Homework 6: Due March 5 (11:59 p.m.)

Instructions

- **Answer each question on a separate page.**
- **Honors questions are optional. They will not count towards your grade in the course. However you are encouraged to submit your solutions to these problems to receive feedback on your attempts. Our estimation of the difficulty level of these problems is expressed through an indicative number of stars** $('*' =$ **easiest**$)$ **to** $('*****' =$ **hardest**$)$**.**
- **You must enter the names of your collaborators or other sources as a response to Question** 0**. Do NOT leave this blank; if you worked on the homework entirely on your own, please write "None" here. Even though collaborations in groups of up to** 3 **people are encouraged, you are required to write your own solution.**

Question 0: List all your collaborators and sources: ($-\infty$ points if left blank)

Question 1: Alternating Coins (1+2+2+5+2=12 points)

Let $n \in \mathbb{N}$ be an *even* number. Consider a row of n coins represented as an array $A = [v_1, \ldots, v_n]$, where v_i is the value of the ith coin. Two players alternate turns, with Player 1 starting. At each turn, a player chooses either the first or the last coin in the row, and takes it. Your goal is to design an algorithm that determines the maximum possible gain Player 1 can definitely win. Assume Player 2 is extremely smart and plays to maximize their payoff. We will use dynamic programming to solve this problem. Define the subproblems $\textsc{MaxGain}(i, j)$ to be the maximum guaranteed payoff for Player 1 if given the subarray $[v_i, \ldots, v_j]$ of *even* length (i.e., $j - i + 1$ must be a positive even number).

1. Suppose the row of coins is
$$A = [1, 1, 1, 2, 2, 2]\ .$$
What payoff can Player 1 definitely win regardless of the opponent's moves? Describe in words the optimal strategy for this particular instance.

2. Fill out the following table for the A defined in the previous part. If entry MaxGain$[i, j] =$ X, then it means that the value is not defined since $j - i + 1$ is not a positive even number.

3. **Base Cases:** State the base cases for $\textsc{MaxGain}(i, j)$ and their values. (Hint: Consider subarrays of length 2.)

4. **Recurrence:** Give and justify the recurrence $\textsc{MaxGain}$ satisfies.

5. **Algorithm:** Describe an algorithm that computes $\textsc{MaxGain}$ and outputs the maximum possible gain of Player 1, regardless of Player 2's moves. Your algorithm should describe in simple words how the $\textsc{MaxGain}$ array is filled. Use pseudocode only if necessary. State and justify the run time of your algorithm as a Θ expression.

$i \backslash j$	1	2	3	4	5	6
1	X	1	X			
2	X	X	1			
3						
4						
5						
6						

Table 1: Fill in the entries for $\textsc{MaxGain}(i, j)$.

Question 2: Coding for Coins (20 points)

Now we will implement an actual code that solves the alternating coins problem from Question 1. Please make sure to:

1. Join the **HackerRank contest (click here)**. You can use HackerRank to check the correctness and efficiency of your algorithm.

2. Submit your code file (for example, .py or .java file) on **Gradescope**. You will be graded based on the correctness and efficiency of the code you submit on **Gradescope** (not HackerRank).

Remark for Python users. We suggest using the bottom-up approach instead of the top-down approach for this problem. If you insist on using top-down, then adding `sys.setrecursionlimit(4096)` might help if you are getting recursion depth errors.

Question 3: Palindromes (2+2+4+2=10 points)

A palindrome is a non-empty string that reads the same forward and backward. Examples of palindromes are “civic”, “racecar”, and “aibohphobia” (fear of palindromes). You are given a string as an array $S[1, \ldots, n]$ of length n. Your goal is to find the length of the longest palindromic subsequence of a string. Note that, by definition, a subsequence can consist of **non-consecutive** elements. For example, for the string “character”, the answer is 5, which corresponds to the palindromic subsequence “carac”.

1. **Subproblems:** Define appropriate subproblems MaxLength in words. You should explain what the entries in MaxLength are intended to store.

2. **Base Cases:** State the base cases for MaxLength and their values.

3. **Recurrence:** State and justify a recurrence for MaxLength.

4. **Algorithm:** Describe an algorithm that computes MaxLength and outputs the *length* of the longest palindromic subsequence. Your algorithm should describe in simple words how the MaxLength array is filled. Use pseudocode only if necessary. State and justify the run time of your algorithm as a Θ expression.

Question 4: Priority Queue (2+1=3 points)

This question is meant to refresh your memory on priority queues (see CLRS, Chapter 6.5). In a priority queue, each element is associated with a *priority value*. Priority queues (or more precisely, max-priority queues) support the following operations. [1] The run-time of each operation is for the heap implementation of priority queues.

- ExtractMax$()$: remove the highest priority (i.e., has the largest priority value) element of the queue and return it. Run-time is $O(\log n)$.
- IncreaseKey(e, k): increases the priority value of the element e to k, which is assumed to be at least as large as e's current priority value. Run-time is $O(\log n)$.
- Insert(e, p): insert the element e with priority value p into the queue. Run-time is $O(\log n)$.

Suppose we have an initially empty priority queue P. Consider the following sequence of operations:

Algorithm 1

```
$P$.Insert$(a, 5)$;
$P$.Insert$(b, 4)$;
$P$.Insert$(c, 8)$;
print($P$.ExtractMax$()$);
$P$.Insert$(d, 6)$;
$P$.Insert$(f, 1)$;
print($P$.ExtractMax$()$);
$P$.IncreaseKey$(f, 3)$;
$P$.IncreaseKey$(b, 9)$;
$P$.Insert$(g, 2)$;
$P$.Insert$(h, 7)$;
print($P$.ExtractMax$()$);
print($P$.ExtractMax$()$);
```

1. What is the output generated by the above sequence of operations?
2. What elements are left in the priority queue after this sequence of operations? Write your answer in the form (e, p) where e is an element and p is its priority value.

Question 5: Honors question (Optional, 0 points)

(**) Here we try to find a *reduction* from the Longest Palindromic Subsequence problem (LPS) (Question 3) to the Longest Common Subsequence (LCS) problem from class. In other words, we want to solve LPS using an algorithm for LCS. Assume that you have an algorithm P that solves LCS (i.e., it finds the length of the longest common subsequence between any two given strings). Show how to use P to solve LPS (i.e., find the length of the longest palindromic subsequence of any given string $S[1, \ldots, n]$). Justify your answer. Minimize the number of times you need to call P. For double credit, try to also output the actual palindromic subsequence (assuming P provides the actual longest common subsequence). (Hint: to appreciate why both parts of this question are not totally obvious, think about the input string "ACBAC"!)

[1] There are also min-priority queues with analogous operations.