Homework 7

Jiahao Liu March 2, 2024

1. A car factory has k assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i indicates that the station is on the i^{th} assembly line (0<i<=k), and j indicates the station is the j^{th} station along the assembly line (0<j<=n). Each station is dedicated to some sort of work like engine fitting, body fitting, painting and so on. So, a car chassis must pass through each of the n stations in that order before exiting the factory. The time taken per station is denoted by $a_{i,j}$. Parallel stations of the k assembly lines perform the same task. After the car passes through station $S_{i,j}$, it will continue to station $Si, j+1$ unless we decide to transfer it to another line. Continuing on the same line incurs no extra cost, but transferring from line x at station j-1 to line y at station j takes time $t_{x,y}$. Each assembly line takes an entry time e_i and exit time x_i which may be different for each of these k lines. Give an algorithm for computing the minimum time it will take to build a car chassis.

a. Define T(i,j): the minimum time needed to build a car chassis through j stations along the assembly line and reaches station $S_{i,j}$, include entry time for j = 1 and exit time for j = n.
b. Initialization: $\forall\ 0 < i <= k,\ T(i, 1) = e_i + a_{i,1},\ \forall\ 1 < j <= n,\ T(i, j) = \infty$
b. Recurrence Formula: Let's assume $\forall\ 0 < i <= k,\ t_{i,i} = 0$

$$T(i,j) = \begin{cases} \min\limits_{0<s<=k} \{T(s, j-1) + t_{s,i}\} + a_{i,j} & \text{if } j < n \\ \min\limits_{0<s<=k} \{T(s, j-1) + t_{s,i}\} + a_{i,j} + x_i & \text{if } j = n \end{cases}$$

c. Describe (using pseudocode) how the value of an optimal solution is obtained using iteration.

```
Bottom Up Pass:
Initialize T[i][1] = e[i] + a[i][1] for 0 < i <= k, T[i][j] = $\infty$ for 0 < i <=k, 1 < j <= n
for j = 2 to n:
    for i = 1 to k:
        for s = 1 to k:
            if j < n:
                if s == i: T[i][j] = min(T[i][j], T[s][j-1] + a[i][j])
                else: T[i][j] = min(T[i][j], T[s][j-1] + t[s][i] + a[i][j])
            else:
                if s == i: T[i][j] = min(T[i][j], T[s][j-1] + a[i][j] + x[i])
                else: T[i][j] = min(T[i][j], T[s][j-1] + t[s][i] + a[i][j] + x[i])
return $\min_{0<i<=k} \{T[i][n]\}$
```

d. Time Complexity: $\Theta(k^2 n)$, it is polynomial time in the size of the input a[k][n] and t[k][k].

2. There are n trading posts along a river numbered n, n-1,... 3, 2, 1. At any of the posts you can rent a canoe to be returned at any other post downstream. (It is impossible to paddle against the river since the water is moving too quickly). For each possible departure point i and each possible arrival point j(< i), the cost of a rental from i to j is known C[i, j]. However, it can happen that the cost of renting from i to j is higher than the total costs of a series of shorter rentals. In this case you can return the first canoe at some post k between i and j and continue your journey in a second (and, maybe, third, fourth, ...) canoe. There is no extra charge for changing canoes in this way. Give a dynamic programming algorithm to determine the minimum cost of a trip by canoe from each possible departure point i to each possible arrival point j. For your dynamic programming solution, focus on computing the minimum cost of a trip from trading post n to trading post 1, using up to each intermediate trading post.

a. Define f(i, j): the minimum cost of a trip from trading post i to trading post j.
b. Initialization: $\forall 1 < i \leq n,\ 1 \leq j < i,\ f(i, j) = \infty$. $\forall\ 1 \leq i \leq n,\ f(i, i) = 0$
b. Recurrence Formula: Consider the following in increasing order of i-j.

$$f(i, j) = \min(C_{i,j}, \min_{j<k<i} \{f(i, k) + f(k, j)\})$$

c. What will be the iterative program for the above?

```
Bottom Up Pass:
Initialize f[i][i] = 0 for 1 <= i <= n, f[i][j] = ∞ for 1 < i <= n, 1 <= j < i
for j = n-1 to 1 by -1:
    for i = n to j+1 by -1:
        f[i][j] = min(f[i][j], C[i][j])
        for k = j+1 to i-1:
            f[i][j] = min(f[i][j], f[i][k] + f[k][j])
return f[n][1]
```

d. Time Complexity: The total time complexity with three loops is $\Theta(n^3)$, it is polynomial time in the size of the input C[n][n].

