HOMEWORK 7

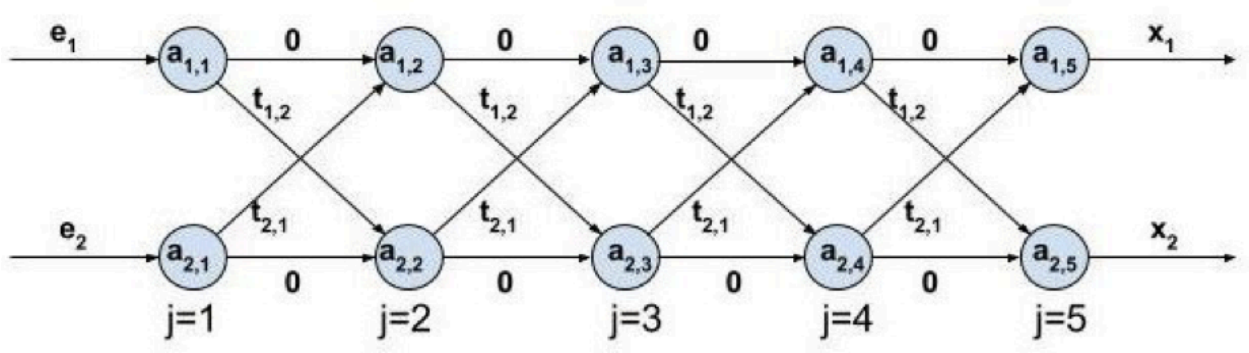
Q1. A car factory has k assembly lines, each with n stations. A station is denoted by $S_{i,j}$ where i indicates that the station is on the i-th assembly line (0<i<=k), and j indicates the station is the j-th station along the assembly line (0<j<=n). Each station is dedicated to some sort of work like 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 $S_{i,j+1}$ 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.

Below figure provides one example to explain the problem. The example shows k=2 assembly lines and 5 stations per line. To illustrate how to evaluate the cost, let's consider two cases: If we always use assembly line 1, the time cost will be:

$e_1 + a_{1,1} + a_{1,2} + a_{1,3} + a_{1,4} + a_{1,5} + x_1$

If we use stations $s_{1,1}$ -> $s_{2,2}$ -> $s_{1,3}$ -> $s_{2,4}$ -> $s_{2,5}$, the time cost will be :

$e_1 + a_{1,1} + t_{1,2} + a_{2,2} + t_{2,1} + a_{1,3} + t_{1,2} + a_{2,4} + a_{2,5} + x_2$



a) Define (in plain English) subproblems to be solved. (2 pts)

Subproblem: minCost(i,j) will be the minimal cost so far, if we use assembly line-i at station-j

b) Recursively define the value of an optimal solution (recurrence formula) (8 pts)

Initialization: minCost(1,1) = e_1+a_{11}, minCost(2,1) = e_2+a_{21}, ... minCost(k,1) = e_k + a_{k1} (1 point)
Recurrence relation: minCost(i,j)= min(1≤l≤k) minCost(l,j-1) + $t_{l,i}$ + $a_{i,j}$ (4 points)
(in this formulation)

Final cost: minCost(i,exit) = minCost(i,n) + x_i (1 point)

Attention: Some students may assume that the car can only pass to the station (i,j) from adjacent lines i-1 and i+1. In this case, you will have 3 points off.

c) Describe (using pseudocode) how the value of an optimal solution is obtained using iteration. Make sure you include any initialization required. (8 pts)

```
// initialization, boundary condition
for i=1:k
        minCost[i][1] = e[i]+a[i][1] (1 point)

// recurrence
For t=2:n
        For i=1:k
                minCost[i][t] = minCost[1][t-1] + t[1][i]+a[i][t]
        for j=2:k
                minCost[i][t] = min(minCost[j][t-1] + t[j][i]+a[i][t], minCost[i][t]) (4 points)

for i=1:k
        minCost[i][n] = minCost[i][n]+x[i]

// final solution

return min( minCost[1][n], minCost[2][n], ... minCost[k][n] ) (1 point)
```

d) What is the complexity of your solution in part b? (4 pts)

From the recurrence loops, we can tell the complexity is O(n×(k2))

Q2.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. It is *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 (in plain English) subproblems to be solved. (2 point)

Let OPT(i,j) = The optimal cost of reaching from departure point “i” to departure point “j”.

b) Recursively define the value of an optimal solution (recurrence formula) (4pts)

Now, let's look at OPT(i,j). assume that we are at post “i”. If we are not making any intermediate stops, we will directly be at “j”. We can potentially make the first stop, starting at “i” to any post between “i” and “j” (“j” included, “i” not included)
This gets us to the following recurrence

OPT(i,j) = min(over j<= k < i)(C[i,k] + OPT(k,j))

The base case is OPT(i,i) = C[i,i] = 0 for all “i“ from 1 to n

c) What will be the iterative program for the above ? (5 points)

Let OPT[i,j] be the array where you will store the optimal costs. Initialize the 2D array with the base cases

```
for (i=1;i<=n;i++)
{
        for (j=1;j<=i-1;j++)
        {
        calculate OPT(i,j) with the recurrence
        }
}
```

Now, to output the cost from the last post to the first post, will be given by OPT[n,1]

d) Analyze the running time of your algorithm in terms of *n*. (4 pts)

For the running time, we are trying to fill up all OPT[i,j] for i<j. Thus, there are $O(n^2)$ entries to fill (which corresponds to the outer loops for “i” and “j”) In each loop, we could potentially be doing at most k comparisons for the min operation . This is O(n) work. Therefore, the total running time is $O(n^3)$

Another possible solution:

Q3 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.

