CS 161 **Problem Set 6**
Spring 2017 **Due:** May 26, 2017, 3pm

Please answer each of the following problems. Refer to the course webpage for the **collaboration policy,** as well as for **helpful advice** for how to write up your solutions.

Note: For all problems, if you include pseudocode in your solution, please also include a brief English description of what the pseudocode does.

1. [**Why not just do divide-and-conquer?**] **(7 points)** Recall the Longest-Common-Subsequence problem from class: given sequences X and Y of length m and n respectively, find the length of a longest common subsequence (LCS) of X and Y. In class we saw a bottom-up dynamic programming approach; in this program will explore a top-down approach.

   For parts (a) and (b), refer to the pseudo-code below, which is a divide-and-conquer approach that returns the length of the longest common subsequence of two sequences X and Y:

```
LCS(X,Y):
   i=length(X)
   j=length(Y)
   if i==0 or j==0:
      return 0
   else:
      if X[i]==Y[j]:
          return 1 + LCS(X[1..i-1],Y[1..j-1])
      else:
          return max(LCS(X[1..i],Y[1..j-1]),LCS(X[1..i-1],Y[1..j]))
```

   (a) (3 points) Suppose that X and Y are sequences over an alphabet of size at least 2. Prove that the worst-case runtime of the above recursive implementation of the algorithm is $\Omega(2^{min(m,n)})$, where m and n are the lengths of the inputs X and Y.

   [We are expecting: A short but formal proof. Make sure you specify what the worst-case input that you consider is.]

   (b) (4 points) In class, we saw a bottom-up dynamic program for Longest-Common-Subsequence. Give pseudocode for a top-down version of this algorithm. That is, modify the divide-and-conquer algorithm above to keep a table of solutions to avoid repeated work. Your dynamic programming solution should have running time $O(mn)$, and your algorithm should have the same recursive structure as `LCS(X,Y)` above.

   [We are expecting: the pseudo-code and a short informal justification of why it runs in time $O(mn)$.]

2. [**Longest path.**] **(7 points)** We saw in class that finding longest simple paths in general graphs doesn't seem to be amenable to dynamic programming. However, in a DAG, it turns out it is. In this question, you'll design a dynamic programming algorithm that finds longest simple paths in a directed acyclic graph.

   For the following, let $G = (V, E)$ be a weighted directed **acyclic** graph, so that the weight on an edge $(u, v) \in E$ is $w(u, v)$. Further suppose that $s, t \in V$, and that **all vertices in V are reachable from s.** Recall that a path is *simple* if it has no cycles. We say that a *longest simple path* from s to t is a path from s to t with maximal cost.

(a) (2 points) For a vertex $x \in V$, let $L(x)$ denote the cost of a longest simple path from s to x. Show that for all $x \neq s$,

$$L(x) = \max_{u:(u,x)\in E} (L(u) + w(u,x)).$$

[We are expecting: a formal proof. Make sure that you explicitly use the fact that G is acyclic; otherwise the statement is false!]

(b) (1 point) Suppose that $v_1, \ldots, v_n$ is a topological sorting of the vertices. That is, if $(v_i, v_j) \in E$, then $i < j$. Show that

i. $s = v_1$, and that

ii.

$$L(v_j) = \max_{i:i<j \text{ and } (v_i,v_j)\in E} (L(v_i) + w(v_i, v_j)).$$

[We are expecting: a short but formal proof of these two statements. (No more than a sentence or two for each).]

(c) (4 points) Design a dynamic programming algorithm to find the *length* of the longest directed path from s to t, which runs in time $O(n + m)$, where $n = |V|$ and $m = |E|$.

Notes: If it is helpful, you may assume that you have access to a method `topologicalSort(G)` which, in time $O(n + m)$, takes a graph G with n vertices and m edges, and returns an array `topoSort` so that `topoSort[i]` contains a pointer to the i'th vertex in a topologically sorted order.[1] As usual, you may also assume that there are methods `getIncomingNeighbors(v)` and `getOutgoingNeighbors(v)` which takes a vertex v and in time $O(1)$ returns a pointer to the head of a list incoming or outgoing neighbors of v, respectively.

[We are expecting: pseudocode, with a short informal explanation of why it is correct and why the running time is $O(n + m)$.]

