ADA Theory Assignment - 1

Aakarsh Jain
Roll Number - 2021507
IIIT - Delhi
aakarsh21507@iiitd.ac.in

Siddhant Rai Viksit
Roll Number - 2021565
IIIT - Delhi
siddhant21565@iiitd.ac.in

February 21, 2023

1 Question 1

1.1 Problem

Consider the problem of putting L-shaped tiles (L-shaped consisting of three squares) in an n × n square-board. You can assume that n is a power of 2. Suppose that one square of this board is defective and tiles cannot be put in that square. Also, two L-shaped tiles cannot intersect each other. Describe an algorithm that computes a proper tiling of the board. Justify the running time of your algorithm.

1.2 Approach

We will divide the board of size $n \times n$ into 4 smaller boards of size $\frac{n}{2} \times \frac{n}{2}$ each. We can then place tiles on each 4 boards recursively, leaving out exactly 1 square on each of the smaller boards un-tiled. For the smaller board containing defective square, we will leave that square un-tiled, and for remaining 3 boards, we will leave the corner squares un-tiled so that we can place a tile there during the combination phase of the algorithm.

1.3 Pre-Processing

We will create a $n \times n$ board using nested arrays to store the tiled squares. Thus, we initialize such a board as $A[n][n]$ which will be filled with a valid tiling. To indicate un-tiled squares, we will use -1. Thus, A is initialized with -1.

1.4 Subproblem

We define a routine BOARD TILING where we pass the the board to tile and the square to be left un-tiled. Let n be the size of board, then we can represent the squares on the board using Cartesian Coordinate System as (x, y), where $1 \leq x, y \leq n$.

If $n > 2$, the routine BOARD TILING will call itself recursively 4 times. If the defective square belongs to that sub-board, we will pass the coordinates of the defective square, otherwise we will pass the coordinates to the center-most square w.r.t. to the larger board to the recursive call.

If $n = 2$, the routine BOARD TILING will place a tile on 3 of its squares, leaving the passed square empty

To indicate the portion of board to tile for every call to BOARD TILING, we also pass the coordinates of top-left and bottom-right corner of the board.

We will also define a subroutine TILE where we pass the coordinates of the 3 squares we need to place a tile on.

1.5 Combine step

Now, we need to place another tile to cover 3 of the remaining un-tiled 4 squares. To achieve this, we will check the resultant 4 central squares. If exactly 3 of the squares are un-tiled, we can simply tile them. If all 4 of the squares are un-tiled, it means that one the central squares are defective and we can tile the remaining 3

squares other than the square passed in the subroutine call for BOARD TILING. We shall do this using the COMBINE TILE routine.

Thus we will call the COMBINE TILE routine for every routine call of BOARD TILING when $n > 2$.

1.6 Pseudo-Code

```
1: procedure TILE(A[1..n][1..n], (x_1, y_1), (x_2, y_2), (x_3, y_3))
2:     Generate random unique positive integer id
3:     A[x_3][y_3] ← A[x_2][y_2] ← A[x_1][y_1] ← id
4:     return A
5: end procedure
6: procedure COMBINETILING(A[1..n][1..n], (x_d, y_d), (x_min, y_min), (x_max, y_max))
7:     x_mid ← (x_max + x_min − 1)/2
8:     y_mid ← (y_max + y_min − 1)/2
9:     if (x_d, y_d) = (x_mid, y_mid) or A[x_mid][y_mid] > 0 then
10:        A ← TILE(A, (x_mid, y_mid + 1), (x_mid + 1, y_mid), (x_mid + 1, y_mid + 1))
11:    else if (x_d, y_d) = (x_mid + 1, y_mid) or A[x_mid + 1][y_mid] > 0 then
12:        A ← TILE(A, (x_mid, y_mid), (x_mid, y_mid + 1), (x_mid + 1, y_mid + 1))
13:    else if (x_d, y_d) = (x_mid, y_mid + 1) or A[x_mid][y_mid + 1] > 0 then
14:        A ← TILE(A, (x_mid, y_mid), (x_mid + 1, y_mid), (x_mid + 1, y_mid + 1))
15:    else
16:        A ← TILE(A, (x_mid, y_mid), (x_mid + 1, y_mid), (x_mid, y_mid + 1))
17:    end if
18:    return A
19: end procedure
20: procedure BOARDTILING(A[1..n][1..n], (x_d, y_d), (x_min, y_min), (x_max, y_max))
21:    x_mid ← (x_max + x_min − 1)/2
22:    y_mid ← (y_max + y_min − 1)/2
23:    if x_mid = x_min then                                          ▷ Base Case
24:        A ← COMBINETILING(A, (x_d, y_d), (x_min, y_min), (x_max, y_max))
25:        return A
26:    end if
27:    if x_d ≤ x_mid & y_d ≤ y_mid then                              ▷ Call for upper left sub-square
28:        A ← BOARDTILING(A, (x_d, y_d), (x_min, y_min), (x_mid, y_mid))
29:    else
30:        A ← BOARDTILING(A, (x_mid, y_mid), (x_min, y_min), (x_mid, y_mid))
31:    end if
32:    if x_d > x_mid & y_d ≤ y_mid then                              ▷ Call for upper right sub-square
33:        A ← BOARDTILING(A, (x_d, y_d), (x_mid + 1, y_min), (x_max, y_mid))
34:    else
35:        A ← BOARDTILING(A, (x_mid + 1, y_mid), (x_mid + 1, y_min), (x_max, y_mid))
36:    end if
37:    ...                ▷ Similarly add recursive calls for bottom left and bottom right sub-square
38:    A ← COMBINETILING(A, (x_d, y_d), (x_min, y_min), (x_max, y_max))
39:    return A
40: end procedure
41: BOARDTILING(A[1..n][1..n], (x_d, y_d), (1, 1), (n, n))
```

