Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

1) Answer to Question 1:

Input: The input is a $2^k x 2^k$ (where $k \in N$) matrix with all legal tile positions set to 0. The defective tile position is set to -1. Let n denote the dimension(s) of the matrix.

Divide Step: Recursively divide the $2^n x 2^n$ matrix into 4 smaller sub-matrices of size $2^{n-1} x 2^{n-1}$ in each recursive call.

Conquer Step: Tile each smaller sub-matrix in valid positions.

Combine Step: Fit the final L-shaped tile properly in the centre of the matrix which was divided into the 4 smaller sub-matrices.

Key Point: We notice that the defective tile must lie in exactly one of the 4 submatrices/quadrants divided in each recursive call.

Subproblems: Our subproblems are tiling smaller matrices of dimensions $2^{k-n} x 2^{k-n}$ where $1 \leqslant n \leqslant k-1$. We tile each of these smaller matrices and then finally complete the tiling of the bigger matrix (formed by 4 of these smaller matrices). The completion of tiling of the bigger matrix is the merge step.

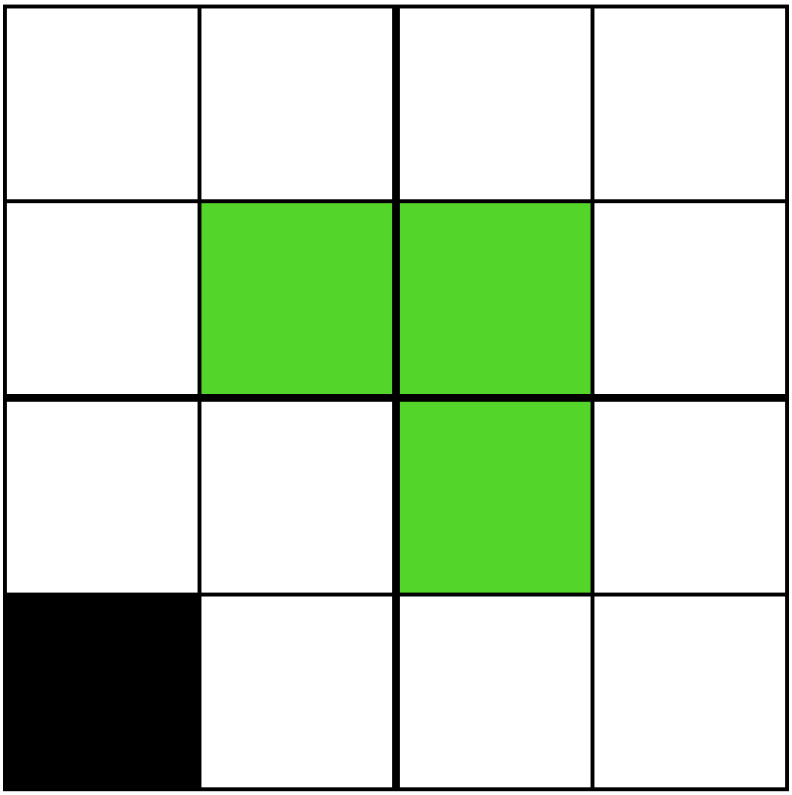
Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Explanation of algorithm:

Since we noticed that we have to align the tile at the centre of a matrix according to the position of the defective tile, we first find the quadrant of the defective tile in the matrix and place the tile at the centre accordingly. This is the combine step which we have performed before dividing the matrix into submatrices. Therefore in each combine step, we will find the quadrant in which the defective tile lies and place the tile at the centre such that it covers tiles from all the quadrants except the one in which the defective tile lies (as shown in the figure given below).



We then divide our matrix into 4 smaller sub matrices and pass the correctly placed tile as the defective tile for the smaller submatrix so that the same tile is not placed upon once again or basically, to avoid any overlappings. Then, we'll perform the same steps as above.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

When the matrix becomes of 2x2 size, we reach the base case and simply fill any tiles that have not already been tiled. Note that 1 of these 4 tiles will either be defective, in which case we don't tile it, or the tile would have already been covered by the tile placed at the centre of a bigger matrix (described above in the merge/combine step).
We also maintain a global 'counter' variable initialised to 1. After tiling every 3 positions the counter variable increments by one. This is done to indicate specific L shaped tiles in the matrix.
We then simply return from the function and don't have to perform any other merge step as it has already been taken care of as explained above.

