CSCI-GA.1170-001/003 Fundamental Algorithms October 31, 2021

Problem Set 7

Lecturer: Yevgeniy Dodis *Due: 8 am on Thursday, November 4*

Problem 7-1 (Maximize Your Scores) (26+11 points)

You are a student of a course "Not-So-Fundamental Algorithms". You have been diligently submitting the solutions to various homework questions and have been assigned scores for each of them. Let us assume that there are n such questions and you are given the scores as an array $A[1 \dots n]$. These scores can be positive or negative.

You are being allowed to pick the homework questions you want to count towards your homework component of the final grade. Clearly, your goal is to maximize the sum of the scores you pick. However, there are some additional conditions imposed on how you pick this subset.

(a) (1 point) Warm-up Question: You are not told any prior information about the values in the array A. You are just asked to pick any or none of the scores. You have to provide the indices $i_1, i_2, \dots i_k$ and the final score would be $\sum_{j=1}^{k} A[i_j]$. You may also choose to pick none of the scores. (When does this happen?) Give me a $\Theta(n)$ solution to return the set of indices so that you maximize the final score.

For parts (b)-(e), you are constrained to pick only continuous scores, i.e pick a subarray from the array A. You may choose to pick none of the elements also. You have to provide two indices i, j and the final score will be $\sum_{k=i}^{j} A[k]$.

(b) (1 point) You are told that $\forall i,\ A[i] \geq 0$. What will your strategy be? Give me a $\Theta(1)$ solution to return the two indices i, j. Justify your strategy briefly.

(c) (2 points) You have no information about the values in the array A. Fill in the blanks to complete the following iterative algorithm that returns the maximum possible score. It is easy to see that this algorithm will have a run-time of $O(n^2)$.

```
1 GetMVCS($A, n$)
2     $maxsum = 0$
3     for $i = 1$ to $n$
4         $current =$ ...............
5         for $j = i$ to $n$
6             .....................
7             if ............ then .....................
8     return $maxsum$
```

(d) (9 points) Consider the problem as defined in Part (c). Construct an $O(n)$ time and $O(n)$ space algorithm using dynamic programming. Justify the runtime of your algorithm. Briefly argue the correctness of your algorithm. The algorithm will utilize a one-dimensional array Best of length n.

These are the asks for the question:

- (1 point) Begin by clearly defining BEST[n]
- (1 point) Give base case(s) for BEST.
- (2 point) Give and justify a recursive formulation for BEST[n] in terms of the subproblems. (**Hint**: There are only two possible moves for you, at a given point. Clearly, you need to do the best you can do out of these.)
- (3 point) Write a **bottom-up iterative** algorithm COMPUTE-BEST(A, n) that computes BEST[n]. You may assume that BEST is a global array initialized to $-\infty$ (for a maximization problem this is typical).
- (1 point) Given array Best, show (in pseudocode) how to find the solution to the original problem of finding the maximum sum obtainable when you pick scores such that they form a subarray.
- (1 point) State and justify the running time of your algorithm.

(e) (3 points) Typically, one uses a helper value to help reconstruct the optimal solution for a dynamic programming problem. However, for this question, one does not need to do that. Indeed, it is not hard to modify the problem from Part (d) to also print the optimal solution. For this question, you will construct an algorithm PRINT-SOLUTION(BEST) that takes as input the correctly computed array BEST from before and prints the solution, i.e, give the starting and ending indices of the contiguous subarray that yields the maximum value. This will have to be a $O(n)$ algorithm.

(f) (10 points) For this question, you are constrained to pick your scores in such a way that you *you cannot pick two adjacent scores* i.e, if index i is picked, indices $i+1$ and $i-1$ cannot be picked. *You need to pick at least one score.* Construct an $O(n)$ time and $O(n)$ space algorithm using dynamic programming. The algorithm will utilize a one-dimensional array BEST of length n.

