CSCI-GA.1170-003/004 Fundamental Algorithms April 2, 2020

Solutions to Problem 1 of Homework 7 ((16+8))

Name: Rahul Zalkikar (rz1567) *Due: 8 pm on Thursday, April 2*

Collaborators: NetID1, NetID2

In the lecture we discussed Longest Common Subsequence and solving it using dynamic programming paradigm. In this question we will explore the idea of Increasing Subsequence. An increasing subsequence of an array $A[1, \ldots, n]$ is given by the indices $i_1 < i_2 < \ldots < i_m$ such that $A[i_{(j-1)}] < A[i_j] \; \forall \; j = 2, 3, \ldots, m$. Note that this increasing subsequence has length m. A possible optimization problem pertaining to this is to return the length of the longest increasing subsequence of an array A.

The goal is as follows: Given an array of distinct numbers $A = (a_1, \ldots, a_n)$, your goal is to find the length of the longest increasing subsequence contained inside A. Consider $A = (10, 11, 12, 1, 8, 2, 7, 3, 4, 4)$. Then the length of the longest increasing subsequence is 4 achieved by subsequence $(1, 2, 3, 4)$. Notice that the sequence $(1, 2, 3, 4, 4)$ is invalid as it is not increasing. Some of the other increasing subsequences are $(10, 11, 12)$, $(1, 2, 7)$.

(a) (4 points) Assume that you have a blackbox algorithm LCS-Solve that takes as inputs two arrays and returns the length of the common subsequence between them in time $\Theta(1)$. Use this and another algorithm discussed in class to construct an $O(n \log n)$ to solve the longest increasing subsequence. You will *not* be designing any new dynamic programming solution specific to this problem. Prove the correctness of your algorithm and the runtime of the algorithm.

Solution:

```
1 LIS-Solve(A, n):
2     B = new array of size n
3     j = 0
4     for i = 1 to n − 1:
5         if A[i] ≠ A[i + 1]:
6             B[j] = A[i]
7             j = j + 1
8     C = HeapSort(B)
9     return LCS-SOLVE(C, A)
```

First we remove the duplicates in A and store them in B. Then we sort B using HeapSort and store it in C. Finally we return the call of LCS-SOLVE(C, A).

C is sorted (in increasing order) and all its elements are distinct, so every subsequence of C is sorted (in increasing order). In addition, any common subsequence between C and A will be sorted (in increasing order) because all subsequences of C are sorted (in increasing order). Thus, when we find the longest common subsequence between C and A it will yield the longest increasing subsequence in A.

Removing duplicates takes $O(n)$ time since we are iterating through the elements in A. We know the runtime of HeapSort is $O(n \log n)$ and the runtime of LCS-SOLVE is $\Theta(1)$. So $T(n) = O(n \log n) + O(n) + \Theta(1) = O(n \log n)$.

□

Recall that the LCS Algorithm discussed in glass runs in time $O(n^2)$ and uses space $O(n^2)$. Therefore, the algorithm from part (a) takes time $O(n^2)$. In the next part we will make it more efficient.

(b) (5 points) Let $D[i]$ be the length of the longest increasing subsequence among the elements $A[1], \ldots, A[i]$, ending at $A[i]$. Write a recursive formulation for $D[i]$ in terms of its subproblems. Prove the correctness of your recursive formula. Further, explain how to use $D[1], \ldots, D[n]$ to solve for longest increasing subsequence problem. (**Hint**: Define $D[0] = 0$, $A[0] = -\infty$.)

Solution:

Let $D[0] = 0$, $A[0] = -\infty$.

$$D[i] = \begin{cases} max_{0 \le j < i} D[j] + 1 & A[j] < A[i] \\ 1 & o.w \end{cases}$$

For each element i we iterate over all subsequences ending at $j \in (1, ..., i-1)$ and check whether $A[j] < A[i]$ (or if the ith element can be added to any subsequence). Then we update $D[i]$ to the length of the maximum of these subsequences, which would be $D[j] + 1$. If there is no subsequence in which $A[i]$ can be added, then $D[i] = 1$ since this will only have the ith element.

Since we check all possible subsequences in the subarray $A[0, ..., i-1]$ to see if $A[i]$ can be added to one, and then take the maximum of these subsequences that end with $A[i]$, we can see that $D[i]$ always finds the length of the longest increasing subsequence ending at $A[i]$.

To solve for longest increasing subsequence we can take $max_{1 \le i \le n} D[i]$.