3. **[Dynamic programming for making change.] (9 points)** Coins in the United States are minted with denominations of $\{1, 5, 10, 25, 50\}$ cents. There are several ways to make change of any given amount of money: for example, to make change of 11 cents, we could use a dime and a penny, two nickels and a penny, or 11 pennies.

In the mythical country of CS161-land, coins are minted with denominations of $\{d_1, ..., d_t\}$ CS161-cents, where $d_1, \ldots, d_t$ are positive integers. In this problem, we will design dynamic programming algorithms to understand how to make change of n CS161-cents.

(a) (1 point) Suppose that the denominations are $\{1, 6, 10\}$. What is the way to make change of 18 CS161-cents that involves the fewest coins? How many different ways are there to make change of 18 CS161-cents?

[We are expecting: just the answers to these questions]

(b) (3 points) Suppose that $M[k]$ is the minimum number of coins needed to represent k CS161-cents. Give an efficient dynamic programming algorithm which maintains a table M, and which takes an integer n and a set D of denominations as input, and outputs a **way to make change** of n CS161-cents using the minimal number of coins. (Not just the number of coins needed).

[We are expecting: pseudocode, and an informal justification that it is correct.]

(c) (1 point) Analyze the running time of your algorithm in part (b).

[We are expecting: The best big-oh running time you can guarantee, and a short justification.]

[1]Not required: how would you implement such an algorithm?

(d) (3 points) Suppose that, for $j \leq t$, $N[j,k]$ is number of ways to make change of k CS161-cents using only the denominations $\{d_1, \ldots, d_j\}$. Give an efficient dynamic programming algorithm which maintains a table N, and which takes an integer n and a set D of denominations as input, and outputs the number of **distinct** ways to make change of n CS161-cents using the denominations in D.

Assume that coins of the same denomination are indistinguishable. For example, if $D = \{1, 6, 10\}$ and $n = 6$, then your algorithm should return 2, because there are two ways to make change of 6: 6 itself and $1, 1, 1, 1, 1, 1$.

[We are expecting: pseudocode, and an informal justification that it is correct.]

(e) (1 point) Analyze the running time of your algorithm in part (e).

[We are expecting: The best big-oh running time you can guarantee, and a short justification.]

4. **[Fish fish eat eat fish.] (6 points)** Plucky the Pedantic Penguin enjoys fish, and he has discovered that on some days the fish supply is better in Lake A and some days the fish supply is better in Lake B. He has access to two tables A and B, where $A[i]$ is the number of fish he can catch in Lake A on day i, and $B[i]$ is the number of fish he can catch in Lake B on day i, for $i = 1, \ldots, n$. Assume that $A[i]$ and $B[i]$ are positive integers for all $i = 1, \ldots, n$.

Plucky's goal is to have as many fish as possible at the end of day n. If he happens to be at Lake A on day i and wants to be at Lake B on day $i + 1$, he may pay L fish to a polar bear who can take him from Lake A to Lake B overnight; the same is true if he wants to go from Lake B back to Lake A. Here $L > 0$ is a positive integer. Assume that before day 1 begins, Plucky is at Lake A, and he has zero fish.

Plucky will save all the fish he gets until the end of day n (at which point he will feast), but the polar bear does not accept credit. So Plucky must have the fish on hand in order to pay the polar bear; he must pay *before* he travels.

In this question, you will design an $O(n)$-time dynamic programming algorithm that finds the maximum number of fish that Plucky will have on day n. Do this by answering the two parts below.

(a) (3 points) What sub-problems will you use in your dynamic programming algorithm? What is the recursive relationship which is satisfied between a problem and the sub-problems? What is the base case for this recursive relationship? Justify your answer.

[We are expecting: a formal definition of your sub-problems, as well as a recursive relationship that they satisfy. We are also expecting an informal justification that your recursive relationship is correct given your definitions.]

(b) (3 points) Write pseudocode for a dynamic programming algorithm that takes as input A, B, L and n, and in time $O(n)$ returns the maximum number of fish that Plucky can eat on day n.

[We are expecting: the pseudocode, a brief explanation (one or two sentences) about why it works, and justification (also one or two sentences) that it runs in time $O(n)$.]

5. **[There is no problem 5.] (0 points)** This problem set is long enough already! But if you finish all of the above and still want more practice with dynamic programming, try the problems in Chapter 15 of CLRS.