Theory Assignment-1

Section:B

Roll No:2021482
Name:PARAS DHIMAN
Specialization:CSB

Roll No:2021304
Name:ADITYA DAIPURIA
Specialization:CSD

Ques1:

we can use divide and conquer approach to solve the problem :-

(a) No Preprocessing Stage is Required.

(b) Division Of SubProblem:

We can divide the board problem into 4 subproblems by taking a board size of (n/2) X (n/2). Then, for the tiling of the boards, we can start with the base case of a 2 X 2 board which can be filled using a single L-shaped tile.

(c) Combining the Solutions:

To combine the solutions, we first need to find the position of the defective square. Based on the location of the defective square, we will place the L-shaped tiles accordingly. If the defective square is in the top-left, we will place one L-shaped tile in the bottom-left and two in the top-right. Similarly, if the defective square is in the top-right, we will place one L-shaped tile in the bottom-right and two in the top-left. If the defective square is in the bottom-left, we will place one L-shaped tile in the top-left and two in the bottom-right. For a defective square in the bottom-right, we will place one L-shaped tile in the top-right and the other two in the bottom-left .

(d) PseudoCode:-

Base case:- At n==1 ,one square left can't put any L-shaped tile there.

Rest explained through the pseudocode:

```
PutTile(n):
        if n==1:
                    Nothing to be put anywhere
return
        If n==2:
                    We will place a single l-shaped
tile in the remaining square
                    return

        We will firstly find out the sub-case which
will contain the defective Square

        #Our sub-cases will be as folows

        PutTile(n/2)
        PutTile(n/2)
        PutTile(n/2)
```

```
      PutTile(n/2)

      We will place a single tile in the mid of the
central board/sub-board(i.e in a case    where we
divided it)

      Then we will fill up the remaing space with
the Lshaped tiles in same order as done for the the
sub problems

PutTile(n)
```

(e)Time Complexity of the Algorithm:

Let the time taken to for (n X n) be T(n)

In the code we are making 4 sub-Prolems/sub-boards

Time taken by them will be: 4T(n/2)

For the combination of subproblem it takes constant Time.So we get the equation as:

$$T(n) = 4T(n/2) + O(1)$$

Master Method:

$$T(n) = 4T(n/b) + f(n)$$

3cases:-

- If f(n) = O(nc) where c < Logba then T(n) = Θ(nLogba)
- If f(n) = Θ(nc) where c = Logba then T(n) = Θ(ncLog n)
- If f(n) = Ω(nc) where c > Logba then T(n) = Θ(f(n))

We have

T(n) = 4T(n/2) + O(1)

