CSCI 570 - Fall 2021 - HW 6 - Solution

Graded Problems

For the grade problems, you must provide the solution in the form of pseudo-code and also give an analysis on the running time complexity.

Problem 1

[**10 points**] Suppose you have a rod of length N, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars. Devise a **Dynamic Programming** algorithm to determine the maximum amount of money you can get by cutting the rod strategically and selling the cut pieces.

Solution: At each remaining length of the rod, we can choose to cut the rod at any point, and obtain points for one of the cut pieces and recursively compute the maximum points we can get for the other piece. It is also possible to recursively attempt to obtain the maximum value for both pieces after the cut, but that requires adding an extra check to see if the value obtained by selling a rod of this length is more than recursively cutting it up.

The bottom-up pseudo-code to obtain the maximum amount of money is:

Algorithm 1 Bottom-Up-Cut-Rod(p, n)

```
Let r[0, ..., n] be a new array
r[0] = 0
for j = 1 to n do
    q = −∞
    for i = 1 to j do
        q = max(q, p[i] + r[j − i])
    end for
end for
return r[n]
```

The time complexity of this algorithm is $\theta(n^2)$ because of the double nested for loop.

Rubric:

- 5 points for a correct dynamic programming solution

- 3 points if the solution runs in $\theta(n^2)$
- 2 points for providing analysis of runtime complexity

Problem 2

[**10 points**] Tommy and Bruiny are playing a turn-based game together. This game involves N marbles placed in a row. The marbles are numbered 1 to N from the left to the right. Marble i has a positive value m_i. On each player's turn, they can remove either the leftmost marble or the rightmost marble from the row and receive points equal to the sum of the remaining marbles' values in the row. The winner is the one with the higher score when there are no marbles left to remove.

Tommy always goes first in this game. Both players wish to maximize their score by the end of the game.

Assuming that both players play optimally, devise a **Dynamic Programming** algorithm to return the difference in Tommy and Bruiny's score once the game has been played for any given input.

Your algorithm **must** run in $O(N^2)$ time.

Solution:
We first calculate a prefix sum for the marbles array. This enables us to find the sum of a continuous range of values in $O(1)$ time.
If we have an array like this: $[5, 3, 1, 4, 2]$, then our prefix sum array would be $[0, 5, 8, 9, 13, 15]$.
Once we've done that, we can define OPT(i, j) as the maximum difference in score achievable by the player whose turn it is to play, given that the marbles from index i to j (inclusive) remain.

The pseudo-code for this algorithm, assuming 0-indexed arrays, is:

Algorithm 2 Max-Difference-Scores(*marbles*)

Let n be the length of the marbles array
Let $prefix\_sum$ be the calculated prefix sum array for the array marbles [takes $\theta(n)$ time]
Let $OPT[[0, ..., 0], ..., [0, ..., 0]]$ be a new $n*n$ array, with values initialized to 0

for $i = n - 2$ to 0 **do**
  for $j = i + 1$ to $n - 1$ **do**
    $score\_if\_take\_i = prefix\_sum_{j+1} - prefix\_sum_{i+1} - OPT_{i+1,j}$
    $score\_if\_take\_j = prefix\_sum_{j} - prefix\_sum_{i} - OPT_{i,j-1}$
    $OPT_{i,j} = max(score\_if\_take\_i, score\_if\_take\_j)$
  end for
end for
return $OPT_{0,n-1}$

The time complexity of this algorithm is $\theta(n^2)$ if a prefix sum array is calculated initially due to there being $n * n$ subproblems to calculate.

Rubric:

- 8 points for a correct dynamic programming solution
- 2 points for providing analysis of runtime complexity

Problem 3

[**10 points**] The Trojan Band consisting of n band members hurries to lined up in a straight line to start a march. But since band members are not positioned by height the line is looking very messy. The band leader wants to pull out the minimum number of band members that will cause the line to be in a *formation* (the remaining band members will stay in the line in the same order as they were before). The formation refers to an ordering of band members such that their heights satisfy $r_1 < r_2 < ... < r_i > ... > r_n$, where $1 \leq i \leq n$.

For example, if the heights (in inches) are given as

$$R = (67, 65, 72, 75, 73, 70, 70, 68)$$

the minimum number of band members to pull out to make a formation will be 2, resulting in the following formation:

$$(67, 72, 75, 73, 70, 68)$$

Give an algorithm to find the minimum number of band members to pull out of the line.

Note: you do not need to find the actual formation. You only need to find the minimum number of band members to pull out of the line, but you need to find this minimum number in $O(n^2)$ time.

For this question, you must write your algorithm using pseudo-code.

