the BFS tree obtained when $G_f(2\Delta)$ has no (s,t)-path and use the fact that certain vertices are not reachable in order to deduce values in the sum.
- If we have $(u,v) \in E$ with $u \in S$ and $v \notin S$, that means $\Delta > \widetilde{c_f}(u,v)$ to prevent (u,v) from being crossed in $G_f(2\Delta)$. But since $\widetilde{c_f}(u,v) = c(u,v) - f(u,v)$, this is equivalent to $f(u,v) > c(u,v) - \Delta$.
- If we have $(w,u) \in E$ with $w \notin S$ and $u \in S$, we need to prevent (u,w) from being crossed in $G_f(2\Delta)$. This happens when $\Delta > \widetilde{c_f}(u,w) = f(w,u)$, equivalently $-f(w,u) > -\Delta$.
- Through that formula,

$$|f| = \sum_{\substack{(u,v)\in E,\\ u\in S, v\notin S}} f(u,v) - \sum_{\substack{(w,u)\in E,\\ w\notin S, u\in S}} f(w,u) > \sum_{\substack{(u,v)\in E,\\ u\in S, v\notin S}} [c(u,v) - \Delta] - \sum_{\substack{(w,u)\in E,\\ w\notin S, u\in S}} [\Delta].$$

  Interpreting the RHS, we sum $c(u,v)$ over all edges crossing the cut in the direction leading out of it, so that is $\text{cap}(S)$. However, we sum $-\Delta$ for edges crossing the cut in both directions. Since there are at most m edges, we have at most that many $-\Delta$s in the sum, so it boils down to $|f| > \text{cap}(S) - m\Delta$, equivalently, $\text{cap}(S) < |f| + m\Delta$.
- What can we do with this formula? Remember that any flow cannot exceed the capacity of any cut, so $|f^*| \le \text{cap}(S) < |f| + m\Delta$, implying $|f^*| - |f| < m\Delta$. And this occurs when we got stuck at 2Δ and move on to Δ, where we improve the flow by at least $\frac{\Delta}{2}$. But that inequality meant that when we got stuck at 2Δ, we can now only improve by at most $m\Delta$ before f reaches f^*. So at Δ, the number of iterations is at most $\frac{m\Delta}{\Delta/2} = O(m)$.
- Now, we can finish this. Since Δ starts at C and halves each time, we have $\log C$ many values of Δ. At each value of Δ, we have $O(m)$ iterations (where we construct $G_f(\Delta)$ and run BFS) and each such iteration is $O(m)$ work. So the total is $O(m^2 \log C)$.

Edmonds-Karp

- Yet, another algorithm for the max flow problem that runs in $O(m^2 n)$ time. POC again, relies on the fact that when we have no (s,t)-path, we're done.
- Algorithm:
  - Initialize $f = 0$ everywhere.
  - While there is still an (s,t)-path, find the one that uses the fewest number of edges (regardless of weights as long as no 0 weight edges are used) and push flow.
  - Output f.
- So we know the condition per the POC, but now the question is, how long until we get stuck? Or another way of looking at it is, how do we know there is no (s,t)-path in G_f? Well, if the number of hops in the shortest path in G_f is bigger than $n-1$, then there's no simple path in G_f, meaning no path at all!
- Suppose we follow the algorithm and we have $d(f)$ denote the fewest number of hops in an (s,t)-path. Then, I claim not only does $d(f)$ never decrease, but it must always increase by at least 1 every m iterations.

- To see this, let stage L start the moment $d(f) = L$. When stage L starts, copy and paste G_f to the side and freeze the copied version (meaning don't change it at all) so that it can be used for future reference.
- Imagine the BFS tree and its layers. We can have edges going from one layer to the next, but we can't have a forward edge skipping over a layer (which contradicts what we know about BFS layers). We could have back edges going backwards multiple layers though.
- A shortest path has to keep moving forward layer by layer (without using a back edge) or else, it wouldn't be the shortest. Whenever we find an (s,t)-path, there's always a bottleneck edge (which has minimum capacity left out of all edges on the path) and then it becomes out of commission by pushing flow on that path.
- There are at most m forward edges in the frozen graph and since we always put 1 edge out of commission per iteration, we definitely have to increase $d(f)$ by 1 every m iterations (since it will reach a point where an (s,t)-path has to resort to back edges, meaning the layer system needs to be changed).
- So with that, we have $O(m)$ iterations per stage with each iteration taking $O(m)$ time (for the BFS and making G_f). There are $O(n)$ stages (since L can be at most $n-1$), so our runtime is $O(m^2n)$.