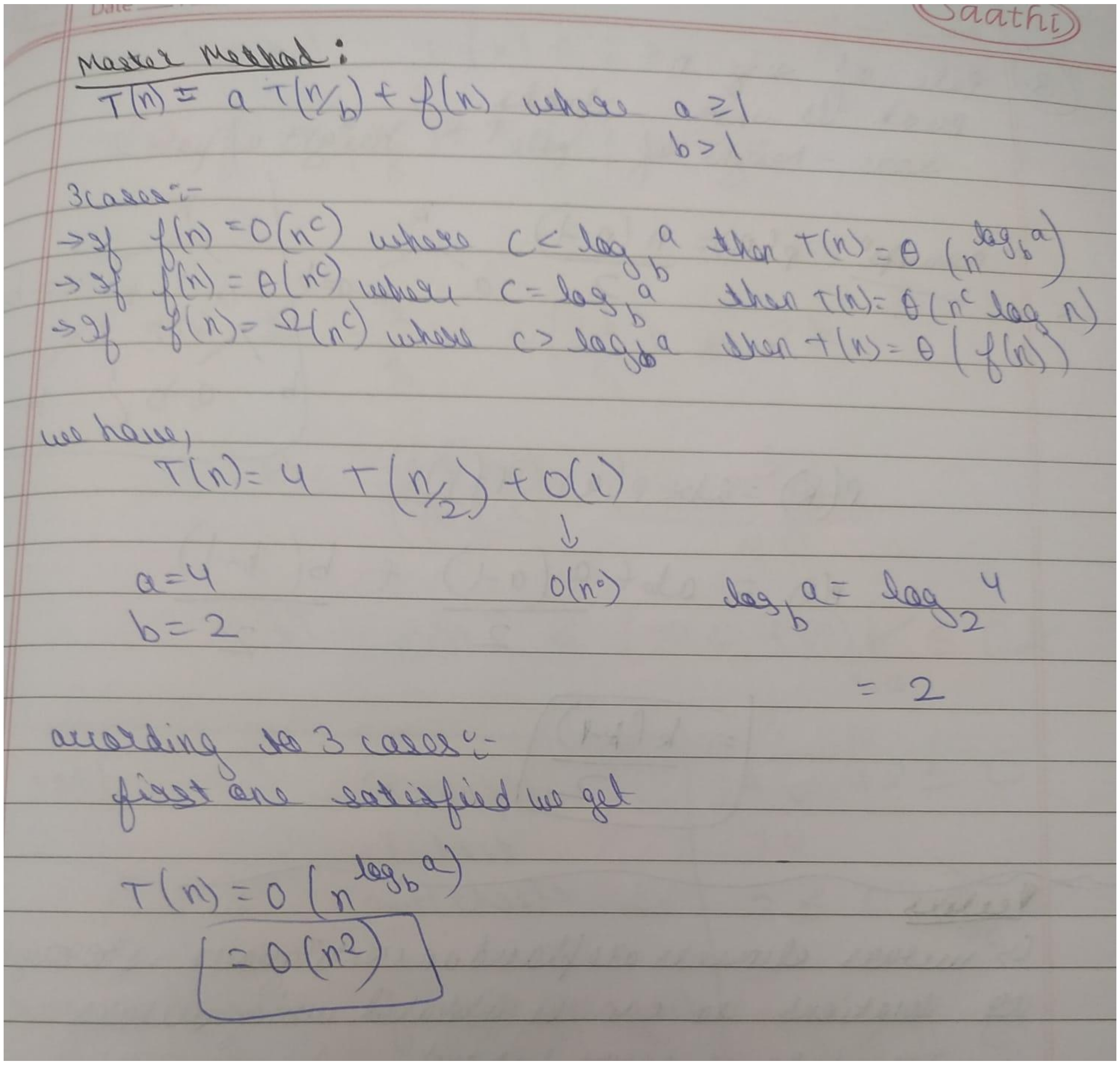
a = 4

b = 2

According to all the three cases:-

T(n) = O(nLogba)

O(n^2)



Master Method:

$T(n) = a\,T(n/b) + f(n)$ where $a \geq 1$

$b > 1$

3 cases:-

→ If $f(n) = O(n^c)$ where $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$

→ If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$

→ If $f(n) = \Omega(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$

we have,

$$T(n) = 4\,T(n/2) + O(1)$$

$$\downarrow$$

$a = 4$ $\quad O(n^0)$ $\quad \log_b a = \log_2 4$

$b = 2$ $\quad = 2$

according to 3 cases:-

first one satisfied we get

$$T(n) = O(n^{\log_b a})$$

$$\boxed{= O(n^2)}$$

Using Master's Theorm we get the time complexity to be O(n^2).

References:

1.https://www.geeksforgeeks.org/how-to-analyse-complexity-of-recurrence-relation/

2.https://www.geeksforgeeks.org/tiling-problem-using-divide-and-conquer-algorithm/
3.https://www.chegg.com/homework-help/questions-and-answers/read-instructions-carefully-answer-attached-question-clearly-solution-uses-dynamic-program-q108603001

Ques2:

we can use dynamic Programming approach to solve the problem :-

(a) Preprocessing Stage: Sort the line segments according to their maximum endpoint on the line y=0.

