CS 344 - Sections 5,6,7,8 - Fall 2022
Homework 4
100 points total (no extra credit on this one)

Very Important: what to write for DP problems

For all the dynamic programming problems in this class, your solution must contain ALL of the following 5 steps, in the exact order below. *Don't forget the include the pseudocode (step 5):* that is the most important part. You can see examples of the 5 steps in my lecture notes for MinBills and for SubsetSum

1. What the values in your table correspond to. So e.g. for longest increasing subsequence (from class) I would write something like: $T[i]$ is the length of the longest increasing sequence ending at $A[i]$
2. The DP-relation that computes one table entry in terms of other table entires. So for longest increasing subsequence I would write something like: $T[i] = \max_j T[j] + 1$, where the maximum is taken over all $j < i$ with $A[j] < A[i]$.
3. How do you initialize the table: so for longest increasing subsequence I would write $T[0] = 1$.
4. Which entry of the table you return at the end of the algorithm. So for example for longest increasing subsequence I would write: return FindMax(T)
5. MOST IMPORTANT STEP: Pseudocode for the final algorithm.

1 Problem 1: Special Numbers – 25 points

We define a number to be *special* if it can be written as $a \cdot 113 + b \cdot 155 + c \cdot 189$, for some *non-negative* integers a, b, and c. For example

- 491 is special because it can be written as $1 \cdot 113 + 0 \cdot 155 + 2 \cdot 189$.
- 1069 is special because $1069 = 2 \cdot 113 + 3 \cdot 155 + 2 \cdot 189$

- 452 is special because it can be written as $4 \cdot 113 + 0 \cdot 155 + 0 \cdot 189$.
- 223 is NOT special. (Note that $223 = (-1) \cdot 113 + 0 \cdot 155 + 2 \cdot 189$, but this does not count because -1 is a negative number.)

The goal of this problem is to write a DP algorithm Special(n):

- INPUT: a positive integer n
- OUTPUT: return numbers a, b, c such that $n = a \cdot 113 + b \cdot 155 + c \cdot 189$ or "no solution" if such numbers do not exist.

The Problem: Write an algorithm Special(n) that runs in $O(n)$ time.

NOTE: you don't need any complicated arithmetic operations for this problem. You don't even need multiplication and division. Try to solve it using dynamic programming.

WAHT TO WRITE: write the five DP steps, including pseudocode for the final algorithm. Note that your algorithm has to output the actual numbers a, b, c.

GRADING NOTE: If you write an algorithm that returns "TRUE" for special numbers and "False" for non-special ones, but your algorithm cannot return the actual numbers a, b, c, you will get 20/25 points.

2 Problem 2 (30 points total)

Sally lives at the bottom of long stairs with n steps. Sally is quite nimble, so she can go up the stairs one step at a time, but she can also skip steps. Sally goes up these stairs every day so she thinks about them a lot. She wonders *how many* different ways she has of going up the stairs.

Formally, Sally goes up the stairs in a sequence of moves. In a single move she can go up one step or two steps or three steps, all the way up to k steps, for some variable $k \leq n$. Write a dynamic program NumCombos(n,k) that determines how many possible sequences of moves there are for Sally to get to step n. She starts at step 0.

For example, if $n = 5$ and $k = 2$ then in each move Sally can go either one step or two steps. In this case, NumCombos(n,k) = 8 because there are eight ways to traverse 5 steps: 11111,1112,1121,1211,122,2111,212,221.

Note that NumCombos(n,k) only outputs the *number of ways*: you do NOT have to enumerate all possible ways.

Part 1 (15 points) Write a dynamic program to solve NumCombos(n,k) in $O(nk)$ time. Make sure to include all elements of a dynamic program detailed at the top of the HW.

EXTRA CREDIT: if your algorithm runs in $O(n)$ time, you will receive 5 extra points. So if you give a correct answer with $O(nk)$ running time you will get 15/15 points. If you give a correct answer with $O(n)$ runtime, you will get 20/15 points. *You may only give one answer.*

Part 2 (15 points) Now say that Sally wants to make ≤ 100 moves total. Write a dynamic program NumCombosH(n,k) that determines the total possible number of ways Sally can climb the stairs with this additional restriction. The running time should be $O(nk)$. Again make sure to include all elements of a dynamic program detailed above.

Note that in this case, you might actually have NumCombosH(n,k) = 0. For example, if $n = 1000$ and $k = 9$ then there is just no way Sally is getting to the top in 100 moves.

For this problem, there is no extra credit for a faster algorithm.

NOTE FOR BOTH PARTS: even though the numbers can get very large for this problem, you can assume all arithmetic operations take $O(1)$ time.

3 Problem 3: Suitcase packing (20 points)

You have a set of objects $S = \{s_1, s_2, ...s_n\}$. Each object s_i has a positive integer weight w_i and a positive integer value v_i. You have a suitcase that can carry a total weight of at most W. A valid suitcase is a set of objects $T \subseteq S$ for which $\sum_{s_i \in T} w_i \leq W$. The value of a suitcase T is then $\sum_{s_i \in T} v_i$. Your goal is to find the maximum possible value of a valid suitcase.