Complexity

- We've been talking about getting algorithms under a certain runtime, but how can we say an efficient algorithm *doesn't* exist? (where "efficient" means polynomial time). The bad news is we don't know any interesting impossibility result, at least in 2024.
- However, we can still look at the computational "hardness" examining a specific class of problems. First, we limit ourselves to decision problems, meaning our input is some language $L \subseteq \{0,1\}^*$ and our output is either Yes or No.
- Then, we have define P to be the set of languages L s.t. $\exists$ algorithm that decides L in polynomial time.
- Also, what problems do we care to solve with computation? Ones that can be verified. Otherwise, I could just lie to you about the answer and if you can't verify if I'm lying, what good is the answer?
- To formalize the concept of *verifying* a language L:
  - If $x \in L$ and $y \in \{0,1\}^*$ is a solution for x, then the algorithm should accept (x,y).
  - Otherwise, if $x \notin L$, then $\forall y \in \{0,1\}^*$, the algorithm should reject (x,y).
- Define NP to be the set of languages L s.t. $\exists$ *verifier* algorithm in polynomial time w.r.t the input size.
- The problem asking if P = NP is a famous open problem you may have heard about. It's clear P $\subseteq$ NP, but we don't know about the other inclusion.

Reductions

- A formal way of relating two problems A and B with the intent to argue that problem B is at least as hard as problem A.
- In particular, problem A reduces to problem B if:
  - There exists a polynomial time algorithm Q that transforms inputs for problem A to inputs for problem B.
  - For any input a to problem A, problem A accepts $a \iff$ problem B accepts $Q(a)$.
- So if problem A reduces to problem B, here's what we can do to solve problem A: we can transform the input of problem A to one of problem B in polynomial time. Then, we use any algorithm for problem B to finish the job.

- Here's the idea when we reduce problem A to problem B: all inputs for problem A can become inputs for problem B, but it is not necessarily true that any input of problem B can be achieved by transforming an input for problem A. So those separate cases for problem B could bog down problem B's overall difficulty extra. But whatever the case is, the inputs for problem B that are achieved by transforming an input for problem A are just as hard as problem A itself. So overall, problem B is at least as hard as problem A.

NP-completeness

- This is a way of describing problems that are the hardest to solve in NP. In general, if you find a problem is NP-complete, you can expect that there is no efficient algorithm to solve it.
- Two conditions are necessary for a problem A to be NP-complete:
  - A has to be in NP, which as we said, there exists a "guess-and-check" verifier algorithm in polynomial time.
  - A has to be NP-hard, which means A has to be at least as hard as every other problem in NP.
- Generally, problems we want to show are NP-complete often involve showing something exists. So to prove the NP step, it would be like going to me and saying "Hey Jeffrey, I found the thing that exists that satisfies the problem's conditions." And then I say back "I don't care how many failed attempts it took you before you obtained that thing in the first place. I just want to be able to verify in polynomial time that the thing actually works."
- The NP-hard step is a lot trickier. Good news is, the Cook-Levin Theorem states that any problem in NP can be reduced to a problem called 3-SAT (which I'll get to later), so 3-SAT is at least as hard as all other NP problems. This implies 3-SAT is NP-hard. (You can look up the proof for Cook-Levin if you're curious, but you do not at all need to understand the proof, just the statement of the theorem)
- It turns out 3-SAT is also in NP, so 3-SAT is NP-complete (meaning it's the hardest to solve in NP). So if you reduce 3-SAT or any other NP-complete problem to problem A, that is sufficient for showing the NP-hard step. You'll build a batch of problems you can use for reductions in the NP-hard step as examples are shown in lecture.