1.7 Time Complexity

We can split the board into 4 sub-boards in constant time. The COMBINE TILE routine can also be executed in constant time.

Thus for every call of BOARD TILING, we will make 4 recursive calls of same routine and our splitting and

merging cost is n^2. Thus the final recurrence relation is

$$T(n) = 4T(\frac{n}{2}) + O(1)$$

using Master Theorem for case $log_b(a) > c$, the runtime complexity of algorithm is $O(n^2)$

Also, since we are only storing one $n \times n$ array to represent the board and passing the same by reference, the space complexity of the algorithm is also $O(n^2)$

2 Question 2

2.1 Problem

Suppose we are given a set L of n line segments in 2D plane. Each line segment has one endpoint on the line y = 0, one endpoint on the line y = 1 and all the 2n points are distinct. Give an algorithm that uses dynamic programming and computes a largest subset of L of which every pair of segments intersects each other. You must also give a justification why your algorithm works correctly.

2.2 Approach

Given a pair of wires (a_i, b_i) and (a_j, b_j), they will intersect when either $a_i < a_j, b_i > b_j$, or $a_i > a_j, b_i < b_j$. Therefore, if we sort the array of wires according to a_i $(i < j \leftrightarrow a_i < a_j)$, we need to count the number of pairs (i, j) s.t $i < j, b_i > b_j$. We are interested in the biggest subset such that the number of pairs in that subset = $\binom{s}{2}$, where s=size of subset. This is only going to be true for a subsequence s where $b_i > b_j \forall 1 \leq i, j \leq |s|, i < j$, i.e a strictly decreasing subsequence. To compute that we can use dynamic programming.

2.3 Pre-Processing

We shall define a subroutine binarySearch, which when given an array a that is strictly decreasing and a number x, returns the smallest index i s.t $a_i < x$.

2.4 Subproblem

We shall define our state dp_i as the maximal number x s.t we can make a decreasing subsequence of length i where the last element is x.

2.5 Iterative transitions

As we iterate over b, the length of the longest decreasing subsequence possible ending on b_i is $max(\{x|dp_x > b_i\}) + 1$. After that we can update $dp_{x+1} = max(dp_{x+1}, b_i)$, and the final answer is $max(\{x|\exists dp_x\})$

2.6 Pseudo-Code

```
1:  procedure BINARYSEARCH(a[1..n],x)
2:      l ← 1
3:      r ← n
4:
5:      while l < r do
6:          m ← (l+r)/2
7:
8:          if a_i < x then
9:              r ← m
10:
11:         else if a_i ≥ x then
12:             l ← m + 1
13:         end if
14:     end while
```

```
    return l
end procedure
procedure LargestSubset(Wires[1..N])
    WIRES ← sort(WIRES)
    dp[N + 1] ← {∞, −∞, ..., −∞}
    ans ← 0

    for k = 1 to n do
        i ← binarySearch(dp, b_k)
        dp[i] ← max(dp[i], b_k)
        ans ← max(ans, i)
    end for
    return ans
end procedure
```

2.7 Time Complexity

Sorting the array will be $O(nlogn)$. Additionally, we are going through the array once and each time performing $BINARYSEARCH$ which is $O(logn)$, which $O(n) \times O(logn) = O(nlogn)$. Thus we have a final time complexity of $O(nlogn)$

3 Question - 3

3.1 Problem

Suppose that an equipment manufacturing company manufactures s_i units in the i-th week. Each week's production has to be shipped by the end of that week. Every week, one of the three shipping agents A, B and C are involved in shipping that week's production and they charge in the following:

- Company A charges a rupees per unit.
- Company B charges b rupees per week (irrespective of the number of units), but will only ship for a block of 3 consecutive weeks.
- Company C charges c rupees per unit but returns a reward of d rupees per week, but will not ship for a block of more than 2 consecutive weeks. It means that if s_i unit is shipped in the i^{th} week through company C, then the cost for i^{th} week will be $cs_i - d$.