□

(c) (4 points) Use part (b) to write the pseudocode for a **bottom-up** algorithm using using dynamic programming. What is the runtime of your algorithm?

Solution:

$D = [0] * n$

```
1 LIS-SOLVE(A, n):
2     D[1] = 1
3     for i = 2 to n:
4         D[i] = 1
5         for j = 0 to i:
6             if A[i] > A[j] and D[j] + 1 > D[i]:
7                 D[i] = D[j] + 1
8     return max(D)
```

The runtime is $O(n^2)$

□

(d) (3 points) We often store a helper value to reconstruct the optimal solution for a dynamic programming problem. Let us assume that you store this in an array P of size similar to D. Briefly explain how you would update P by making minor modifications to your algorithm from Part (c). Further, write the pseudocode for an $O(n)$ algorithm PRINT-SOLUTION(P, D, n) to reconstruct an optimal solution.

Solution:

Every time $A[i] > A[j]$ and $D[j] + 1 > D[i]$ for $j = 0, .., i$ I would set the ith value of P to j.

$D = [0] * n$

$P = [-1] * n$

```
1 LIS-SOLVE(A, n):
2     D[1] = 1
3     for i = 2 to n:
4         D[i] = 1
5         for j = 0 to i:
6             if A[i] > A[j] and D[j] + 1 > D[i]:
7                 D[i] = D[j] + 1
8                 P[i] = j
9 return max(D)
```

```
1 PRINT-SOLUTION(P, D, n):
2     maxInd = 1
3     for i = 2 to n:
4         if D[i] > D[maxInd]:
5             maxInd = i
6     c = maxInd
7     while c ≠ −1:
8         print(c)
9         c = P[c]
```

□

(e) (8 points) **(Extra credit:)** Here we will design a more clever algorithm running in time $O(n \log n)$ total. We will maintain two arrays $M[j]$ and $L[i]$, where

- $M[j]$ contains the *smallest possible largest element* a_k of all increasing subsequences of length j seen *so far* (as we scan A). E.g., in our example $A = (10, 11, 12, 1, 8, 2, 7, 3, 4, 4)$ above, $M[3]$ is initially infinity ($i = 0$), then becomes 12 when $i = 3$ is processed, then becomes 7 when $i = 7$, then becomes 3 when $i = 8$, and stays at 3 till the end.

- $L[i]$ contains the pointer to the index k of the *predecessor* of a_i in the longest increasing subsequence *ending* in a_i ($L[i] = 0$ if no such predecessor). E.g., in our example, $L[3] = 2$ (since $a_2 = 10$ is predecessor of $a_3 = 11$ in $10, 11, 12$) and $L[9] = 8$ (since $a_8 = 3$ is predecessor of $a_9 = 4$ in $1, 2, 3, 4$).

In our example $A = (10, 11, 12, 1, 8, 2, 7, 3, 4, 4)$, this is how these arrays look like as we scan over the original array A:

$i = 0$: $M = (\infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty)$, $L = (-, -, -, -, -, -, -, -, -, -)$.

$i = 1$: $M = (10, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty)$, $L = (0, -, -, -, -, -, -, -, -, -)$.

$i = 2$: $M = (10, 11, \infty, \infty, \infty, \infty, \infty, \infty, \infty, \infty)$, $L = (0, 1, -, -, -, -, -, -, -, -)$.

$i = 3$: $M = (10, 11, 12, \infty, \infty, \infty, \infty, \infty, \infty, \infty)$, $L = (0, 1, 2, -, -, -, -, -, -, -)$.

$i = 4$: $M = (1, 11, 12, \infty, \infty, \infty, \infty, \infty, \infty, \infty)$, $L = (0, 1, 2, 0, -, -, -, -, -, -)$.

$i = 5$: $M = (1, 8, 12, \infty, \infty, \infty, \infty, \infty, \infty, \infty)$, $L = (0, 1, 2, 0, 4, -, -, -, -, -)$.

... and so on until at $i = 10$ you get some "useful" values (e.g., $M[4] = 4$, $M[5] = \infty$, and $L[9] = 8$, for example).

First, finish the sample evolution of M and L above which I stopped at $i = 5$.

Then, returning to the general problem, show how you can maintain the two arrays in time $O(\log n)$ per index $1 \leq i \leq n$, and then how this allows you to solve the original problem in total time $O(n \log n)$.

Solution: INSERT YOUR SOLUTION HERE □