3-SAT

- Problem details:
  - Input: A 3-CNF formula ϕ on n boolean variables $x_1, x_2, \cdots, x_n$ with m clauses. A clause in a 3-CNF formula always involves 3 variables, where each variable is a literal (which means it is in form x_i or $\overline{x_i}$ for some $i \in [n]$). There are ORs ($\vee$) between the 3 variables. Examples of clauses in a 3-CNF formula would be $(x_1 \vee x_2 \vee \overline{x_3})$ and $(x_2 \vee \overline{x_4} \vee x_6)$
  - Decide: Whether or not there exists an assignment of True or False to each of the n variables s.t. all m clauses are simultaneously True (this is called a satisfying assignment).
- Proof of NP: Our guess is the assignment of the n variables. To verify that all clauses are true, it takes constant time to check if 1 clause is true (only 3 variables per clause), so it takes $O(m)$ time to see if all clauses are true. This is polynomial time.
- Proof of NP-hard: I stated before that all problems in NP can be reduced to 3-SAT by the Cook-Levin Theorem, so 3-SAT is NP-hard. At this point, 3-SAT can be used to reduce other problems to prove they are NP-complete.

Independent Set

- Problem details:
  - Input: Graph $G = (V, E)$ and $k \in \mathbb{N}$
  - Decide: If $\exists S \subset V$ of size $\geq k$ s.t. $\forall (u, v) \in E$, $|S \cap \{u, v\}| \leq 1$. In other words, no two vertices in S are connected by an edge.

- We prove Independent Set is NP-complete.
- Proof of NP: Our guess is the the subset $S \subset V$ of size $\geq k$. We brute-force check that $\forall (u,v) \in E$, $|S \cap \{u,v\}| \leq 1$, which is $O(mk)$ time (a polynomial w.r.t the input size).
- Proof of NP-hard: The reduction will be done from 3-SAT. Refer to slides 7 to 20 from here: https://courses.grainger.illinois.edu/cs374/fa2020/lec_prerec/23/23_2_0_0.pdf. Essentially, we create a triangle for each clause and then connect up opposing literals because for any $i \in [n]$, we can't have x_i and $\overline{x_i}$ to be true at the same time.
- So if we have n variables and m clauses, it gets transformed into a graph with $3m$ vertices. So that means $\binom{3m}{2} = O(m^2)$ many edges. Also, for the $k \in \mathbb{N}$ that is the integer input to Independent Set, we use $k = m$. The transformation is clearly polynomial in the input size.
- The validity of the reduction is showing that there is a satisfying assignment for the m clauses and n variables $\iff$ the resulting graph has an independent set of size m.
- ($\Rightarrow$): We have a satisfying assignment. Like the slides say, one of the true variables from each clause can be picked from its corresponding triangle in the graph (we can't pick more than one from the same triangle). So if we do this for each triangle, we have m vertices. Since the assignment was satisfying, there's no opposing literals. And the only edges between different triangles are those with opposing literals, so the m vertices from each triangle don't share edges between them, so it's an independent set.
- ($\Leftarrow$): Suppose we did have an independent set of m vertices. No independent set can use 2 vertices from the same triangle, but we only have m triangles, so we need to have 1 vertex from each triangle. For every vertex we selected, set its corresponding literal to be True. The assignment doesn't have conflicts (like x_i and $\overline{x_i}$ being simultaneously true) because this is an independent set and there's always an edge between x_i and $\overline{x_i}$. Furthermore, since 1 vertex was selected from each triangle, each clause does have at least 1 true literal.
- Like the slides say, we cleverly encoded the features of 3-SAT into Independent Set: we needed to have at least 1 true literal from the clause to be selected, but we can't have opposing literals at the same time, akin to no two vertices in an independent set sharing an edge.

