COT 5405 ANALYSIS OF ALGORITHMS

PROGRAMMING ASSIGNMENT 2
DYNAMIC PROGRAMMING

Spring 2023

Group No: 60	
Team Members	
Vedant Patil	3948-5964
Shivani Patil	5354-9503

Table of Content

Problem Definition:

Consider a town named Greenvale that has a grid layout of m × n plots of land. Associated with each plot (i, j) where i = 1, . . . , m and j = 1, . . . , n, Greenvale's Parks and Recreation Department assigns a non-negative number p[i, j] indicating the minimum number of trees that must be planted on that plot. A local environmental group named GreenHands is interested in finding the largest possible square shaped area of plots within the town that requires a minimum of h trees to be planted on each plot individually. More formally, problems are stated below, where Problem1 is a special case of Problem2 and Problem3.

We have discussed all the given 3 problems, their design in required time complexity and implementation in the below sections, followed by comparative study of each of the implementation.

Steps to run:

1. Go to Source Folder
2. Execute "make run#" command to run particular strategy.
   e.g. "make run1" to execute TASK1
   "make run5A" to execute TASK 5A

DESIGN AND ANALYSIS OF ALGORITHMS

PROBLEM 1

Given a matrix p of m × n integers (non-negative) representing the minimum number of trees that must be planted on each plot and an integer h (positive), find the bounding indices of a square area where each plot enclosed requires a minimum of h trees to be planted

##ALG1 TASK 1 – Brute Force:

1.1 DESIGN

1. The variables start_row, start_col, end_row, and end_col determine the beginning and ending points of the sub-square that is being examined.

2. To implement the Brute force approach, we will use a nested loop to traverse the matrix p, where each element p[i,j] represents the minimum number of trees that must be planted on that plot. When we come across a plot with p[i,j]>=h, we will mark that plot as the starting point of our square area.

3. Next, we will use a while loop to traverse all the squares that start from the plot identified above until we reach the end of the matrix (end_row < m and end_col < n).

4. Within the while loop, we will use two nested loops for i and j, ranging from start_row to end_row and start_col to end_col respectively, to traverse through the square area.

5. For each plot within the square area, we will check whether p[i,j] < h and set a flag to False if it is true.

6. If the flag is True, which means we didn't come across any plot with p[i,j] < h, we will calculate the area of the sub-square and compare it with the maximum area obtained so far. We will update the maximum area accordingly.

7. End while loop, when each sub-square is considered. Return the bounding upper left corner and right bottom right corner of the max square area found. The time complexity of this brute force approach is $O(m^3n^3)$.

Taking these pointers in mind we devise the following algorithm

1.2 PSEUDO CODE

TASK1 (m, n, h, matrix)

```
res ← [0,0,0,0]

FOR start_row = 1 to m:
        FOR start_col = 1 to n:
                IF (matrix[start_row] [start_col] >= h) THEN
                        end_row ← start_row
                        end_col ← start_col
                        WHILE (end_row <= m and end_col <= n )
                                flag ← True
                                FOR i = start_row to end_row:
                                        FOR j = start_col to end_col:
                                                IF (matrix[i][j] >= h) THEN CONTINUE
                                                ELSE
                                                        flag ← False
                                IF (flag == True) THEN
                                        temp ← [start_row, start_col, end_row, end_col]
                                        temp_area ← (end_row-start_row) * (end_col-start_col)
                                        max_area ← (res[2] – res[0]) * (res[3] – res[1])
                                        IF (max_area < temp_area) THEN
                                                res ← temp
                                end_row ← end_row + 1
                                end_col ← end_col +1
                        END WHILE
RETURN res
```

1.3 ANALYSIS

Proof of Correctness

Loop Invariant:

At the start of each iteration **res** contains the upper left corner and bottom right corner of the max square area obtained till now.

Initialization:

For every iteration, first start_row and start_col are chosen, after that end_row and end_col are initialized to start_row and start_col only if matrix[start_row][start_col]>=h.