These are the asks for the question:

- (1 point) Begin by clearly defining BEST[n]
- (1 point) Give base case(s) for BEST.
- (2 point) Give and justify a recursive formulation for BEST[n] in terms of the subproblems. (**Hint**: There are only two possible moves for you, at a given point. Clearly, you need to do the best you can do out of these.)
- (3 point) Write a **top-down recursive** algorithm COMPUTE-BEST(A, n) that computes BEST[n]. Do not forget to state your invocation call. You may assume that BEST is a global array initialized to $-\infty$ (for a maximization problem this is typical).
- (1 point) State the invocation call(s) that will completely fill the table BEST.
- (1 point) Given array BEST, show (in pseudocode) how to find the solution to the original problem of finding the maximum sum obtainable when you pick scores such that no two adjacent scores can be picked.
- (1 point) State and justify the running time of your algorithm.

In the previous problems, we looked at subproblems that are linear, i.e., grow in one-direction and this is typically defined by a one-dimensional array. In this problem, we will throw in some additional constraints which will make it a two-dimensional problem. This can be considered as a dimensional lift, i.e., increasing the dimensions of the subproblem.

(g) (**Extra Credit**)(11 points) You are given an integer k and are asked to pick $i_1, \ldots, i_k$ with your new sum $\sum_{j=1}^{k} A[i_j]$ being the homework component of your final grade. Critically $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. It is evident that you wish to maximize the sum. Construct an $O(nk)$ time and space algorithm using dynamic programming to solve this problem. You will use an $n \times k$ matrix BEST to store your intermediate answrs.

These are the asks for the question:

- (1 point) Begin by clearly defining BEST$[i, j]$
- (2 point) Give base case(s) for BEST.
- (2 point) Give and justify a recursive formulation for BEST$[i, j]$ in terms of the subproblems.
- (3 point) Write a **top-down recursive** algorithm COMPUTE-BEST(A, i, j) that computes BEST$[i, j]$. Do not forget to state your invocation call. You may assume that BEST is a global array initialized to $-\infty$ (for a maximization problem this is typical).
- (1 point) State the invocation call(s) that will completely fill the table BEST.
- (1 point) Given array BEST, show (in pseudocode) how to find the solution to the original problem of finding that maximum sum obtainable when you have to pick k scores.
- (1 point) State and justify the running time of your algorithm.

Problem 7-2 (Spacing Out LCS) 19+11 points

Recall the LCS problem and the approach we took to solving it. We let X_i and Y_j be the prefix, i.e., $\langle x_1, \ldots, x_i \rangle$ and $\langle y_1, \ldots, y_j \rangle$ respectively. We then let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X_m and Y_m and proved the following:

1. If $x_m = y_m$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
2. If $x_m \neq y_n$, then $z_k \neq x_m \implies Z$ is an LCS of X_{m-1} and Y_n
3. If $x_m \neq y_n$, then $z_k \neq y_n \implies Z$ is an LCS of X_m and Y_{n-1}

(a) (3 points) Recall that, in an effort to recover the solution, our LCS-LENGTH algorithm used an additional matrix b of size $m+1 \times n+1$ to record our decisions. This was later used to recover a optimal solution. However, we can conserve space, and avoid using the additional matrix b. Present a recursive pseudocode PRINT-LCS-INDEX(c, X, Y, i, j) where c is the correctly computed $m + 1 \times n + 1$ matrix, as defined in LCS-LENGTH algorithm. Your algorithm will print the actual LCS of X, Y. Briefly argue the correctness and running time of your algorithm.

(b) (3 points) With the help of the previous part, we can certainly cut down on the additional matrix b. However, the asymptotic space complexity is still $O(mn)$. In this problem, you will reduce the space complexity from $O(mn)$ to $O(n)$.

