Homework 5

CSCI 570

Summer 2023

Problem 1

Solve Kleinberg and Tardos, Chapter 6, Exercise 5.

Solution:

Let $Y_{i,k}$ denote the substring $y_i y_{i+1} \dots y_k$. 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 $y_i \dots y_k$) 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 $k = 1, 2, \dots, n$ 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.

Rubric:

- 8 points for a correct dynamic programming recurrence relation with definitions
- 2 points for providing analysis of runtime complexity

Problem 2

Adam wants to surprise his fiance with exquisite jewellery made from gold. He owns many gold mines and wants the goldsmith to make different items. He gives the goldsmith a list of n different items, each of different weights w_i grams and each having value v_i. Adam's fiance likes jewellery but only has space for some items, as her jewellery box has a maximum capacity of W grams. Adam wants the goldsmith to make items from the list and insists that the combined weight of all the items must be equal to or less than W grams, but at the same time maximizing the total value of all the created items. Design a DP algorithm to output the total optimal value of the items, whose total weight doesn't exceed

W grams.
Hint: Adam can procure as much gold as required for making those items, as his fiance has a small jewellery box.

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 $OPT(0, 0) = 0$.

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 3

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 1 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 4

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 $O(nL)$.

Rubric:

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

Problem 5

You have n+1 rooms. There is a heater in Room 0. It can make Room 0 to Room r warm. ($1 \leq$ r $\leq$ n) Now you are a robot with k thermometers. Every time you bring one thermometer into a room. If the room is warm, the thermometer will work well and can be used again, and if the room is cold, the thermometer will be broken and can not be used again. Design a dynamic program algorithm to find the minimum count of entering a room that you need to determine the exact value of r.
a. Write down the recursive formula, and the meaning of each part.
b. What is the time complexity of the algorithm?

Solution:

1. dp(k,n) is the optimal count when there are k thermometers and n floors.
   $dp(k,n) = 1 + \min_{1 \leq x \leq n}(\max(dp(k-1, x-1), dp(k, n-x)))$
   dp(k-1, x-1) is the state when the thermometer is broken.
   dp(k, n-x) is the state when the thermometer works well. x denotes the Room x.

2. O(kn^2), or O(kn log n) [if use binary search]

Rubric:

- 8 points for a correct dynamic programming recurrence relation with definitions
- 2 points for providing analysis of runtime complexity

Problem 6

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 $= min($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