Base Case: When n=2, cover all the tiles that have not been covered and are not defective.

Pseudocode:

```
TileMatrix(n,matrix,x,y,defective_x,defective_y)

If n=2 do
        For x ⩽ i< x+n do
            For y ⩽ j< y +n do
                If i=defective_x and j=defective_y do
                        Pass
                End
                Else
                        matrix[i][j] = counter
                End
            End
        End
        counter = counter+1
End
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

```
//If the missing tile is in the 1st quadrant, cover the tiles towards the centre in the rest of the quadrants and increment counter by 1
//If the missing tile is in the 2nd quadrant, cover the tiles towards the centre in the rest of the quadrants and increment counter by 1
//If the missing tile is in the 3rd quadrant, cover the tiles towards the centre in the rest of the quadrants and increment counter by 1
//If the missing tile is in the 4th quadrant, cover the tiles towards the centre in the rest of the quadrants and increment counter by 1
If defective_x < x+n/2 AND defective_y < y + n/2 do
        defective_x1 = defective_x
        defective_y1 = defective_y
        defective_x2 = x + n/2 - 1
        defective_y2 = y + n/2
        defective_x3 = x + n/2
        defective_y3 = y + n/2 - 1
        defective_x4 = x + n/2
        defective_y4 = y +n/2
        matrix[defective_x2][defective_y2] = counter
        matrix[defective_x3][defective_y3] = counter
        matrix[defective_x4][defective_y4] = counter
        counter = counter + 1
End
If defective_x < x+n/2 AND defective_y >= y + n/2 do
        defective_x1 = x + n/2 - 1
        defective_y1 = y + n/2 - 1
        defective_x2 = defective_x
        defective_y2 = defective_y
        defective_x3 = x + n/2
        defective_y3 = y + n/2 -1
        defective_x4 = x + n/2
        defective_y4 = y + n/2
        matrix[defective_x1][defective_y1] = counter
        matrix[defective_x3][defective_y3] = counter
        matrix[defective_x4][defective_y4] = counter
        counter = counter + 1
End
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

```
If defective_x >=x+n/2 AND defective_y < y + n/2 do
        defective_x1 = x + n/2 - 1
        defective_y1 = y + n/2 -1
        defective_x2 = x + n/2 -1
        defective_y2 = y + n/2
        defective_x3 = defective_x
        defective_y3 = defective_y
        defective_x4 = x + n/2
        defective_y4 = y + n/2
        matrix[defective_x1][defective_y1] = counter
        matrix[defective_x2][defective_y2] = counter
        matrix[defective_x4][defective_y4] = counter
        counter = counter + 1
End
If defective_x >= x+n/2 AND defective_y >= y + n/2 do
        defective_x1 = x + n/2 -1
        defective_y1 = y + n/2 -1
        defective_x2 = x + n/2 -1
        defective_y2 = y + n/2
        defective_x3 = x + n/2
        defective_y3 = y + n/2 -1
        defective_x4 = defective_x
        defective_y4 = defective_y
        matrix[defective_x1][defective_y1] = counter
        matrix[defective_x2][defective_y2] = counter
        matrix[defective_x3][defective_y3] = counter
        counter = counter + 1
End

TileMatrix(n/2, matrix, x, y, defective_x1, defective_y1)
TileMatrix(n/2, matrix, x, (y+n/2), defective_x2, defective_y2)
TileMatrix(n/2, matrix, (x+n/2), y, defective_x3, defective_y3)
TileMatrix(n/2, matrix, (x+n/2), (y+n/2), defective_x4, defective_y4)

return
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Running Time: Since in each step of the function call we are dividing the matrix into 4 sub-matrices of size n/2 (it may seem at first that we should be dividing 4 matrices into sizes of n/4, but we are dealing with areas not lengths) and we are doing constant work in our conquer step (covering the tiles can be done in constant time since we know the indices), therefore our recurrence relation turns out to be:

$$T(n) = 4T(n/2) + C$$

Which gives us a time complexity of $O(n^2)$ by Master Theorem.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

2) Answer to Question 2:

Input: We are given a set L of N line segments of which one end lies on the line y=0 and the other end lies on the line y=1. For identifying the lines, we are given the x-coordinates of both the ends of the line segments (x and x'). We are given the input in the form of a n*2 matrix where each row represents a line segment. Each row has 2 columns which represent the x (x coordinate of one end at y=0) and x'(x coordinate of other end at y=1) coordinate of the line segment.