(b) Division Of SubProblem:

Let there be the largest subset of the line segments int the array LS[i] such that the line segments comes under this intersect with each other.And the maximum endpoint we get is the ith one in the largest subset arrays.

(c) Recurrence:

For the calculation of the subsets in LS .We are going to have two recurrence relations where if we use L[] containing the line segments having the maximum endpoint on y=0..

We can recursively define LS[i]:

The below condition is applied if no intersection took place with the previous one

LS[i] = L[i]

The Second condition written below is applied when there is k largest index less than the i and intersecting with the line L[i]

LS[i] = LS[k] + L[i]

(d) Subproblem Solving the original Problem:

AS we know that LS[i] is the array6 containg the largest subset of the line segments from 1 to n where all line segments intersects and such that the last segment here i will intersect with line segment k (k < i).

The subproblem we get as LS[i] which is required to be solved.

(e) PseudoCode:

```
Here n--> has been supposed as the length for the given
array L containg the line segements

Here LS will e a 2D matrix that will store every
possible subset
```

```
SubsetFind(L,n):
        Sorting the L superset according to there
highest coordinate on the line y=0.

        LS[i] = L[0];
        For i having values (1,....,n-1):
                    Lsegment = L[i]
                    k = -1

                    for j = i - 1 to 0:
                                if L[j] intersect
Lsegment:
                                            k = j
                                            break
        if k == -1:
                    we will put Lsegment in the
subsets array i.e. LS[]
        otherwise:
                     we will put LS[k] + Lsegment in
the subsets array i.e LS[]

        Assigning the set initially for the
comparision and comparing them
        setIni = LS[0]
        For i = 1 till n-1:
                    If len(LS[i]) > len(setIni):
                                setIni = LS[i]

        return setIni
```

(f) The time complexity of the algorithm we get as –

For the sorting step which takes O(n log n) time

For searching for the line segment ,we in turn first search for the largest index k that intersects the line k<I ---O(n^2)

Iterating over all the computed subsets in LS[] this steps also take O(n^2)

So as we know the sorting step takes O(n logn) time and the total number of operations we perform O(n^2)

Therefore the time Complexity we get comes out to be is O(n^2) as O(n^2) dominates the O(n logn).

Hence Proven the time complexity which comes out to be equal to O(n^2)

Justification:

1.)Correctness of Subproblem solution:

For every iteration I, algo finds the largest subset having the maximum endpoint at y = 0

Such that this has been finded out from L[I] with L[i] and the largest subset that will be including l[i] will surely include LS[k} as well.

2.)Optimal Structure:

Optimal solution at an index I is used as we get the maximum endpoint at y=0 where L[k] is the largest segment that will intersect with L[k]. So the largest subset that include L[i+1] get the optimal Siolution.

3.) Optimal Solution to the original Problem:

As we have included the line segment those which intersect with each other.

Hence it is satisfying the optimal Substructure and overlapping subproblem properties and computing the optimal Solution for the largest subset very well.

References:

1.) https://www.geeksforgeeks.org/given-a-set-of-line-segments-find-if-any-two-segments-intersect/

2.) https://www.chegg.com/homework-help/questions-and-answers/suppose-given-set-l-n-line-segments-plane-segment-one-endpoint-line-y-0-one-endpoint-line--q67225730

Ques3:

we can use dynamic Programming approach to solve the problem :-

(a) Preprocessing Stage:

No preprocessing stage is required.

(b) Division Of SubProblem:

We have divided the problem into subproblem based on the week number i and the company used in the p[revious week c.We can represent the subproblem as f(i,c) which gives us the minimum cost of shipping the units produced in week1 to week i, ansd where the company was used in the week i - 1.

(c) Recurrence:

Let dp[i][c] be the minimum cost of shipping i units in the first week using one of the three shipping companies A,B, and C where c is the number of consecutive weeks company c has used until the jth week.