3. Alice and Bob are playing an interesting game. They both each have a string, let them be a and b. They both decided to find the biggest string that is common between the strings they have. The letters of the resulting string should be in order as that in a and b but don't have to be consecutive. Discuss its time complexity.

This is a simpler version of DNA Sequence Alignment Problem during this week's lecture.
a. Define f(i, j): the length of the longest common subsequence of a[1..i] and b[1..j].
b. Initialization: Say len(a) = m, len(b) = n: $\forall\, 0 \le i \le m,\ f(i, 0) = 0$, $\forall\, 0 \le j \le n,\ f(0, j) = 0$
c. Recurrence Formula: $\mathbf{1}_{\{a_i=b_j\}} = 1$ if $a_i = b_j$, otherwise 0.

$$f(i, j) = \min(f(i-1, j-1) + \mathbf{1}_{\{a_i=b_j\}}, f(i-1, j), f(i, j-1))$$

d. Bottom Up Pass:

```
Bottom Up Pass:
Initialize f[i][0] = 0 for 0 <= i <= m, f[0][j] = 0 for 0 <= j <= n
for i = 1 to m:
    for j = 1 to n:
        if a[i] == b[j]:
            f[i][j] = max(f[i-1][j-1] + 1, f[i-1][j], f[i][j-1])
        else:
            f[i][j] = max(f[i-1][j-1], f[i-1][j], f[i][j-1])
return f[m][n]
```

We also need to backtrack the grid to find the subsequence.

```
Initialize i = m, j = n, result = " "
while i > 0 and j > 0:
    if a[i] == b[j]:
        result += a[i]
        i -= 1, j -= 1
    else if f[i-1][j] > f[i][j-1]:
        i -= 1
    else:
        j -= 1
return reverse(result)
```

e. Time Complexity: The time complexity is $\Theta(mn)$, it is polynomial time in the size of the input a and b.

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.

Define f[i][0]: the maximum profit we can make starting at day i without a unit of stock. **f[i][1]:** the maximum profit we can make starting at day i with a unit of stock.
Initialization: $\forall\, 0 \le i <= N+1 : f[i][0] = f[i][1] = 0$
Recurrence Formula: Suppose we only apply transaction fee δ during the sale of a stock:

$$f[i][0] = \max(-price[i] + f[i+1][1], f[i+1][0]) \qquad f[i][1] = \max(f[i+1][0] + price[i] - \delta, f[i+1][1])$$

Bottom Up Pass:

```
for i = n to 0 by -1:
    f[i][0] = max(-price[i] + f[i+1][1], f[i+1][0])
    f[i][1] = max(f[i+1][0] + price[i] - $\delta$, f[i+1][1])
return f[0][0]
```

Total Time Complexity would be $O(n)$, which is Strong Polynomial in number of integers of input.

Let's first traverse the price array to find all the price increasing subsequences, denoted its index as $[l_i, h_i]$. Suppose there are n such subsequences, then $1 \le l_1 \le h_1 \le l_2 \le h_2 \dots l_n \le h_n \le N$.

a. Define f(i, j): the maximum profit we can achieve through various buy/sell actions from day l_i to day h_j.

b. Initialization: Suppose each transaction fee is δ, $\forall\ 1 \le i \le n,\ f(i,i) = h_i - l_i - 2\delta$, all other $f(i,j) = 0$

c. Recurrence Formula: Consider the following in increasing order of j-i.

$$f(i,j) = \max(h_j - l_i - 2\delta, \max_{i \le k < j}\{f(i,k) + f(k+1,j) - 2\delta\})$$

d. Pseudocode to solve the problem by Bottom Up Pass:

```
Bottom Up Pass:
Initialize f[i][j] = 0 for 1 <= i <= n, 1 <= j <= n, f[i][i] = h[i] - l[i] - 2*$\delta$ for 1 <= i <= n,
for len = 1 to n-1:
    for i = 1 to n-len:
        j = i + len
        f[i][j] = max(f[i][j], h[j] - l[i] - 2*$\delta$)
        for k = i to j-1:
            f[i][j] = max(f[i][j], f[i][k] + f[k+1][j] - 2*$\delta$)
return f[1][n]
```

e. Time Complexity: The time complexity for finding $[l_i, h_i]$ takes $\Theta(N)$. The time complexity for Bottom Up Pass is $\Theta(n^2) < \Theta(N^2)$. So the total time complexity is $\Theta(N^2)$, which is polynomial time of input prices[N].

