Homework #4 Solutions

CS 344: Design and Analysis of Computer Algorithms (Fall 2022)

Problem 1

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.

(1) Table values: If i is special, $T[i]$ is a triple $(T[i]_1, T[i]_2, T[i]_3)$ satisfying

$$i = T[i]_1 \cdot 113 + T[i]_2 \cdot 155 + T[i]_3 \cdot 189$$

where $T[i]_{1,2,3} \geq 0$. $T[i]$ is NIL otherwise. T is indexed by $[-189..n]$.

(2)

$$T[i] = \begin{cases} T[i-113] + (1,0,0) & \text{if } T[i-113] \neq \text{NIL} \\ T[i-155] + (0,1,0) & \text{else if } T[i-155] \neq \text{NIL} \\ T[i-189] + (0,0,1) & \text{else if } T[i-189] \neq \text{NIL} \\ \text{NIL} & \text{else} \end{cases}$$

(3) Initialize $T[0] = (0,0,0)$ and $T[i < 0] = \text{NIL}$.

(4) Return $(a, b, c) = T[n]$.

(5) Finally, the pseudocode is as follows:

```
SPECIAL(n):
  Initialize T[−189..n] ← NIL in each entry
  Initialize T[0] ← (0, 0, 0)
  for i ← 1 to n
      if T[i − 113] ≠ NIL
          T[i] ← T[i − 113] + (1, 0, 0)
      else if T[i − 155] ≠ NIL
          T[i] ← T[i − 155] + (0, 1, 0)
      else if T[i − 189] ≠ NIL
          T[i] ← T[i − 189] + (0, 0, 1)
  return T[n]
```

The running time is $O(n)$ since we spend $O(1)$ time determining each entry of T, and T has length $O(n)$.

Problem 2

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: 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.

(1) 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.

(1) Table values: $T[i]$ is the number of ways Sally can climb the stairs up to the ith step. T is indexed by $[-k..n]$.

(2) To climb to the ith step, Sally's final step is of size 1, or 2, and anything up to k. The number of ways to climb to the $i-1$st, and $i-2$nd, or anything up to $i-k$th steps have already been computed.

$$T[i] = \sum_{1 \leq s \leq k} T[i-s]$$

(3) Initialize $T[0] = 1$ (there is 1 way for Sally to be at the bottom of the stairs) and $T[i < 0] = 0$ (there is no way Sally can be below the bottom of the stairs).

(4) Return $T[n]$.

(5) Before writing the pseudocode, let us make an observation to speed up the computation of $T[i]$, which would otherwise take $O(k)$ time. We can use a sliding window to get $T[i] = \sum_{1 \leq s \leq k} T[i-s]$ from a window-value $w = \sum_{1 \leq s \leq k} T[(i-1)-s]$, by adding $T[i-1]$ and subtracting $T[(i-1)-k]$ from w. That is,

$$T[i] = w + T[i-1] - T[i-1-k],$$

which can be computed in $O(1)$ steps provided w, $T[i-1]$, and $T[i-1-k]$ have been computed. Finally, the pseudocode is as follows:

```
NumCombos(n, k):
    Initialize T[−k..n] ← 0 in each entry
    Initialize T[0] ← 1
    w ← 0
    for i ← 1 to n
        w ← w + T[i − 1] − T[i − 1 − k]
        T[i] ← w
    return T[n]
```

The loop iterates n times, and each iteration takes $O(1)$ time (see explanation above). The total running time is thus $O(n)$.

(2) 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.

(1) Table values: $T[i][j]$ is the number of ways Sally can climb the stairs up to the ith step using exactly j moves. T has dimension $[-k..n] \times [0..100]$.

(2) To climb to the ith step with exactly j moves, Sally's final step is of size 1, or 2, or anything up to k, and she'd used exactly $j - 1$ moves to get there.

$$T[i][j] = \sum_{1 \leq s \leq k} T[i - s][j - 1]$$

(3) Initialize $T[0][0] = 1$ (there is 1 way for Sally to be at the bottom of the stairs using 0 moves) and $T[i < 0][j] = 0$ for all j (there is no way Sally can be below the bottom of the stairs) and $T[i > 0][0] = 0$ (there is no way Sally can be anywhere on the stairs using 0 moves).

(4) Return $\sum_{1 \leq j \leq 100} T[n][j]$.