The recurrence relation can be set as follows:-

If i = 1 || i = 2:

If c =2, then dp[i][c] = min(a*s[i] + f(i-1,0))

Else, dp[i][c] = min(a * s[i] + f(i - 1,0),f(i-1,c+1) + c*s[i]-d)

If c = 2:dp[i][c] = min (a*s[i] + f(i-1,0),f(i-3,0) + 3b)

Else, dp[i][c] = min (f(i-1,0) + a*s[i],f(i-3,0) + 3b, f(i-1,c+1) + c*s[i] - d)

(d) Subproblem Solving the original Problem:

To solve the original problem of finding the minimum cost schedule for shipping all units, we need to solve the subproblems first. For this, we start by solving the subproblem for shipping the units produced in the last week.

Then, for each preceding week, we can use the result of the subproblem for the following week and add the cost of shipping the units produced in that week using one of the three shipping agents.

We repeat this process for all preceding weeks, and at the end, we obtain the solution to the original problem by using the result of the subproblem for shipping all units.

Specifically, for the last week, we can simply use the cost of shipping all units produced in that week using one of the three shipping agents, i.e., either A, B, or C, whichever is the minimum.

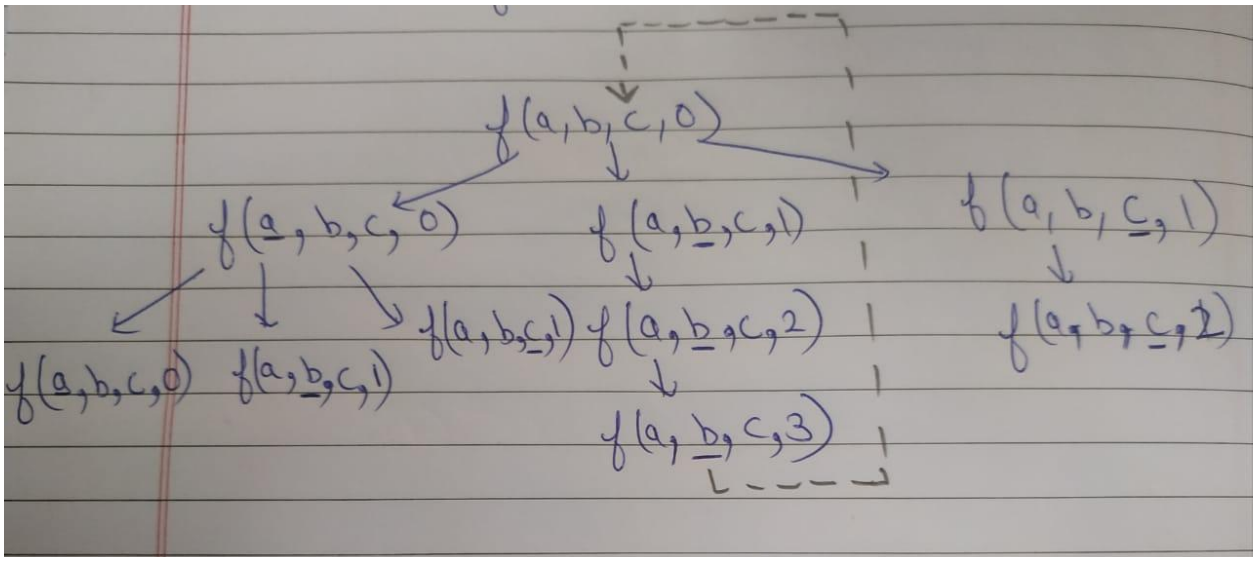
For each preceding week i, we need to consider three cases:

If we use shipping agent A, the cost would be the cost of shipping the units produced in that week using A plus the result of the subproblem for the previous week (i.e., f(i-1)).