Write the subproblems in English, Recurrence Relation and Pseudo code as well (20 Points)

a) Define (in plain English) subproblems to be solved. (2 pts)

Let dp[i][j] be the length of the longest common subsequence using the first i characters of string a and first j character of string b

b) Write down the base cases: (2 Points)

```
if i==0 or j==0:
    dp[i][j] = 0
```

c) Write the recurrence relation: (6 Points)

We can use Bottom Up DP for this problem.

```
if (a[i-1] != b[j-1]):
    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
else:
    dp[i][j] = dp[i - 1][j - 1] + 1;
```

d) Write the pseudoCode for telling if such a string can be found and if so, find the resulting string. (8 Points)

```
for i … n
  for j….m:
     if i==0 or j==0 dp[i][j] = 0
     Else  if (a[i-1] != b[j-1]):
         dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
     else:
         dp[i][j] = dp[i - 1][j - 1] + 1;

ans_len = dp[n][m] // this will give us the length

// Now we can backtrack the grid to find the subsequence. This will give us the subsequence.

seq = “”
i = n, j=m
while i>0 and j>0:
    If a[i-1] == b[j-1]:
        seq += a[i-1]
```

```
            j-=1
            i-=1
        else if dp[i-1][j] > dp[i][j-1]
            i-=1
        else
            j-=1
reverse (seq)
return [ans_len, seq]  //return len of seq and the seq
```

e) Write the time complexity (2 Points)

O(n*m), where n, m are the size of the strings a and b respectively.

UNGRADED QUESTIONS

Q4. 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. (20 pt) Solution:

a. Define (in plain English) subproblems to be solved. (4 pts)

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.

b. Write a recurrence relation for the subproblems (6 pts)

We will apply the transaction fee during the sale of a stock. The recurrence relation is as follows: buy(i) = max(sell(i + 1) − prices(i), buy(i + 1)) sell(i) = max(buy(i + 1) + prices(i) − fee, sell(i + 1))

c. Using the recurrence formula in part b, write pseudocode to solve the problem. (5 pts)

Algorithm: Input: an arrays: prices, of length N for N days containing the stock price and fee for each day. A number fees denoting the fee per transaction Initialize: 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 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: return buy[0]

d. Make sure you specify :

1. base cases and their values (2 pts)

   buy[0, ..., n + 1] initiallized to 0 sell[0, ..., n + 1] initiallized to 0.

2. where the final answer can be found (1 pt)

   buy[0]

e. What is the complexity of your solution? (2 pts)

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

Q5. 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.

a) Define (in plain English) subproblems to be solved. (2 pts)

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.

b) Write down the recurrence relation (8 Points)

We first calculate a prefix sum for the marbles array. This enables us to find the sum o 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].

score if take i = prefix sum sum_{j+1} − *prefix* sum_{i+1} − *OP* $T_{i+1,j}$
score if take j = prefix sum_j − *prefix* sum_i − *OP* $T_{i,j-1}$

OPT(i,j) = *max*(*score if take i, score if take j*)

c) Write down the the pseudo-code for this algorithm (8 Points)

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 *OP T*[[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 sum_{j+1} − *prefix* sum_{i+1} − *OP* $T_{i+1,j}$ *score if take j = prefix* sum_j − *prefix* sum_i − *OP* $T_{i,j-1}$
OP $T_{i,j}$ = *max*(*score if take i, score if take j*)
end for
end for
return *OP* $T_{0,n-1}$

d) Write down the time complexity (2 Points):

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.

Q6. Consider a group of people standing in a line of size n. Everyone's heights are given in an array height = [h0, h1, ….. Hn-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.

For example: height = [6,1,2,3,4,5], ans = [1,2,3,4,5]

One of the solutions:

a) Define (in plain English) subproblems to be solved. (2 pts)

dp[i] = length of subsequence with maximum number of people with strictly increasing heights and the end element is ith person.

b) Write down the base cases (1 Points)

dp[i] = 1 for all i<=n-1 and i>=0

c) Write down the recurrence relation (3 Points)

dp[i] = max(dp[i], dp[j] + 1) for all 0<=j and j<i considering height[i] > height[j].

Final answer can be found at max(dp[i]) for all i<=n-1 and i>=0.

d) Write down the pseudoCode: (4 Points)

```
For 0<=i<n:
  val[i] = [i]
   For 0<=j<i:
        If height[i] > height[j]:
             If dp[j] + 1 > dp[i] :
                dp[i] = dp[j] + 1
                val[i] = val[j] + [i]
```

e) Write down the time complexity and space complexity (2 Points)

Time Complexity : O(n^2), Space Complexity: O(n)