To receive full credit on this problem, you can simply complete the following pseudocode and fill in the blanks where blanks are denoted by .... Note that the algorithm returns one of the two arrays, in entirety.

```
Version: Two arrays
LCS-LENGTH($X, Y, m, n$)
    let $c[0][0 \dots n]$ and $c[1][1 \dots n]$ be the two arrays.
    $c[0][0] = c[1][0] = \dots\dots\dots$
    **for** $i = 1$ **to** $m$
        **for** $j = 1$ **to** $n$
            **if** $X[i] = Y[j]$
                $c[\dots\dots\dots][\dots\dots\dots] = c[\dots\dots\dots][\dots\dots\dots] + 1$
            **else** $c[\dots\dots\dots][\dots\dots\dots] = \max(c[\dots\dots\dots][\dots\dots\dots], c[\dots\dots\dots][\dots\dots\dots])$
    **return** $c[m \mod 2]$
```

(c) (**Extra Credit**)(3 points) We can further optimize the above algorithm to only use one additional array, rather than two. Complete the pseudocode below LCS-LENGTH(X, Y, m, n) that fills up the array $c[0, \dots, n]$ and returns the final array c such that $c[j]$ is the length of the LCS of X_m and Y_j.

```
LCS-LENGTH($X, Y, m, n$)
    let $c[0 \dots n]$ be array
    $c[0] = \dots\dots\dots$
    **for** $i = 1$ **to** $m$
        $prev = \dots\dots\dots$
        $temp = \dots\dots\dots$
        **for** $j = 1$ **to** $n$
            $\dots\dots\dots = \dots\dots\dots$
            **if** $X[i] = Y[j]$
                $c[\dots\dots\dots] = \dots\dots\dots + 1$
            **else** $c[\dots\dots\dots] = \max(c[\dots\dots\dots], c[\dots\dots\dots])$
            $\dots\dots\dots = \dots\dots\dots$
    **return** $c$
```

While we have improved the space complexity to $O(n)$, we have lost the ability to recover the actual subsequence. For the remainder of this problem, we will look at how to recover the solution. In the original formulation, we defined the prefixes of of a string. Specifically, we defined X_i as the prefix of X of length i, i.e., $X_i = \langle x_1, \dots, x_i \rangle$ and similarly, $Y_i = \langle y_1, \dots, y_i \rangle, Z_i = \langle z_1, \dots, z_i \rangle$. where $Z = \langle z_1, \dots, z_k \rangle$ is an LCS of $X = \langle x_1, \dots, x_m \rangle, Y = \langle y_1, \dots, y_n \rangle$.

Similarly, one could define X^i as the suffix of X of length i, i.e., $X^i = \langle x_{m-i+1}, \dots, x_m \rangle$ and similarly of Y and Z. Now, we have that the concatenation of X_i and X^{m-i} yields X. This is easy to verify and we leave it to you as an exercise.

(d) (7 points) These are the asks for the question:

- (2 points) In the lecture, we proved a theorem for the correctness of our algorithm that used prefixes. *State* the corresponding theorem using suffixes. You do not need to prove the theorem.
- (2 points) Use this stated theorem to derive a recursive formula $c[i, j]$ where $c[i, j]$ is the length of LCS of X^{m-i}, Y^{n-j}.
- (3 points) LCS-LENGTH(X, Y, m, n) is reproduced in an abridged fashion from the textbook. Make modifications to exactly 7 lines of this algorithm to produce the new algorithm LCS-LENGTH-SUFFIX which will complete the table c according to the new definition where $c[i, j]$ is the length of LCS of X^{m-i}, Y^{n-j}.