If we use shipping agent B, the cost would be the cost of shipping the units produced in the last three weeks using B plus the result of the subproblem for the week before the last three weeks (i.e., f(i-3)).

If we use shipping agent C, the cost would be the cost of shipping the units produced in that week using C minus the reward offered by C plus the result of the subproblem for the week before the last two weeks (i.e., f(i-2)).

We can choose the minimum of the above three cases to obtain the result of the subproblem for week i, i.e., f(i). Finally, the minimum cost schedule for shipping all units can be obtained by using the result of the subproblem for the last week, i.e., f(n).



(e) PseudoCode:

```
A charge → a/unit
B charge → b/unit but three consecutive
```

```
C charge → c/unit - d/week but 2 consecutive

Now let i be units and j be weeks

A → dynamic[i - 1][j - si] + A.si
B → dynamic[i - 3][j] + B1
C → dynamic[i - 2][j - si] + csi - d

Taking the min from al of the above and checking it
Now i can be for n unit and
Weeks j are (1,s1,s2,.....,sn)

Let the dynamic array dp[n + 1][2]
Initialising
Dp[n +1][2] = {-1}

Now we get the function to be as:
f(i,c)
{
    if(dp[i][c] != 0)
    {
        Return dp[i][c];
    }
    if(i == 1 || i == 2)
    {
        if(c == 2)
        {
            Return dp[i][c] = min(a*s[i] +
f(i-1,0))
```

```
        }
        Return dp[i][c] = min(a * s[i] + f(i -
1,0),f(i-1,c+1) + c*s[i]-d)
    }
If (c == 2)
{
    return dp[i][c] = min (a*s[i] +
f(i-1,0),f(i-3,0) + 3b)
}
Return dp[i][c] = min (f(i-1,0) + a*s[i],f(i-3,0) +
3b, f(i-1,c+1) + c*s[i] - d)
}
```

(f) The time complexity of the algorithm we get as –

The total complexity of the algorithm that we get is (2n) i.e. O(n)

As there will be two for loops running for n times that

When summed up is 2n

After taking the Big o notation of it we get the order to be as O(n)

The stack space taken up is also (n + 2) i.e. O(n)

Justification for the correctness:

The correctness of the algorithm can be justified by showing that it satisfies the properties of optimal substructure and overlapping subproblems, which are the two fundamental properties of dynamic programming.

1.) Optimal Substructure:

    The problem exhibits optimal substructure because an optimal solution to the problem can be constructed from optimal solutions to its subproblems. Let dp[i][c] denote the minimum cost of shipping i units in the first j weeks, where the i-th week is shipped by company C consecutively for c weeks. The final solution to the problem is given by dp[n][0], which represents the minimum cost of shipping n units in the first n weeks without any consecutive shipping by company C.

    The optimal substructure is apparent from the recurrence relation used to compute dp[i][c]. The recurrence relation considers all possible ways of shipping the i-th week, and computes the minimum cost for each of these options. Since the minimum cost for a particular i and c is computed using the minimum cost for the previous i and c, the optimal solution for the original problem can be constructed from optimal solutions to its subproblems.

2.) Overlapping Subproblems:

    The problem exhibits overlapping subproblems because the same subproblem is often encountered multiple times during the computation of the final solution. For example, computing dp[i][c] requires computing dp[i-1][c+1] and dp[i-2][c], which are subproblems that have already been computed.

To avoid redundant computation, we can use dynamic programming to store the solutions to these subproblems in a table. The table is filled in a bottom-up manner, starting with the base cases (i=1, i=2, and i=3), and moving on to the general cases (i>3). As a result, the solutions to subproblems that are encountered multiple times are reused from the table, rather than being recomputed.

Therefore, the algorithm satisfies the properties of optimal substructure and overlapping subproblems, which justifies its correctness

References:

1.) https://drive.google.com/file/d/168D3maL2g4d6y0I91bPEqmAbgZ89DxS7/view?usp=sharing