CS256: Algorithm Design and Analysis **Spring 2021**

Assignment 5 (due 03/30/2021)

Instructor: Shikha Singh *Solution template:* Overleaf

Note. This assignment is due a day early than usual: it is due on Tuesday March 30.

For full credit when describing a dynamic programming algorithm, you must clearly state the following parts:

(a) **Subproblem definition:** your subproblem must have an optimal substructure.

(b) **Recurrence:** how should the next subproblem be computed using the previous ones? This is the core of your algorithm and its correctness.[1] You must **explain in words** how you arrived at the particular recurrence.

(c) **Base case(s):** you need to start somewhere!

(d) **Final output:** in terms of your subproblem.

(e) **Memoization data structure:** this may often be obvious but should not be skipped.

(f) **Evaluation order:** describes the dependencies between the subproblems.

(g) **Time and space analysis**

For this assignment, the above pieces are sufficient to justify your solution—that is, **you do not need to formally prove correctness via induction**. The first problem is solved to give you a template of how to organize your solutions.

The problems in this assignment are in (approximate) order of difficulty and you are encouraged to attempt them in that order.

[1] A less ideal alternative to a recurrence: clear pseudocode of the iterative DP algorithm.

Problem 1. (Erickson 3.3) Suppose you are given an array $A[1, \ldots, n]$ of numbers, which may be positive, negative, or zero, and which are not necessarily integers.

Describe and analyze an algorithm that finds the largest sum of elements in a contiguous subarray $A[i \ldots j]$. So if the array consists of only positive numbers, then the largest sum of contiguous elements would just be the sum of all the elements. If the array is only made up of negative numbers, then the largest subarray is made up of the smallest such number. (To simplify things, we are not allowing empty subarrays).

As an example consider an array $A = [-6, 12, -7, 0, 14, -7, 5]$, then the contiguous subarray with the largest sum is $[12, -7, 0, 14]$ with the sum of 19. If the subarray is $B = [-4, -1]$, then the contiguous subarray with the largest sum is $[-1]$ with the sum of -1.

Note. This problem has been a standard computer science interview question since at least the mid-1980s and can be solved via a variety of approaches. We will use a dynamic-programming approach for this question.

Solution. This dynamic programming problem is similar to the longest increasing subsequence problem we discussed in class.

- **Subproblem definition:** Let $M(i)$ denote the optimal (largest) subarray sum ending at index i of the array (and including $A[i]$).
- **Recurrence:** Let us think of an element $A[j]$, either $A[j]$ extends a contiguous subarray ending at $A[j-1]$ or it does not and starts its own singleton subarray. Taking the maximum value of the two cases gives us the largest subarray sum ending at index j.

  Thus, we get the following recurrence for $M(j)$:

$$M(j) = \max\{M(j-1) + A[j], A[j]\}$$

- **Base case:** $M(0) = 0$ or $M(1) = A[1]$.
- **Final solution:** given by taking the maximum over all the $M(j)$'s that is, $\max_{1 \le j \le n} M(j)$.
- **Memoization structure:** We can store the values of $M[0, 1, \ldots, n]$ in a linear size array. (We can actually do a lot better, we can just keep $O(1)$ state since we only need $M(j-1)$ while evaluating $M(j)$, not the previous values, and need to maintain a maximum-so-far for the final solution. Just a clever space optimization).
- **Evaluation order:** the evaluation of the dynamic program proceeds left to right, starting from $j = 1$ and going up to $j = n$.
- **Time and space analysis:** To evaluate $M(j)$ takes $O(1)$ time for each j. To compute the final solution, we need to take a maximum over these values, which can be done in $O(n)$ with a linear scan afterwards or done with $O(1)$ time per step of the dynamic programming evaluation. Thus, the overall running time is linear. The space usage is linear if you store $M(j)$'s or constant if you don't remember the $M(j)$ values.

□

Problem 2. (Kleinberg Tardos 6.1) Let $G = (V, E)$ be an undirected graph with n nodes. A subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is simple enough.

Call a graph $G = (V, E)$ a path if its nodes can be written as $v_1, v_2, ..., v_n$, with an edge between v_i and v_{i+1}, for $i \in \{1, 2, \ldots, n-1\}$. With each node v_i, we associate a positive integer weight w_i. The problem we want to solve is the following: Find an independent set in a path G whose total weight is as large as possible.