Vertex Cover

- Problem details:
  - Input: $G = (V, E)$, $L \in \mathbb{N}$
  - Decide: If $\exists S \subset V$ of size $\leq L$ s.t. $\forall (u,v) \in E$, $|S \cap \{u,v\}| \geq 1$. In other words, every edge of the graph has to be covered by S at one of its two endpoints (or both).
- Proof of NP: Our guess is the subset $S \subset V$ of size $\leq L$. We iterate through (u,v) and check $|S \cap \{u,v\}| \geq 1$, which can be done in time $O(mL)$.
- Proof of NP-hard: We reduce from Independent Set. Take an input $G = (V, E)$ and k from Independent Set and transform it to $G = (V, E)$ and $n - k$ for Vertex Cover. Clearly a polynomial transformation. Since $|S| \geq k$, we have $|V \setminus S| \leq n - k$.
- To prove the correctness of the reduction, I will show S is an independent set of $G \iff V \setminus S$ is a vertex cover of G.
- Basically the idea to show both directions is to relate what is forbidden between both problems. In particular, the only thing that is forbidden in Independent Set is to have $u, v \in S$ and $(u,v) \in E$. The only thing forbidden in a Vertex Cover T is to have $u, v \notin T$ and $(u,v) \in E$ (where neither endpoint is covered). When $T = V \setminus S$ though, it's saying that it's forbidden to have $u, v \in S$ and $(u,v) \in E$. This is equivalent to the forbidden condition for Independent Set, so this proves both directions in one swoop.
- So Vertex Cover is NP-complete.

Hamiltonian Cycle

- Problem details:
  - Input: Directed $G = (V, E)$
  - Output: Whether or not there exists a simple cycle (no repeated vertices besides the first and last) that goes through all vertices of G
- Proof of NP: Our guess is the cycle and then we walk through it to check that it's simple and we visit every vertex once. This is $O(n)$ time.
- Proof of NP-hard: Oh boy, this is one complicated reduction, so bear with me. Refer to slides 16 to 56 from here: https://courses.grainger.illinois.edu/cs374/sp2021/scribbles/A-2021-04-27.pdf As you can see, we have a first phase which is encoding the assignments of the variables are encoded in these disparate rows of vertices. The direction that the row i is traversed in determines the assignment of variable x_i.
- In the second phase, we add clause vertices c_j and connect them to row i if x_i or $\overline{x_i}$ shows up in clause j. In particular, the orientation of the edges going from row i to c_j and then back to row i matters depending on whether it was x_i that appeared in clause j or $\overline{x_i}$. This is to reward us with the opportunity to visit c_j if we picked the right direction to go down the row, in the same way we're rewarded for a clause being satisfied if we pick the right truth value for a variable.
- The transformation is polynomial: we have the two start vertices, the m clause vertices, and then n rows of vertices representing the variable assignments. In each row, we need at least $2m$ vertices (since a variable assignment can reward us with up to m clauses being satisfied, and we need 2 vertices in the row for each clause: to get to and back from the clause vertex). So we have $2 + m + n \cdot 2m = O(mn)$ vertices in the graph. A naive bound for the edges is $\binom{mn}{2} = O(m^2n^2)$, so this is still all a polynomial in the input size.
- I won't go into excruciating detail about proving the bidirectional, but rather, explain it at a high level.
- On one hand, if we start with a satisfying assignment, we can use that to decide which direction we traverse each row in. The fact that the assignment was satisyfing allows us to visit each of the clause vertices, so we're good there.
- On the other hand, if we start with a Hamiltonian cycle, we have to traverse each row either left or right by the way the graph is designed. We can use the directions to determine an assignment to each of the n variables. Since we were able to visit each of the clause vertices in the cycle, that means each clause is satisfied by our assignment of variables.
- So Hamiltonian Cycle is NP-complete. As you can see, the reductions can be highly sophisticated, but you can still encode the various attributes of one problem into another.

3-Coloring

- Problem details:
  - Input: $G = (V, E)$
  - Decide: $\exists c : V \to \{\text{red}, \text{blue}, \text{green}\}$ s.t. $\forall (u, v) \in E$, $c(u) \neq c(v)$. Basically, we want to color the vertices of G with only 3 colors s.t. no two vertices that share an edge have the same color.
- Proof of NP: We guess the coloring and then check by iterating through the edges. Clearly polynomial time.
- Proof of NP-hard: We reduce from 3-SAT. Refer to pages 83 to 98 from here https://courses.grainger.illinois.edu/cs374/sp2021/scribbles/A-2021-04-27.pdf because this is another complicated reduction.