5. 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.

Define f(i, j): The score difference in Current Player and his Competitor's once the game has been played for marbles indexed from i to j, i.e. only marbles with value $m_i, \dots, m_j$.
Note that the Current Player in f(i, j) definition does not have to be Tommy, the required difference in Tommy and Bruiny's score is $(-1)^{N-(j-i+1)} * f(i,j)$
Initialization: $\forall\ 1 \le i \le N, 1 \le j \le i,\ f(i,j) = 0$
Recurrence Formula:

$$f(i,j) = max(\sum_{k=i+1}^{j} m_k - f(i+1,j), \sum_{k=i}^{j-1} m_k - f(i,j-1))$$

Note that we can use prefix sum to calculate $\sum_{k=i}^{j} m_k$ in constant time.

```
Bottom Up Pass:
Initialize f[i][j] = 0 for 1 <= i <= N, 1 <= j <= i
for i = N - 1 to 1 by -1:
    for j = i+1 to N-1 by 1:
        $f[i][j] = max(\sum_{k=i+1}^{j} m_k$ - f[i+1][j], $\sum_{k=i}^{j-1} m_k$ - f[i][j-1])
return f[1][N]
```

Time Complexity: The time complexity for Bottom Up Pass is $\Theta(N^2)$, which is polynomial time of input marble values m[N].

6. Consider a group of people standing in a line of size n. Everyone's heights are given in an array height $[h_0, h_1, \dots, h_{n-1}]$. Write an efficient algorithm to find the longest subsequence of people with strictly increasing heights and return it. Give the pseudo code and the recurrence relation.

Define f[i]: the length of the longest subsequence of people with strictly increasing heights and end with height h_i. **Initialize:** $\forall 0 \le i < n, f[i] = 1$. **Recurrence Formula:**

$$f[i] = \max_{0 \le j < i} \{f[j] + \mathbf{1}_{h[j] < h[i]}\}$$

Bottom Up Pass:

```
Initialize an string array Seq[n] to store the sequence index string end with i.
for 0 <= i < n: f[i] = 1, Seq[i] = " " + i;
for i = 1 to n-1:
    for j = i-1 to 0 by -1:
        if h[i] > h[j]:
            f[i] = max(f[i], f[j] + 1)
            Seq[i] = Seq[j] + " " + i;
return $\max_{0 \le i < n}\{f[i]\}, Seq[i]$
```

a. Define f(i): the length of the longest subsequence of people with strictly increasing heights for the first i person. **Define End(i):** the index of highest person in the required subsequence for the first i person, if there are more than one sequence, store the one with **smaller hieghest person**.

b. Initialization: $\forall\ 0 \le i < n,\ f(i) = 1, End(i) = -1$

c. Recurrence Formula:

$$f(i) = \max_{0 \le j < i} \{f(j) + \mathbf{1}_{h[i] > h[End(j)]}\}$$

$End(i)$ the index of above j with max value, store the sequence with smaller highest hieght.

d. Pseudocode to solve the problem by Bottom Up Pass and Top Down Pass:

```
Bottom Up Pass:
Initialize f[i] = 1, End[i] = -1 for 0 <= i < n
for i = 1 to n-1:
    for j = i-1 to 0:
        if h[i] > h[End[j]]:
            if f[j] + 1 >= f[i]:
                f[i] = f[j] + 1
                End[i] = End[j]
return f[n]
```

```
Top Down Pass:
Initialize i = n
while i >= 0:
    output h[End[i]]
    i = End[i]
```

e. Time Complexity: The time complexity for Bottom Up Pass is $\Theta(n^2)$, which is polynomial time of input heights h[n]. The time complexity for Top Down Pass is $\Theta(n)$.

e. Space Complexity: The space complexity for Bottom Up Pass is $\Theta(n)$, which is polynomial space of input heights h[n].