Survivor's Manual for CIS 3200

Jeffrey Saito (Hitman7128)

Last Update: October 30, 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.

§1 Midterm 1

Takes place on October 14.

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

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

§1.1 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 ith 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 jth 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 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 $[(n+1)..n]$.
- Runtime is in the form (number of subproblems) × (time to combine subproblems)
- 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 $i, j \in S$, 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. Our output is $G(1)$.
- Then, $G(i) = 0$ if $i > n$ 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
- 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 of $t_1 - t_r$ 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 $= 0$ 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 $nk \times O(n) = O(n^2 k)$

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 $t_1 - t_r$, we place the station at the median of t_1 and 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 $nk \times O(n^2) = O(n^3 k)$

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 $n^2\times O(1)=O(n^2)$

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 useful for proving the correctness of a Greedy algorithm G that goes 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$(\widetilde{OPT})\ge$ obj(OPT) (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 that Exchange Argument 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.

§1.2 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. 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 (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.

§2 Midterm 2

Takes place on November 18.

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

- Mean:
- Median:
- Standard Deviation:

§2.1 List of Content

BFS

-

DFS

-

MST

-

Dijkstra

-

Bellman-Ford

-

Floyd-Warshall
Network Flows
Ford-Fulkerson
Max-flow and min-cut
Capacity-Scaling
Intro to Complexity
Cook-Levin
Reductions

§2.2 Test Specific Strategies

§3 Final

Takes place on December 17.

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

- Mean:
- Median:
- Standard Deviation:

§3.1 List of Content

§3.2 Test Specific Strategies