Northeastern University
CS5800 – Algorithms
Spring 2024 Semester, Jonathan Mwaura

Problem Set #6

This Problem Set is due at **11:30PM on Monday, 1st, April** and will be submitted on GRADESCOPE and CANVAS(only the codebase).

The problem set will be marked out of 60.
Suggested working schedule:

- Problem 1 - Wed 3/20/24
- Problem 2 - Sat 3/23/24
- Problem 3- leetcode- 3/25 - 3/29
- Review and Submission - Sat 3/30/24

All the Problems apart from problem 3 (part ii) can be carried out as a group assignment. Make sure that you write the name of all your colloborators on top of the page for this problem if you work with others.

Please type (or neatly handwrite) your solutions on standard 8.5×11 paper, with your name at the top of each solution. Ensure that you submit you solutions in one file PDF file on Gradescope. **each problem sets solution should be on in its own individual page, Gradescope will help ensure you submit each solution under its correct problem number**

While a solution must be absolutely perfect to receive full marks, I will be generous in awarding partial marks for incomplete solutions that demonstrate progress.

So that there is no ambiguity, there are two non-negotiable rules. A violation of either rule constitutes plagiarism and will result in you receiving an F for this course.

(a) If you meet with a classmate to discuss any of the Individual Problems, your submission must be an individual activity, done in your own words, away from others. The process of finding a solution might take 3 - 5 iterations or even more BUT you learn from all these attempts and your confidence grows with each iteration.

(b) These problem sets might seem hard on a first look. They are designed to be so. We learn by attempting problems, struggling through them and coming on top. I encourage you to make this learning exercise worth your while. What do I mean? Open the problem sets as early as you get them, then do not look at hints or answers any where (including on the internet and consulting other students for direct answers), give it the best shot you can. If you get stuck come to Professor or TA's office hour and we shall be glad to listen to your rationale and work with you till you are able to tackle the problem sets.

Problem #1 - 20 points

The *Longest Subsequence Problem* is a well-studied problem in Computer Science, where given a sequence of distinct positive integers, the goal is to output the longest subsequence whose elements appear from smallest to largest, or from largest to smallest.

For example, consider the sequence $S = [9, 7, 4, 10, 6, 8, 2, 1, 3, 5]$. The longest increasing subsequence of S has length three ($[4, 6, 8]$ or $[2, 3, 5]$), and the longest decreasing subsequence of S has length five ($[9, 7, 4, 2, 1]$ or $[9, 7, 6, 2, 1]$).

And if we have the sequence $S = [531, 339, 298, 247, 246, 195, 104, 73, 52, 31]$, then the length of the longest increasing subsequence is 1 and the length of the longest decreasing subsequence is 10.

(a) Find a sequence with nine distinct integers for which the length of the longest increasing subsequence is 3, and the length of the longest decreasing subsequence is 3. Briefly explain how you constructed your sequence.

(b) Let S be a sequence with ten distinct integers. Prove that there *must* exist an increasing subsequence of length 4 (or more) or a decreasing subsequence of length 4 (or more).

Hint: for each integer k in the sequence you found in part (a), define the ordered pair $(x(k), y(k))$, where $x(k)$ is the length of the longest increasing subsequence beginning with k, and $y(k)$ is the length of the longest decreasing subsequence beginning with k. You should notice that each of your ordered pairs is different. Explain why this is not a coincidence, i.e., why it is impossible for two *different* numbers in your sequence to be represented by the *same* ordered pair $(x(k), y(k))$.

(c) In class, we unpacked the Longest Common Subsequence (LCS) problem, where we showed that if our two sequences have size n, then our Dynamic Programming algorithm runs in $O(n^2)$ time.

Let S be your 9-integer sequence from part (a), and let S^* be the same sequence where the 9 numbers are sorted from smallest to largest. Using the LCS algorithm from class, determine the length of the longest common subsequence of S and S^*. (Your answer will be 3. Do you see why?)

Now prove the general case. Specifically, if S is a sequence of n distinct integers, prove that the length of the longest **increasing** subsequence of S must equal the length of the longest **common** subsequence of S and S^*, where S^* is the sorted sequence of S.

(d) The results of part (c) immediately gives us an $O(n^2)$ time algorithm to determine the length of the longest increasing subsequence of an input sequence S with n distinct integers. But this is (unsurprisingly) not the optimal algorithm. Your goal is to improve this result.

Given an input sequence S with n distinct integers, design a linearithmic algorithm (i.e., running in $O(n \log n)$ time) to output the length of the longest increasing subsequence of S. Clearly explain how your algorithm works, why it produces the correct output, and prove that the running time of your algorithm is $O(n \log n)$.

If you are unable to come up with an $O(n \log n)$ algorithm, you will receive partial credit for designing an $O(n^2)$ algorithm that is different from the one described in part (c).

