Homework 4
Sara Pesavento
spesaven@usc.edu
7618843154

Sara Pesavento
Homework 6
7618843154

1. 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 pi 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.

Subproblems:
OPT(i) = the maximum amount of money you can get from strategically cutting the pieces in a rod of length i.
For the last piece of the rod, there must be some location j (0 < j < N) where the rod is cut to create an optimal solution.
0 ---------- j ------ i
So we can try cutting the rod at different **j** locations from 1 to i. Following this logic, the recurrence formula would be:
OPT(i) = $max_{(0<j<i)}$(p(j) + OPT(i - j)) meaning max (price of rod length j + optimal last piece))

The bottom up approach would then be:
OPT(0) = 0
For i in 1….n:
OPT(i) = $max_{(0<j<i)}$(p(j) + OPT(i - j))
endfor
Return OPT(n)

Sample loop:
First start at i = 1
Then loop j from 1 to i (1)
OPT(1) = p(1) + OPT(0)
Start at i = 2
Then loop j from 1 to i (2)
OPT(2) = max (p(1) + OPT(1) , p(2) + OPT(0)) 0 --- j --- i or just 0 -------- i

The run-time of this solution is O(n^2) (due to the nested for loop).

2. 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 mi. 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.

1 ……. N
m1 ….. mN
We'll need to use a prefix sum to be able to compute the sum of a continuous range of values in O(1) time.
e.g. [5,3,1,4,2] → [0,5,8,9,13,15] (example from solution)

If Tommy goes first, then they have a choice to remove one marble from the left side OR from the right side. They will remove the one that maximizes the prefix sum.

Let's say a particular game has marbles from i …. j.

Subproblem:
OPT(i, j) = the maximum difference in score achievable by the player whose turn it is to play, given that the marbles from index i through j remain

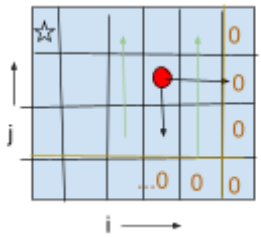
If they take a marble from the left side (lowest), they'd get:
- prefix_sum(j + 1) - prefix_sum(i + 1) - OPT(i + 1, j) #score without left marble - (max diff in score from the rest of the marbles)

(e.g. [5,3,1,4,2] values
[0,5,8,9,13,15] prefix → 15 - 5 = 10

If they take a marble from the right side, they'd get: #score without right marble - (max diff in score from the rest of the marbles)
- prefix_sum(j) - prefix_sum(i) - OPT(i, j - 1)

Bottom up algorithm:
n = [1.....n] #marbles array length N
prefix_sum = [0……] #calculate from marbles array in O(n)

dp = [[0] * N for _ in range(N)] #for memoization



#start at the bottom right and fill up column by column
for i in n…2 to 0:
  for j in i + 1 to n - 1:
    remove_left_i = prefix_sum(j + 1) - prefix_sum(i + 1) - OPT(i + 1, j)
    remove_right_j = prefix_sum(j) - prefix_sum(i) - OPT(i, j - 1)
    OPT(i, j) = max(remove_left_i, remove_right_j)
  endfor
endfor
return OPT(0, n - 1)

The time complexity is O(n^2) because we use a prefix sum and there are n*n subproblems.

3. The Trojan Band consisting of n band members hurries to line 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 r1< r2< ... < ri > ... > rn, where 1 <= i <= 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.

This problem requires that there be a band member ri (1 <= i <= n) that serves as a "peak" in the heights of the band members.
minimum pull from left side, ri, minimum pull from right side.
Therefore, given that i is a peak, we want to find the longest line from each end through i that makes i the peak (increasing heights up till i).

Subproblems:
OPT_left(i) = the maximum length of the line with increasing heights formed from band member 1 through i, given that we can remove some members.
OPT_right(i) = the maximum length of the line with increasing heights formed from band member n through i, given that we can remove some members.
Then the final answer will involve looping through the line and finding the "peak" that results in the minimum number of removals.

Recurrence:
OPT_left(i) = $max_{(0<j<i),\, rj < ri}$(OPT_left(i), OPT_left(j) + 1) #max of choosing i, or going 1 to the right
OPT_right(i) = $max_{(0<j<i),\, rj < ri}$(OPT_right(i), OPT_right(j) + 1)

Bottom up pseudocode:
#initialize each value to 1 because they are each of length 1 (no longer lines formed yet)

```
For i in 1 … n:
    OPT_left(i) = 1
    OPT_right(i) = 1

#calculate the longest line formed from the left side of the array.
for i in 2 ... n:
   for j in 1 ... i - 1:      #lags behind and counts all band members SHORTER than band member i (thus part of the line)
        if rj < ri:
            OPT_left(i) = max(OPT_left(i), OPT_left(j) + 1)   #keep current max or add i to the max line up till j

#calculate the longest line formed from the right side of the array.
#(Fill in left to right, this way when you compare pos i, it's longest from 1 to i, and longest from i to n)
for i in 2 ... n:
   for j in 1 ... i - 1:
       if r(n - j + 1) < r(n - i + 1):
            OPT_right(i) = max(OPT_right(i), OPT_right(j) + 1)   #keep current max or add i to the max line up till j
```

```
minimum_removals = -inf
for i in 1 ... n:
    minimum_removals = min(minimum_removals, n - (OPT_left(i) + OPT_right(i) + 1))
return minimum_removals
```

The time complexity is O(n^2)

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

OPT(i) = the maximum profit that can be achieved from day i forward.

On any given day i, you either have stock already, or you don't. We can expand these as follows.
If you don't have any stock, you have two choices:

1. Buy a unit of stock today, and the profit you make will be the best sell point after today the profit will be the best sell. So you have: sell[i + 1] minus the cost of buying the stock today.
   sell[i + 1] - prices[i]
2. Don't buy any stock today. In this case, we consider buying it sometime from the next day onward.
   buy[i + 1]

If you HAVE stock, you have two choices:

1. Sell the stock today and gain prices[i] minus the transaction fee. Then continue buying at a later date.
   buy[i + 1] + prices[i] - fee
2. Don't sell today. Hold so you can sell it on a later day.
   sell[i + 1]

Recurrence:
OPT_buy(i) - maximum profit that can be achieved from day i forward if you **don't** have any stock.
OPT_sell(i) - maximum profit that can be achieved from day i forward if you already have stock.

```
OPT_buy(i) = max(sell[i + 1] - prices[i], buy[i + 1])
OPT_sell(i) = max(buy[i + 1] + prices[i] - fee, sell[i + 1])
```

In the end, we want to return the value of OPT_buy[0], since on the first day we don't have any stock.

Bottom up approach:
initialize arrays OPT_buy and OPT_sell as arrays [0] * N

```
for i in n … 1:
    OPT_buy(i) = max(sell[i + 1] - prices[i], buy[i + 1])
```

```
    OPT_sell(i) = max(buy[i + 1] + prices[i] - fee, sell[i + 1])
return OPT_buy[0]
```

The run time complexity would be O(n)