- We first a reference triangle given by True-False-Base. Then, we have $2n$ vertices, one for each literal x_i and $\overline{x_i}$. We connect x_i to $\overline{x_i}$, as well as all $2n$ vertices to the Base vertex, preventing them from sharing a color with it. And since x_i to $\overline{x_i}$ are connected, they can't have the same color. This forces each $(x_i, \overline{x_i})$ pair to have one vertex be the True color and the other to be the False color. So we have encoded the notion of picking whether x_i is True or False and now, we have to encode a clause being satisfied.
- As you can see on pages 88 to 91, we can create a 3 variable OR gadget from the vertices and connect the right end of that gadget to the Base and False vertices from the reference triangle. This forces the right end of the gadget to be the True color in a 3-coloring.
- It turns out that right end of the gadget can only be the True color in a 3-coloring $\iff$ at least one of a, b, c is the True color. So we can encode the notion of clause satisfiability through at least one of the 3 literals being true.
- For each of the m clauses, we can take the 3 literal vertices in the clause and have them be the a, b, c in an OR gadget. Have a separate OR gadget for each clause.
- Again, I won't go into excruciating detail about the correctness of the reduction, but it's very much alluded to in my walkthrough:
- To get from 3-CNF to 3-Coloring, you color the literal vertices based on whether they were True or False. Then, you are able to color each of the OR gadgets in a way that doesn't create a conflict because at least one of the 3 literal vertices for each OR gadget is the true color.
- To get from a 3-Coloring to 3-CNF, the True color isn't directly given to you. However, you can identify it by checking the right tips of each OR gadget (since all tips of the OR gadget are connected to two vertices of the reference triangle, it is forced for each tip to be the same color). Once you identify the True color, you go through the $2n$ literal vertices and see which ones have the True color. That forms the assignment. It actually works as an assignment, since each OR gadget is 3-colored, which is equivalent to a clause being satisfied. Also, in each literal pair $(x_i, \overline{x_i})$, the vertex that is not the True color has to be the False color (and not the Base color), since each literal vertex is connected to a vertex in the reference triangle.
- As for the reduction being polynomial time, the $2n$ vertices and reference triangle accounts for $O(n)$ vertices. Then, each OR gadget has 6 vertices and there is 1 OR gadget for each clause, so that's $O(m)$ vertices. So we have $O(m+n)$ vertices. A naive upper bound on the edges is $\binom{m+n}{2}$, which is still a polynomial in m and n. So the reduction is polynomial time.
- Hence, 3-Coloring is NP-complete.

Subset Sum

- Problem details:
  - Input: Integers $a_1, a_2, \cdots, a_n \geq 0$ and T.
  - Decide: If $\exists S \subset [n]$ s.t. $\sum_{i \in S} a_i = T$.
- There's a Dynamic Programming algorithm with runtime $O(nT)$. However, this is not a polynomial w.r.t the input size because T requires $\log_2 T$ many bits to express (and T is exponential w.r.t the input size).
- Proof of NP: The guess is $S \subset [n]$. Then, we can do up to n many additions, each of which takes $O(\log T)$ time (since the sum can't exceed T and so we add up to $\log T$ columns of bits). So the runtime is $O(n \log T)$, which is a polynomial.
- Proof of NP-hard: We reduce from 3-SAT. We will create a matrix of digits with $n+m$ columns and $2n+2m$ rows.
- We will have a total of $2n+2m$ integer inputs to the Subset Sum problem. Each integer a_i to the Subset Sum problem will be obtained by reading off the digits in the i^{th} row of the matrix.