Solution: This problem performs a $Longest - Increasing - Subsequence$ operation twice. Once from left to right and once again, but from right to left. Once we have the values for the longest subsequence we can make after we "pull out" a few band members, we can iterate over the array and determine what minimum number of pull-outs will cause our line of band members to satisfy the height order requirements.

Let $OPT_{left}(i)$ be maximum length of the line to the left of band member i (including i) which can be put in order of increasing height (by pulling out some members) Note: the problem is symmetric, so we can flip the array R and find the same values from the other direction. Let's call those values $OPT_{right}(i)$.

The recurrence relations are:

$$OPT_{left}(i) = max(OPT_{left}(i), OPT_{left}(j) + 1) \text{ such that } r_i > r_j \quad \forall \quad 1 \leq j < i$$
$$OPT_{right}(i) = \text{same once the array is flipped}$$

Pseudo code:

for $i = 1$ to n **do**
  $OPT_{left}(i) = OPT_{right}(i) = 1$ ▷ only choose itself
end for
for $i = 2$ to $n - 1$ **do**
  for $j = 1$ to $i - 1$ **do**
    if $r_i > r_j$ **then**
      $OPT_{left}(i) = max(OPT_{left}(i), OPT_{left}(j) + 1)$
    end if
  end for
end for
for $i = 2$ to $n - 1$ **do**
  for $j = 1$ to $i - 1$ **do**
    if $r_{n-i+1} > r_{n-j+1}$ **then**
      $OPT_{right}(i) = max(OPT_{right}(i), OPT_{right}(j) + 1)$
    end if
  end for
end for
result $= -\infty$
for $i = 1$ to n **do**
  result $= min(\text{result}, n - (OPT_{left}(i) + OPT_{right}(i) + 1))$
end for
return result

The runtime of this algorithm is dominated by the nested for loops used to calculate the LIS both ways. This takes $\theta(n^2)$ each time. Therefore, the time complexity of the solution is $\theta(n^2)$.

Rubric:

- 8 points for a correct dynamic programming solution in pseudo-code
- 2 points for providing analysis of runtime complexity

Problem 4

[**15 points**] You've started a hobby of retail investing into stocks using a mobile app, RogerGood. You magically gained the power to see N days into the future and you can see the prices of one particular stock. Given an array of prices of this particular stock, where $prices[i]$ is the price of a given stock on the ith day, find the maximum profit you can achieve through various buy/sell

actions. RogerGood also has a fixed *fee* per transaction. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction. The transaction fee is on a buy-sell pair. It can be considered either during the buying process or the selling process of a stock.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Solution:

There are many ways to do this. If a greedy solution is given, the student must prove that it is correct. The dynamic programming way of solving this can be done in many ways.

One way to do it is:

Consider $buy(i)$ to be the maximum profit you can make if you start at day i, assuming you currently do not own a unit of stock.
Consider $sell(i)$ to be the maximum profit you can make starting at day i, assuming you already own a unit of the stock.

We will apply the transaction fee during the sale of a stock.

Algorithm 3 Bottom-Up-Max-Profit(*prices*, *fee*)

Let n be the length of the prices array
Let $buy[0, ..., n+1]$ be a new array, with values initialized to 0
Let $sell[0, ..., n+1]$ be a new array, with values initialized to 0

for $i = n - 1$ to 0 **do**
  $buy[i] = max(sell[i+1] - prices[i], buy[i+1])$
  $sell[i] = max(buy[i+1] + prices[i] - fee, sell[i+1])$
end for

return buy[0]

The time complexity of this solution is $\theta(n)$. We use a single for-loop to compute the maximum profits at each stage.

Rubric:

- 10 points for a correct dynamic programming solution
- 3 points if the solution runs in $\theta(n)$
- 2 points for providing analysis of runtime complexity

Practice Problems

Problem 5

[**10 points**] From the lecture, you know how to use dynamic programming to solve the 0-1 knapsack problem where each item is unique and only one of each kind is available. Now let us consider knapsack problem where you have infinitely many items of each kind. Namely, there are n different types of items. All the items of the same type i have equal size w_i and value v_i. You are offered with infinitely many items of each type. Design a dynamic programming algorithm to compute the optimal value you can get from a knapsack with capacity W.

Solution:

Similar to what is taught in the lecture, let $OPT(k, w)$ be the maximum value achievable using a knapsack of capacity $0 \leq w \leq W$ and with k types of items $1 \leq k \leq n$. We find the recurrence relation of $OPT(k, w)$ as follows. Since we have infinitely many items of each type, we choose between the following two cases:

- We include another item of type k and solve the sub-problem $OPT(k, w - w_k)$.
- We do not include any item of type k and move to consider next type of item this solving the sub-problem $OPT(k-1, w)$.