For example, the maximum weight of an independent set in the path in Figure 1 is 14.

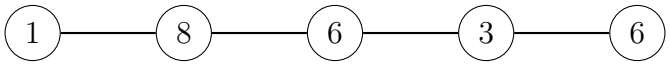

Figure 1: The maximum weight of an independent set is 14 in this example.

(a) Give a counterexample to show that the following "pick the heaviest weight" greedy algorithm does not always work. You should say what the greedy algorithm returns, and what the correct answer is.

- Start with $S = \emptyset$
- While some node remains in G
  - Pick a node v_i of maximum weight and v_i to S
  - Delete v_i and its neighbors from G
- Return S

(b) Give a dynamic-programming algorithm that takes an n-node path G with weights and returns the *value* of the independent set of maximum total weight.

Problem 3. (Kleinberg Tardos 6.3) Let $G = (V, E)$ be a directed acyclic graph where the topological ordering of nodes is $v_1, \ldots, v_n$. A reminder that a topological ordering of a DAG has the following property: *each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form v_i, v_j with $i < j$.* Thus, the source node v_1 has no incoming edges.

We further assume that in graph G each node except v_n has at least one outgoing edge. See Figure 2 for an example.

Given G and its topological ordering, we want to find the *length* of the longest path that begins at v_1 and ends at v_n. (The length of a path is the number of edges in it.)

(a) Show that the following algorithm does not correctly solve this problem, by giving a counterexample. In your example, you should say what the correct answer is and what the above algorithm finds.

- Set $w = v_1$ and $L = 0$
- While there is an edge out of node w:
  - Choose the edge (w, v_j) for which j is as small as possible
  - Set $w = v_j$ and increment L by 1

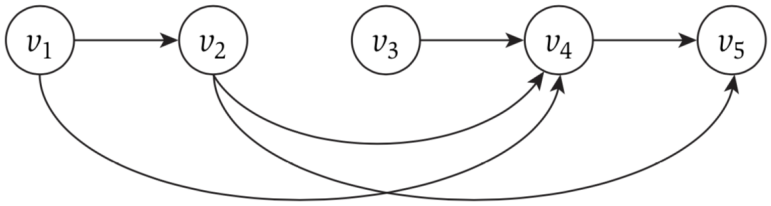

Figure 2: The correct answer for this ordered graph is 3: the longest path from v_1 to v_n uses the three edges $(v_1, v_2), (v_2, v_4)$, and (v_4, v_5).

- Return L as the length of the longest path

(b) Give an efficient dynamic programming algorithm that returns the *length* of the longest path that begins at v_1 and ends at v_n.

Problem 4. (Erickson 3.6) A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of $X$and Yin the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

$$\mathsf{BANANA_{ANANAS}} \qquad \mathsf{BAN_{ANA}ANA_{NAS}} \qquad \mathsf{B_{AN}AN_{A}A_{NA}NA_{S}}$$

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:

$$\mathsf{PRO_{D}G_{Y}R_{NAM}AMMI_{I}N_{C}G} \qquad \mathsf{DY^{PRO}{}_{N}G_{A}R_{M}AMM_{IC}ING}$$

(a) Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

(b) **(Extra credit: 5 pts)** A **smooth** shuffle of X and Y is a shuffle of X and Y that never uses more than two consecutive symbols of either string. For example,

- $\mathsf{PR_{D}O_{Y}G_{NA}R_{A}M_{M}MI_{I}C_{NG}}$ is a smooth shuffle of the strings DYNAMIC and PROGRAMMING.
- $\mathsf{DY_{PR}N_{OGR}A_{A}M_{MM}IC_{ING}}$ is a shuffle of DYNAMIC and PROGRAMMING, but it is not a smooth shuffle (because of the substrings OGR and ING).
- $\mathsf{XX_{X}X_{X}X_{XX}XX_{XX}XX_{X}X_{X}XX}$ is a smooth shuffle of the strings XXXXXXX and XXXXXXXXXXX.
- There is no smooth shuffle of the strings XXXX and XXXXXXXXXXXX.

Describe and analyze an algorithm to decide, given three strings X, Y,and Z, whether Z is a smooth shuffle of X and Y.

Hint: What do you need to change in order to build up a smooth shuffle rather than a normal shuffle? What do you need to keep track of to ensure that you can make this distinction?