- Also, have $T = \underbrace{11\ldots11}_{n \text{ ones}}\underbrace{33\ldots33}_{m \text{ threes}}$, so it has $n+m$ digits total.
- Label the first n columns with $v_1, v_2, \cdots, v_n$ (one for each of the n variables). Label the first $2n$ rows with $x_1, \overline{x_1}, x_2, \overline{x_2}, \cdots, \overline{x_n}$ (one for each of the $2n$ possible literals that could show up). Then, label the last m columns with $c_1, c_2, \cdots, c_m$. Label the last $2m$ rows with $h_{(1,1)}, h_{(1,2)}, h_{(2,1)}, h_{(2,2)}, \cdots, h_{(m,2)}$ (one for each ordered pair (a, b) s.t. $a \in [m]$ and $b \in [2]$).
- Now, here's how the table should be filled out:
  - In column v_i, place a 1 in row x_i and $\overline{x_i}$, $\forall i \in [n]$
  - In column c_j, within the first $2n$ rows (the rows corresponding to literals), place a 1 if the literal appears in clause c_j, $\forall j \in [m]$. Also, in column c_j, place a 1 in rows $h_{(j,1)}$ and $h_{(j,2)}$, $\forall j \in [m]$.
  - Place 0s everywhere else.
- The table is of dimensions $n+m$ by $2n+2m$, so the size of it is a polynomial in n and m, meaning the reduction is polynomial.
- Remember the rows represent our integer inputs, so it remains to choose a subset of the rows to get a sum of T. Note there are at most 5 ones in each column (and 0s everywhere else), so there's no carrying when we add. So it remains to consider the columns separately and have the sum in each column check out in comparison to T.
- The reduction's correctness will be shown by: there exists a satisfying assignment for the n variables and m clauses $\iff$ we can choose a subset of the rows to get a sum of T.
- The first n columns are like a "variable gadget." The first n digits of T are 1, so we have to obtain 1 in each of those n columns. The only non-zero entries in column v_i are at x_i and $\overline{x_i}$ (and we have to pick exactly 1). This encodes the property that only one of x_i or $\overline{x_i}$ is true.
- The last m columns are like a "clause gadget." The last m digits of T are 3, so we have to obtain 3 in each of those m columns. In column c_j, we have $h_{(j,1)}$ and $h_{(j,2)}$ that could give us a 1, and selecting either of those h rows doesn't affect other columns because they only have a 1 in column c_j.
- So those two h rows for c_j are "helper" rows and depending on how many we select, we can obtain 0, 1, or 2 extra ones.
- We need a sum of 3, so we need at least 1 extra one in column c_j, namely from the $2n$ rows with literals. We get a one from those rows only when at least one literal we selected appears in c_j. This encodes the satisfiability aspect of a clause. Also, note the helper rows accommodate when we have 2 true literals from a clause in which we choose 1 extra one to get a sum of 3 in the column (or 3 true literals). The point is, they never accommodate 0 true literals from a clause.
- From the insights, it's pretty clear to get a general idea of both directions for the reduction.
- So subset sum is NP-complete.

§4.2 Test Specific Strategies

- While subjective, the hardest topics are arguably: Divide and Conquer, Greedy algorithms, Flow, and Reductions. Why these four? In the case of Divide and Conquer and Flow, the solution is often hard to come up with. For Greedy algorithms, there's a lot of things that need to be shown, and there are places to make a mistake. Reductions have both of these sources of difficulty.
- Don't underestimate the difficulty of the True/False questions! There's a lot of variety in the material and the fact that the exam is cumulative. Not only that, but you have to be familiar enough with each topic because these questions may involve the machinery of a particular algorithm.

- Expect a Dynamic Programming problem, especially after seeing it in the context of graph algorithms. Regardless of DP being in graphs or not, the principles of devising a High Level Question still applies. Primarily, you want to think about what should be the variables for the G-function, what choices do you have available, and what happens when you pick or forgo a choice.
- Flow algorithms always have the same end condition: when there's no (s, t)-path. But what distinguishes the flow algorithms presented are how they reach that end condition and how progress is measured.
- Remember the problems you can use in a reduction to prove NP-completeness! It could vary which problems you can use depending on the semester, or they may force you to use a specific problem to reduce from.
- For a problem where there's a lot to prove like a Greedy algorithm proof or an NP-complete proof, **write the list of things you need to show before you get to work on the problem**. In particular, for an NP-complete proof, you need to show 4 things: that it's in NP, describe the reduction and say why it's polynomial, the forward direction of the reduction proof, and the backward direction.
- It's easy to get caught up in the reduction (the much harder part) that you forget to show it's in NP. In fact, the professor remarked how it was a common error that people forgot to show it was in NP (presumably for the reason I mentioned).
- To facilitate actually thinking about the reduction, it helps to take your problem that you're reducing **from** and consider the elements comprise it.
  - For example, when we reduced from 3-SAT, we had two primary elements: only one of x_i or $\overline{x_i}$ is true and that each clause had to have at least 1 true literal. We then created "gadgets" to help encode both of those key elements.
  - You can also form your own interpretations of a problem. Take Vertex Cover. The idea is to place k coins on the vertices in order to "hit" or "cover" every edge. So you can transfer that "hitting/covering" idea if you need to reduce from it.