For example, say that your max allowed weight is W = 17, and you have 6 objects:

- Object s_1 has $w_1 = 14$ and $v_1 = 20$
- Object s_2 has $w_2 = 15$ and $v_2 = 14$
- Object s_3 has $w_3 = 7$ and $v_3 = 12$
- Object s_4 has $w_4 = 5$ and $v_4 = 7$
- Object s_5 has $w_5 = 4$ and $v_5 = 5$

- Object s_6 has $w_6 = 2$ and $v_6 = 2$

Then, the maximum value you can get is $12 + 7 + 5 = 24$: you get this by picking objects s_3, s_4, and s_5. note that the total weight of this suitcase is $7 + 5 + 4 = 16$, which is less than the maximum allowed 17, so it is indeed a valid suitcase.

The Problem: The goal is to write an algorithm for this problem that runs in $O(nW)$ time. *But I will write most of the algotithm for you!*. I will leave two key lines blank and your goal is to fill in these lines.

The idea is of course to use dynamic programming. The algorithm will use a two dimensional table $T[0...n][0...W]$. I will tell you the final goal of what we want to put in each $T[i][j]$ (step 1 in the 5-step process above): we want $T[i][j]$ to contain the maximum possible value of a suitcase that is allowed to use any subset of $\{s_1, \ldots, s_i\}$ and that has max weight j.

The key step is then to figure out the DP-relation, i.e. how we can quickly compute $T[i][j]$ in terms of earlier table entries. Those are the lines I am leaving blank in the pseudocode below, and the lines you will have to fill in.

Pseudocode:

- Initialize $T[0...n][0...W]$ with 0 in each entry
  - For $i = 1$ to n
    - For $j = 1$ to W
      1. If $w_i > j$
         - ANSWER 1 HERE
      2. Else
         - ANSWER 2 HERE
- Return $T[n][W]$

The Problem: Fill in the two answers. Note that both Answer 1 and Answer 2 should each be *at most two lines of pseudocode.* You in fact only need a single line per answer, but two lines is fine; more than that is not allowed.

NOTE: *Don't write anything except the answer lines.* So your final answer should take the form:

- Answer 1: your answer here. (Two lines of pseudocode max.)
- Answer Line 2: your answer here. (Two lines of pseudocode max.)

NOTE: recall that the total running time of the algorithm has to be $O(nW)$.

4 Problem 4: longest switching sub-sequence (25 points)

Given an array A, recall that we call $A[i_1], A[i_2], ..., A[i_k]$ a subsequence if $i_1 < i_2 < ... < i_k$.

Definition: We say that subsequence $A[i_1], ..., A[i_k]$ is a *switching* subsequence if the following is true:

- $A[i_1] < A[i_2]$
- $A[i_2] > A[i_3]$
- $A[i_3] < A[i_4]$
- $A[i_4] > A[i_5]$
- $A[i_5] < A[i_6]$
- $A[i_6] > A[i_7]$
- and so on...

Note that the first element has to be smaller than the second, and that after that the signs of the inequalities are switching direction. Another way to think about this is that in class we defined an increasing subsequence, where each element is bigger than the previous one. In a switching subsequence, on the other hand, every other element of the subsequence – $A[i_2], A[i_4], A[i_6], ...$ must be larger than *both* the element before it *and* the element after it.

example: Consider the array

$$A = [1000, 0, 100, 1, 20, 200, 160, 2, 3, 90, 150, 4, 5, 6, 103, 7, 8, 9, 50, 10, 11, 12, 13]$$

- There are several longest switching subsequences in this array. Here is one of them: $0, 100, 20, 200, 2, 90, 5, 103, 7, 50, 11, 13$
- Note that the following is NOT a switching subsequence, because the first element is larger than the second, which is not allowed: $1000, 0, 100, 20, 200, 2, 90, 5, 103, 7, 50, 11, 13$

The problem: Given an array A, write pseudocode for a DP algorithm that finds the switching subsequence $A[i_1], ..., A[i_k]$ of maximum length. (That is, you want the number of indices k to be as large as possible.) The running time of you algorithm should be $O(n^2)$ time. If there are multiple switching subsequences that attain this maximum length, you only have to return one of them. Note that a single element $A[i_i]$ always qualifies as a switching subsequence, so there always exists a switching subsequence of length at least 1.

GRADING NOTE: you must return not just the length of the longest switching subsequence, but the subsequence itself. If your algorithm correctly returns the length in $O(n^2)$ time, but not the subsequence itself, you will get 20/25 points.

HINT: I strongly recommend first figuring out the algorithm to return just the length (not the subsequence itself). Once you feel comfortable with this, use a last choice array to also return the subsequence as well.

HINT: This problem is quite similar to longest increasing subsequence from class, and I really recommend looking at that as a starting point. The main difference is that in longest increasing subsequence you stored all the relevant information in a table T[0...n-1]. For this problem you will need a add an extra dimension, but it will be a small dimension. In particular, in the solution I have in mind, you have a table of T[0...n-1][0...1]; that is, the table has n rows and 2 columns. I will leave it up to you to figure what kind of information you should put it $T[i][0]$ and $T[i][1]$. (There are actually several equally good ways to do this problem, though all the ones I have in mind use a table with n rows and 2 columns).