(5) Before writing the pseudocode, let us make an observation to speed up the computation of $T[i][j]$. The previous method of using sliding windows still works here, but elementary algebra provides an alternative method.

$$\begin{aligned} & T[i][j] = \sum_{1 \leq s \leq k} T[i - s][j - 1] \\ \implies & T[i][j] = T[i - 1][j - 1] + \sum_{2 \leq s \leq k} T[i - s][j - 1] \\ \implies & T[i][j] = T[i - 1][j - 1] + T[i - 1][j] - T[i - k - 1][j - 1]. \end{aligned}$$

Finally, the pseudocode is as follows:

```
NumCombos(n, k):
    Initialize T[−k..n][0..100] ← 0 in each entry
    Initialize T[0][0] ← 1
    for j ← 1 to 100
        for i ← 1 to n
            T[i][j] ← T[i − 1][j − 1] + T[i − 1][j] − T[i − k − 1][j − 1]
    return ∑_{1≤j≤100} T[n][j]
```

The outer loop iterates 100 times, and the inner loop iterates n times per outer-iteration. Each inner-iteration takes $O(1)$ time and so the total running time is $O(n)$.

Problem 3

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.

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
      (a) If $w_i > j$
        · ANSWER LINE 1 HERE

      (b) Else
        · ANSWER LINE 2 HERE
- Return $T[n][W]$

(Answer 1) $T[i][j] \leftarrow T[i-1][j]$

(Answer 2) $T[i][j] \leftarrow \max(T[i-1][j], T[i-1][j-w_i] + v_i)$

Problem 4

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 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.

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.

(1) Table values: $T[i][0]$ (resp. $T[i][1]$) is the length of the longest even-length (resp. odd-length) switching subsequence ending at $A[i]$. T has dimensions $[0..n-1] \times [0, 1]$.

(2) The length of the longest even-length (resp. odd-length) switching subsequence ending at $A[i]$ is one more than the length of the longest odd-length (resp. even-length) switching subsequence ending at $A[j]$ where $i < j$ and $A[j] < A[i]$ (resp. $A[j] > A[i]$).

$$T[i][0] = \begin{cases} 0 & \text{if } A[j] \geq A[i] \text{ for all } j < i \\ 1 + \max\limits_{\substack{0 \leq j < i: \\ A[j] < A[i]}} (T[j][1]) & else \end{cases}$$

$$T[i][1] = \begin{cases} 1 & \text{if } A[j] \leq A[i] \text{ for all } j < i \\ 1 + \max\limits_{\substack{0 \leq j < i: \\ A[j] > A[i]}} (T[j][0]) & else \end{cases}$$

(3) The shortest even-length switching subsequence ending at $A[i]$ is empty, and the shortest odd-length switching subsequence ending at $A[i]$ is $A[i]$ itself; these

are the minimal candidates for switching subsequences ending at $A[i]$. Initialize $T[i][0] = 0$ and $T[i][1] = 1$ for all i.

(4) Return S, the longest switching subsequence. S can be computed by backtracking on a last choice array L once T has been fully processed, starting from the maximizer of $T[i][0], T[i][1]$.

(5) Finally, the pseudocode is as follows:

```
LongestSwitchingSubsequence(A[0..n − 1]):
  Initialize T[0..n − 1][0] ← 0 in each entry
  Initialize T[0..n − 1][1] ← 1 in each entry
  // L is the last choice array.
  Initialize L[i][0, 1] ← i for all i ← 0 to n − 1
  for i ← 1 to n − 1
      for j ← 0 to i − 1
          if A[j] < A[i] and T[i][0] < 1 + T[j][1]
              T[i][0] = 1 + T[j][1]
              L[i][0] = j
          if A[j] > A[i] and T[i][1] < 1 + T[j][0]
              T[i][1] = 1 + T[j][0]
              L[i][1] = j
  // Recover the longest switching subsequence by backtracking through L.
  // Find optimal length m′ with its corresponding index i′.
  (m′, i′) ← (−∞, 0)
  for i ← 0 to n − 1
      for d ← 0 to 1
          if T[i][d] > m
              (m′, i′) ← (T[i][d], i)
  // Backtrack through L, starting from index i′. Store answer in S.
  Initialize S[0..m′ − 1] ← 0 in each entry
  while m′ > 0
      S[m′ − 1] ← A[i′]
      i′ ← L[i′][m′ mod 2]
      m′ ← m′ − 1
  return S
```

Recovering S from L and A takes $O(n)$ time. The bottleneck is computing T from the double loop, which takes $O(n^2)$ time.