We will be using the dynamic programming approach to solve this problem.

Preprocessing Step: Before applying the algorithm, first we need to sort the matrix in descending order according to any of the one coordinates (x or x'). Say, we chose x'. Then we have to sort the input matrix in descending (decreasing order) order according to the x' coordinate of each line segment.

Algorithm: After sorting the matrix in accordance with the x' coordinates, we get a corresponding sequence of x coordinates.
On this sequence of x coordinates, we will apply the Longest Increasing Subsequence algorithm.

Longest Increasing Subsequence Algorithm: In this problem, we have to find the longest increasing subsequence (not necessarily contiguous) from the array (the array consisting of the x coordinates).

Subproblems:
Given an array $X_i = (x_0, x_1, \ldots, x_i)$, $LS(X_i)$ is a problem which represents the length of the longest increasing subsequence of X_i which contains x_i as the last/max element of the sequence or in other words, which ends at x_i.
In terms of the lines, the subproblem $LS(X_i)$ represents the longest subset of intersecting lines which ends at the line with the x coordinates of bottom end as x_i.

Recurrence: $L[i] = \{L[i] \text{ if } x_i <= x_j ,$
$\max(L[i], L[j]+1) \text{ if } x_i > x_j\}$

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Cases: Given a sequence of integers, (x_0, x_1, …, x_{n-1}) there can be 2 cases:
Case 1: Either x_{n-1} does not belong to the longest increasing subsequence -
In this case the longest increasing subsequence for the entire array (x_0, x_1, …, x_{n-1}) will also be the longest increasing subsequence for the array (x_0, x_1, …, x_{n-1}) as x_{n-1} does not exist in the solution.

Case 2: Or x_{n-1} belongs to the longest increasing subsequence

Algorithm using Tabular Form Explanation: To solve this problem using tabular form we have to use our subproblem, i.e., we know that our subproblem is LS(X_i) which has been explained in the subproblem section.
Now, we will define a 1D array L[] of size n where the i^{th} index i.e., L[i] represents the length of the longest increasing subsequence ending at the element x_i.
The array L can be initialised with all 1's as a sequence of one element can also be considered as an increasing subsequence and each element of the array is a subsequence of length 1.
Now we use 2 loops, one outer loop with counter as i and one inner loop with counter as j. Then, for each index i starting from 1 in the array, we will iterate through all the elements before the index i, i.e., from j=0 to j<i. For each iteration of j, if the i^{th} element of L is greater than the j^{th} element (condition for increasing order), then we will update L[i] as the maximum of L[i] and L[j]+1 or else we will leave L[i] as unchanged.
After both the loops end, we get an array where L[i] represents the length of longest increasing subsequence which ends at index i or the element x_i.
To get the final solution, we find the maximum value from the array L[] and return that as the answer.

Final Solution: After running the algorithm described above, we will get a filled up array in which the i^{th} index of the array will contain the length of the longest increasing subsequence ending at the element at the i^{th} index or at the element x_i.
So, the solution to our whole problem would be the maximum of the whole array as the longest increasing subsequence would definitely end at any one of the elements, i.e., max{L}. This max value represents the length of the largest subset of lines of which every pair intersects.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Base Case: The base case here is when the size of the array is 1, In this case we have to put the value=1 in the array as for an element, the longest increasing subsequence would be the element itself and hence the length of the longest increasing subsequence would be equal to 1. Suppose, if we have only 1 line in the set, then the longest subset of lines of which every pair intersects would be the line itself and the length would be equal to 1.

Pseudocode:

```
input[n][2]=2d matrix of coordinates
Sort input according to the x coordinates(input[i][2])
Initialize an array L[] of size n
Put all the elements of input[i][1] in L
For every 0<= i<n, initialize L[i]=1.
For every i=1,2,....,n
        For every j=0,1,....,i-1
                If x_i>x_j do
                        L[i]=max{L[i], L[j]+1}
return max{L}
```

Explanation: We know that given any 2 lines, L1 and L2 with the coordinates of both the ends as $(x_1,0)$ and $(x_1',1)$ for line L1 and $(x_2,0)$ and $(x_2',1)$ for line L2.
According to the question, the y coordinates are fixed, either y=0 or y=1, hence we are dealing with only the x coordinates. So, given these 2 lines L1 and L2, in order to check whether they intersect or not, there can be 2 cases,
Case 1: If $x_1<x_2$ and $x_1'>x_2'$. In this case the bottom end of line L1 is behind the bottom end of line L2 and the top end of line L1 is ahead of the top end of line L2 and hence they intersect.
Case 2: If $x_1>x_2$ and $x_1'<x_2'$. In this case the bottom end of line L1 is ahead of the bottom end of line L2 and the top end of line L1 is behind the top end of line L2 and hence they intersect.
So, these are the 2 cases to check for checking the intersection between 2 lines.
Now, in the preprocessing step of the algorithm, we first sorted the matrix according to the x coordinates of the top ends of all the lines in descending order. After sorting, we get a corresponding sequence of the x coordinates of the bottom ends of all the lines.
As we know that the x coordinates of the top ends of the lines are arranged in decreasing order, from Case 1 we can infer that the x coordinates of the bottom ends of the lines must be an increasing sequence to satisfy the intersection condition. If the sequence is not

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

increasing, we know that the lines can't intersect as the coordinates don't satisfy any of the 2 cases written above.

Hence, we find out the length of the longest subset of intersecting lines by finding the length of the longest increasing subsequence on the sequence of the x coordinates of bottom ends of the lines obtained after sorting procedure.

Running Time: Since there is sorting algorithm applied mergesort which takes O(nlogn) time and there are two for loops essentially running from 1 to n, the time complexity turns out to be $O(n^2+nlogn)$ which is $O(n^2)$.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

3) Answer to Question 3:

Preprocessing Stage: None

Subproblem: A brute force approach would be to try out all possible combinations of shipments and choose the one with the minimum cost.
Given i weeks, dp[i][j] represents the subproblem that what is the minimum cost in the i^{th} week given that we have used Company C j times in the i^{th} and $(i-1)^{th}$ weeks. So, the value in dp[i][j] is the minimum cost uptill week i when we have used Company C, j times in the i^{th} and $(i-1)^{th}$ weeks. Each row in the i^{th} week can be filled according to the recurrence described below.
The minimum cost in the i^{th} week will simply be the minimum of {dp[0][i], dp[1][i], dp[2][i]}.

Recurrence: We have an 2-D array of dimensions 3x(N+1) where N is the number of weeks.
s_i: is the number of units to ship in the i^{th} week.
a: cost of shipping per product of Company A
b: cost of shipping per week of Company B
c: cost of shipping per product of Company C
d: discount given by Company C

$$
\begin{aligned}
dp[0][i] = \min\{ & \\
& dp[0][i-1] + as_i , \\
& dp[0][i-3] + 3b , \\
& dp[1][i-1] + as_i , \\
& dp[1][i-3] + 3b , \\
& dp[2][i-1] + as_i , \\
& dp[2][i-3] + 3b \\
& \}
\end{aligned}
$$

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

$dp[1][i] = dp[0][i-1] + cs_i - d$
$dp[2][i] = dp[0][i-2] + (cs_i - d) + (cs_{i-1} - d)$

Base Cases:

$dp[0][0] = 0$
$dp[1][0] = 0$
$dp[2][0] = 0$
$dp[0][1] = as_1$
$dp[1][1] = cs_1 - d$
$dp[2][1] = 0$
$dp[0][2] = \min\{as_1, cs_1 - d\} + as_2$
$dp[1][2] = as_1 + cs_2 - d$
$dp[2][2] = (cs_2 - d) + (cs_1 - d)$

Pseudocode:

```
//Initialize all the base cases as above
For 3 ⩽ i < N+1 do
        dp[0][i] = min{
                dp[0][i-1] + as_i ,
                dp[0][i-3] + 3b ,
                dp[1][i-1] + as_i ,
                dp[1][i-3] + 3b ,
                dp[2][i-1] + as_i ,
                dp[2][i-3] + 3b
               }

        dp[1][i] = dp[0][i-1] + cs_i - d

        dp[2][i] = dp[0][i-2] + (cs_i - d) + (cs_{i-1} - d)
End

return min{ dp[0][n], dp[1][n], dp[2][n] }
```

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

Explanation:

Let us first explain how we are taking Company B's constraints into consideration. Since Company B must be used 3 weeks consecutively (if hired) , in order to find the minimum cost in the i^{th} week, we compare, the cost in the $(i-1)^{th}$ week + cost of using Company A in the i^{th} week AND the cost in the $(i-3)^{th}$ week + cost of using Company B for 3 weeks.
If we choose the latter of the two choices (in the above sentence) then it is as if we are using Company B for the weeks (i-2), (i-1) and i , adding the cost *b* to each of these weeks. Thus, whenever Company B is chosen, it is hired for 3 consecutive weeks only.

Now let us explain how we are taking Company C's constraints into consideration. In any i^{th} week, there are three possibilities, namely,
1. We have used Company C in the current week as well as the last week.
2. We have used Company C in the current week only and not the last week.
3. We are not using Company C in the current week.

The row dp[0][i] represents the minimum cost in the i^{th} week when we don't use company C in that week.
The row dp[1][i] represents the minimum cost in the i^{th} week when we use company C in that week.
The row dp[2][i] represents the minimum cost in the i^{th} week when we use company C in the i^{th} week as well as in the $(i-1)^{th}$ week (thereby reaching the limit to how many times company C can be used consecutively).

Let us try to understand how the code works.
The base cases are fairly simple and can be understood easily by inspection.
The dp[0][i] = min{} condition first finds the cost from the $(i-1)^{th}$ week and $(i-3)^{th}$ week from all rows of the dp matrix. These costs are the minimum costs in the $(i-1)^{th}$ week and $(i-3)^{th}$ week when we have used Company C none, once and twice.

Theory Assignment-1

Roll No: 2021055 **Section: A**

Name: Jyotirmaya Singh **Specialisation: CSE**

It then adds the cost of Company A or Company B (in the i^{th} week) as per the recurrence

relation and takes the minimum cost out of them all. So clearly in the i^{th} week we have not used Company C, BUT we have taken into account all the previous weeks in which we may have used Company C (either once or twice or zero times) due to the min{} condition.
Then dp[1][i] uses Company C in the i^{th} week and takes the value from dp[0][i-1]. Since dp[0][i-1] has taken into account the cost all the previous weeks which may or may not have used Company C, therefore dp[1][i] has also taken into account all the previous weeks in which we may or may not have used Company C.
Similarly dp[2][i] uses Company C in the i^{th} and $(i-1)^{th}$ week and takes the value from dp[0][i-2]. The same argument made above for dp[1][i] holds for how dp[2][i] has taken all optimum possibilities into account.

To put it simply, the recursive relation described is exhaustive and is calculating the most optimum cost for each week.

The final answer is min{ dp[0][n], dp[1][n], dp[2][n] }.

Running Time: Since there are two for loops essentially running from 0 to n (N), and 0 to 3, the time complexity turns out to be O(3n) = O(n).