  Reference: Simplified LCS-Length from Textbook

```
LCS-LENGTH($X, Y, m, n$)
1    let $c[0 \dots m, 0 \dots n]$ be new table
2    **for** $i = 1$ **to** $m$
3        $c[i, 0] = 0$
4    **for** $j = 0$ **to** $n$
5        $c[0, j] = 0$
6    **for** $i = 1$ **to** $m$
7        **for** $j = 1$ **to** $n$
8            **if** $x_i == y_j$
9                $c[i, j] = c[i-1, j-1] + 1$
10           **else**
11               $c[i, j] = \max(c[i-1, j], c[i, j-1])$
12   **return** $c$
```

(e) (2 points) Denote by $L(X, Y)$ the length of the LCS of the strings X, Y. Then prove that, there exists $0 \leq j^* \leq n$ such that:

$$L(X, Y) = L(X_{m/2}, Y_{j^*}) + L(X^{m/2}, Y^{n-j^*})$$

where the variables inherit the meanings defined before. For simplicity, you may assume that m is a power of 2.

(f) (4 points) Let us assume that there is an algorithm LCS-LENGTH-PREFIX(X, Y, m, n) that computes $c_p[0, \dots, n]$ such that $c_p[j]$ is the length of LCS of X, Y_j. This is basically the algorithm seen in parts (b), (c). One can define the suffix counterpart LCS-LENGTH-SUFFIX(X, Y, m, n) that computes $c_s[0, \dots, n]$ such that $c_s[j]$ is the length of LCS of X, Y^{n-j}. In this part, present an algorithm that returns j^* and $L(X, Y)$ such that:

$$L(X, Y) = L(X_{m/2}, Y_{j^*}) + L(X^{m/2}, Y^{n-j^*})$$

What is the running time of your algorithm and its space complexity? Justify its correctness. Your algorithm will invoke the algorithms defined in the problem. (**Hint**: The question is how to find j^* exactly. One simple solution is to try every j and find the best j.)

(g) (**Extra Credit**)(8 points) Call your algorithm from part (f) as FIND-J(X, Y, m, n). Present a recursive algorithm based on the divide-and-conquer paradigm that invokes FIND-J and returns the actual LCS between X and Y. Then, let $T(m, n)$ be the worst-case running time of your algorithm to find the LCS of X and Y. Derive a recurrence relation for $T(m, n)$ and prove by induction that $T(m, n) = O(mn)$. Therefore, you have created an algorithm that runs in time $O(mn)$ and uses space $O(n)$ (ignoring the recursion stack). Briefly justify the correctness of your algorithm.

Problem 7-3 (Counting BSTs) 12 points

Let us recall the definition of *binary search tree property*: Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

(a) (2 points) Consider the following algorithm: For each node, check if its left child is lesser than its key and its right child is greater than its key. If yes, pass the recursion onto the left and right child. This pseudocode is as follows:

```
1 IsBST-A($x$)
2     if ($x$ = NIL)
3         return true
4     if ($x.left != $ nil  &  $x.left.key > x.key$)
5         return false
6     if ($x.right != $ nil  &  $x.right.key < x.key$)
7         return false
8     if (IsBST-A($x.left$)=false  Or IsBST-A($x.right$)=false )
9         return false
10    return true
```

Show that this algorithm is wrong by constructing an example of a binary tree which is not a binary search tree but for which the algorithm returns true.

In the remainder of this we will try to count the number of possible BSTs with n distinct nodes. For simplicity you may assume that the possible labels are $[1 \ldots n]$.

For example, when $n = 1$, you know that the only possibility is just a root which is also the leaf. However, for $n = 2$, we have two choices for the root: We can make 2 the root and 1 its left child or make 1 the root and 2 its right child.

(b) (2 points) Illustrate all the possible BSTs for $n = 3$.

(c) (4 points) Let F_n denote the number of possible BSTs with n elements. How many trees are possible with i as the root? Use this to write a recursive formulation for computing F_n. (**Hint**: If i is the root, what are the possible elements that can occur in the left child and what about the ones on the right child?)

(d) (4 points) Use the above formulation to write a *recursive* algorithm (top-down) based on Dynamic Programming to compute the value of F_n. You may find it useful to declare a global array $M[1 \ldots n]$. Assume that the array can store the large values of F.