The total cost of the schedule is the total cost to be paid to the agents. If s_i unit is produced in the i^{th} week, then s_i unit has to be shipped in the i^{th} week. Then, give an efficient algorithm that computes a schedule of minimum cost. (Hint: use dynamic programming)

3.2 Approach

For this, we can use a dynamic programming approach. We are interested in 2 parameters, the current week and the shipping company hired for the particular week. Thus, we can use a Dynamic Programming routine to iterate over all possible combinations in an efficient manner. We can then find the minimum cost of hiring every company for every week, while keeping in mind the provided constraints.

3.3 Pre-Processing

We need a DP Table to store the values of calculated subproblems in order to implement memoization.

Thus, we will initialize an array *memo* of dimensions $N \times 3$ and initialize with -1 to indicate that the states are yet to be calculated.

3.4 Subproblem

Our subproblem will have 2 parameters, the current week we are iterating upon and the company hired for the week. Thus we will represent the subproblems as $dp_{i,j}$ where $i \in [1, n]$ and $j \in A, B, C$

Hence, to find the minimum cost of shipping for n weeks, we will have to find $dp_{N,A}, dp_{N,B}$ & $dp_{N,C}$. The final answer will be $min(dp_{N,A}, dp_{N,B}, dp_{N,C})$

3.5 Recurrence Relation

Now, by using the constraints provided in the problem statement, we can write the following recurrence relation:-

$$dp_{i,A} = \left\{ \begin{array}{ll} s_i \times a & \text{if } i = 1 \\ min(dp_{i-1,A}, dp_{i-1,B}, dp_{i-1,C}) + s_i \cdot a & \text{if } i \geq 1 \end{array} \right\} \tag{1}$$

$$dp_{i,B} = \left\{ \begin{array}{ll} \infty & \text{if } i \leq 2 \\ 3b & \text{if } i = 3 \\ min(dp_{i-3,A}, dp_{i-3,B}, dp_{i-3,C}) + 3b & \text{if } i \geq 4 \end{array} \right\} \tag{2}$$

$$dp_{i,C} = \left\{ \begin{array}{ll} s_i \times c - d & \text{if } i = 1 \\ min(dp_{i-1,A}, dp_{i-1,B}, dp_{i-1,C}) + s_i \times c - d & \text{if } i = 2 \\ min(min(dp_{i-2,A}, dp_{i-2,B}) + (s_{i-1} + s_i) \cdot c - 2d, min(dp_{i-1,A}, dp_{i-1,B}) + s_i \cdot c - d) & \text{if } i \geq 3 \end{array} \right\} \tag{3}$$

As we always have to ship with company B in 3 consecutive weeks, we will only calculate the cost of shipping for all 3 weeks in the 3^{rd} week. Thus, $dp_{1,B}$ and $dp_{2,B}$ are ∞, even though we may ship using company B in 1^{st} and 2^{nd} week.

Similarly, for company C, since we can ship for maximum 2 consecutive weeks, we will compute 2 possibilites, that we are only shipping using the company for previous week, and for current and previous week.

3.6 Pseudo-Code

```
1: procedure DP(i,j)
2:     if memo[i][j] ≠ −1 then
3:         return memo[i][j]
4:     end if
5:     if j = A then
6:         if i = 1 then
7:             return memo[i][j] ← a · s_1
8:         else
9:             return memo[i][j] ← min(DP(i − 1, A), DP(i − 1, B), DP(i − 1, C)) + a · s_1
10:        end if
11:    else if j = B then
12:        if i ≤ 2 then
13:            return ∞
14:        else if i = 3 then
15:            return memo[i][j] ← 3 · b
16:        else
17:            return memo[i][j] ← min(DP(i − 3, A), DP(i − 3, B), DP(i − 3, C)) + 3b
18:        end if
19:    else if j = C then
20:        if i = 1 then
21:            return memo[i][j] ← c · s_1 − d
22:        else if i = 2 then
23:            return memo[i][j] ← min(DP(i − 1, A), DP(i − 1, B)) + s_i · c − d
24:        else
25:            return memo[i][j] ← min(min(DP(i−2, A), DP(i−2, B))+c(s_i+s_{i−1})−2d, min(DP(i−1, A), DP(i−
    1, B)) + cs_i − d)
```

```
        end if
    end if
end procedure
```

3.7 Time Complexity

Since we need to calculate $3 \times N$ subproblems, and each subproblem requires a constant time to compute, the runtime complexity of the above algorithm will be $O(3\dot{N}) = O(N)$

Further, since we are also storing all the states, the space complexity is also same, i.e. $O(N)$. But it can be improved to $O(1)$ if we implement this question using an iterative approach as we only need to remember values of DP Table for previous 3 states.