Problem #2 - 20 points

For those of you who follow professional sports, you will know that there are many team sports that are clock-based, i.e., the game lasts a fixed amount of time and the winner is the team that scores the most points or goals during that fixed time.

In all of these sports (e.g. basketball, football, soccer, hockey), you will notice that near the end of the game, the team that is behind plays very aggressively (in order to catch up), while the team that is ahead plays very conservatively (a practice known as "stalling", "stonewalling", and "killing the clock").

In this problem we will explain why this strategy makes sense, through a simplified game that can be solved using Dynamic Programming. This game lasts n rounds, and you start with 0 points. You have two fair coins, which we will call X and Y. The number n is known to you before the game starts.

In each round, you select one of the two coins, and flip it. If you flip coin X, you gain 1 point if it comes up Heads, and lose 1 point if it comes up Tails. If you flip coin Y, you gain 3 points if it comes up Heads, and lose 3 points if it comes up Tails. After n rounds, if your final score is *positive* (i.e., at least 1 point), then you win the game. Otherwise, you lose the game.

All you care about is winning the game, and there is no extra credit for finishing with a super-high score. In other words, if you finish with 1 point that is no different from finishing with $3n$ points. Similarly, every loss counts the same, whether you end up with 0 points, -1 point, -2 points, or -$3n$ points.

Because you are a Computer Scientist who understands the design and analysis of optimal algorithms, you have figured out the best way to play this game to maximize your probability of winning. Using this optimal strategy, let $p_r(s)$ be the probability that you win the game, provided there are r rounds left to play, and your current score is s. By definition, $p_0(s) = 1$ if $s \geq 1$ and $p_0(s) = 0$ if $s \leq 0$.

(a) Clearly explain why $p_1(s) = 0$ for $s \leq -3$, $p_1(s) = \frac{1}{2}$ for $-2 \leq s \leq 1$, and $p_1(s) = 1$ for $s \geq 2$.

Explain why you must select X if s is 2 or 3, and you must select Y if s is -2 or -1.

(b) For each possible value of s, determine $p_2(s)$. Clearly explain how you determined your probabilities, and why your answers are correct. (Hint: each probability will be one of $\frac{0}{4}$, $\frac{1}{4}$, $\frac{2}{4}$, $\frac{3}{4}$, or $\frac{4}{4}$.)

(c) Find a recurrence relation for $p_r(s)$, which will be of the form $p_r(s) = \max(\frac{??+??}{2}, \frac{??+??}{2})$. Clearly justify why this recurrence relation holds.

From your recurrence relation, explain why the optimal strategy is to pick X when you have certain positive scores (be conservative) and pick Y when you have certain negative scores (be aggressive).

(d) Compute the probability $p_{100}(0)$, which is the probability that you win this game if the game lasts $n = 100$ rounds. Use Dynamic Programming to efficiently compute this probability.

For a bonus mark, determine the limit of $p_n(0)$ as $n \to \infty$, and clearly prove why your answer is correct. (Shockingly, or perhaps not shockingly, the answer is way more than 50%.)

Problem #3 - 20 points

There are over 200 LeetCode problems on Dynamic programming. In this question, you will create a **mini-portfolio** consisting of Three (3) LeetCode problems on Dynamic Programming Algorithms, chosen from the following website.

`https://leetcode.com/tag/dynamic-programming/`

As always, you may code your algorithms in the programming language of your choice.

Here is how your mini-portfolio will be graded.

(i) There will be a total of 15 points for all the problems that your are including in the mini-portfolio: For each of these,

- Provide the problem number, problem title, difficulty level, and the screenshot of you getting your solution accepted by LeetCode
- Source code used - this can be uploaded in Canvas
- Provide a written analysis of the problem investigating the time complexity of your algorithm

Note: Leetcode has three different problem variations: *Easy*, *Medium*, *Hard*

For this problem, the following different problem combinations will get the total points:

- **5 Easy** problems
- **4 Medium** problems
- **3 Hard** problems
- **3 Easy** and **1 Hard** problem
- **4 Easy** and **1 Medium** problem
- **3 medium** and **1 Hard** problem
- **2 medium** and **2 Hard** problem
- **2 Easy** and **1 medium** and **1 Hard** problem

You will get full credit for *any* correct solution accepted by LeetCode, regardless of how well your runtime and memory usage compares with other LeetCode participants.

(ii) (5 marks - **INDIVIDUAL WORK**) For **one** of the problems you are including in your mini-portfolio, explain the various ways you tried to solve this problem, telling us what worked and what did not work. Describe what insights you had as you eventually found a correct solution. Reflect on what you learned from struggling on this problem, and describe how the struggle itself was valuable for you.

The choice of problems is yours, though you may only include problems that took you a minimum of 30 minutes to solve.