Therefore, we have

$$OPT(k, w) = \max\{OPT(k-1, w), OPT(k, w - w_k) + v_k\}.$$

Moreover, we have the initial condition $OPT(0, 0) = 0$.

Problem 6

Given n balloons, indexed from 0 to $n-1$. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get $nums[left] * nums[i] * nums[right]$ coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent. You may assume $nums[-1] = nums[n] = 1$ and they are not real therefore you can not burst them. Design an dynamic programming algorithm to find the maximum coins you can collect by bursting the balloons wisely. Analyze the running time of your algorithm.

Here is an example. If you have the nums arrays equals $[3, 1, 5, 8]$. The optimal solution would be 167, where you burst balloons in the order of 1, 5 3 and 8. The left balloons after each step is:

$$[3, 1, 5, 8] \to [3, 5, 8] \to [3, 8] \to [8] \to []$$

And the coins you get equals:

$$167 = 3 * 1 * 5 + 3 * 5 * 8 + 1 * 3 * 8 + 1 * 8 * 1.$$

Solution:

Let $OPT(l, r)$ be the maximum number of coins you can obtain from balloons $l, l+1, \ldots, r-1, r$. The key observation is that to obtain the optimal number of coins for balloon from l to r, we choose which balloon is the last one to burst. Assume that balloon k is the last one you burst, then you must first burst all balloons from l to $k-1$ and all the balloons from $k+1$ to r which are two sub problems. Therefore, we have the following recurrence relation:

$$OPT(l, r) = \max_{l \leq k \leq r} \{OPT(l, k-1) + OPT(k+1, r) + nums[k] * nums[l-1] * nums[r+1]\}$$

We have initial condition $OPT(l, r) = 0$ if $r < l$. For running time analysis, we in total have $O(n^2)$ and computation of each state takes $O(n)$ time. Therefore, the total time is $O(n^3)$.

Problem 7

Solve Kleinberg and Tardos, Chapter 6, Exercise 5.

Solution:

Let $Y_{i,k}$ denote the substring $y_i y_{i+1} \ldots y_k$. Let $Opt(k)$ denote the quality of an optimal segmentation of the substring $Y_{1,k}$. An optimal segmentation of this substring $Y_{1,k}$ will have quality equalling the quality last word (say $y_i \ldots y_k$) in the segmentation plus the quality of an optimal solution to the substring $Y_{1,i}$. Otherwise we could use an optimal solution to $Y_{1,i}$ to improve $Opt(k)$ which would lead to a contradiction.

$$Opt(k) = \max_{0 < i < k} Opt(i) + quality(Y_{i+1,k})$$

We can begin solving the above recurrence with the initial condition that $Opt(0) = 0$ and then go on to compute $Opt(k)$ for $k = 1, 2, \ldots, n$ keeping track of where the segmentation is done in each case. The segmentation corresponding to $Opt(n)$ is the solution and can be computed in $\Theta(n^2)$ time.

Problem 8

Solve Kleinberg and Tardos, Chapter 6, Exercise 6.

Solution:

Let $W = \{w_1, w_2, \ldots, w_n\}$ be the set of ordered words which we wish to print. In the optimal solution, if the first line contains k words, then the rest of the lines constitute an optimal solution for the sub problem with the set $\{w_{k+1}, \ldots, w_n\}$. Otherwise, by replacing with an optimal solution for the rest of the lines, we would get a solution that contradicts the optimality of the solution for the set $\{w_1, w_2, \ldots, w_n\}$.

Let $Opt(i)$ denote the sum of squares of slacks for the optimal solution with the words $\{w_i, \ldots, w_n\}$. Say we can put at most the first p words from w_i to w_n in a line, that is, $\sum_{t=i}^{p+i-1} c_t + p - 1 \leq L$ and $\sum_{t=1}^{p+i} w_t + p > L$. Suppose the first k words are put in the first line, then the number of extra space characters is

$$s(i,k) := L - k + 1 - \sum_{t=i}^{i+k-1} c_t$$

So we have the recurrence

$$Opt(i) = \begin{cases} 0 & \text{if } p \geq n - i + 1 \\ \min_{1 \leq k \leq p}\{(s(i,k))^2 + Opt(i+k)\} & \text{if } p < n - i + 1 \end{cases}$$

Trace back the value of k for which $Opt(i)$ is minimized to get the number of words to be printed on each line. We need to compute $Opt(i)$ for n different values of i. At each step p may be asymptotically as big as L. Thus the total running time is $O(nL)$.