Maintenance:

At every step, we are checking each plot, enclosed within a square defined by start_row, start_col, end_row, and end_col. Also, we are maintaining and updating res which contains the upper left and bottom right corner of the max square area till now

Termination:

The program will terminate when start_row = m or start_col = i.e., when all possible elements are considered to be the upper left corner of the largest square area. "res" will store top left and bottom right corners of overall largest square are for given matrix. Therefore, the loop invariant holds true after termination of algorithm.

Time Complexity

As we can see there are three loops for traversing through rows and three loops for traversing through columns of the matrix. First, two loops are to find the start_row and start_col of the square area. The next two loops are to find the end_row and end_col of the square area. And last two loops are to check each plot enclosed within a square area.

This gives us a time complexity of $\mathbf{O(m^3n^3)}$.

Space Complexity

We do not maintain any previous state here. Space complexity here is **O (1).**

##ALG 2 TASK 2 – ($O(m^2n^2)$):

2.1 DESIGN

1. The algorithm starts with initializing the result list to [0,0,0,0] where the upper left corner (starting row and starting column) of the largest possible square area is (result [0], result [1]) and the bottom right corner(ending row and ending column) is (result[3], result[4]).

2. The algorithm iterates through every element of the matrix by using two nested loops, start_row, and start_col, which represent the starting row and starting column of the square area. Inside loops, we define parameter square_size as 0.

3. While traversing through the matrix, each element is checked to find out if the number of trees in the plot is greater than or equal to h. If it is true, the algorithm increases the square area by adding the consecutive right column and bottom row to the current square.

4. Now, the algorithm checks if all the plots in the newly added row and column have the minimum number of trees required, instead of checking every plot in a square area.

5. If all the plots of the newly added rows and columns have the minimum number of trees required, the algorithm will update the upper left and bottom right corners of the max square area found.

6. After traversing through each element, at termination latest upper left and the bottom right corner is returned.

Consider following example with m = 6, n =5 and h =3

2	3	4	5	1
4	3	4	5	4
1	6	7	8	6
2	4	4	3	4
2	6	6	8	4
4	7	7	2	7

Grey area represents, newly added row and column to earlier largest square area.

Taking these pointers in mind we devise the following algorithm

2.2 PSEUDO CODE

TASK2 (m, n, h, matrix)

```
res ← [0,0,0,0]

FOR start_row = 1 to m:
        FOR start_col = 1 to n:
                IF (matrix[start_row][start_col] >= h) THEN
                        flag ← True
                        square_size ← 0
                        end_row ← start_row
                        end_col ← start_col
                        WHILE (end_row <= m and end_col <= n and flag)
                                FOR j = start_col to end_col:
                                        IF (matrix[end_row] [j] < h) THEN
                                                flag ← False
                                                break
                                FOR i = start_row to end_row:
                                        IF (matrix[i] [end_col] < h) THEN
                                                flag ← False
                                                break

                                IF (flag == True) THEN
                                        temp ← [start_row, start_col, end_row, end_col]
                                        temp_area ← (end_row-start_row) * (end_col-start_col)
                                        max_area ← (res[2] – res[0]) * (res[3] – res[1])

                                        IF (max_area < temp_area) THEN
                                                res ← temp

                                        end_row ← end_row + square_size
                                        end_col ← end_col + square_size
                                        square_size ← square_size + 1


                        END WHILE
RETURN res
```

2.3 ANALYSIS

Proof of Correctness

Loop Invariant:

At the start of each iteration **res** contains the upper left corner and bottom right corner of the max square area obtained till now, where as temp keep tracks of

bounding indices of the square area which has particular plot as its upper left corner.

Initialization:

For every iteration, first start_row and start_col is chosen, this is done by traversing through every element of matrix. The square_size is set to 0 and end_row and end_col are initialized to start_row and start_col only if matrix[start_row] [start_col]>=h.

Maintenance:

At every iteration, we are checking only newly added row and column, instead of every plot enclosed within a square area. If every plot of newly added row and column have the minimum number of trees required, res is updated for the upper left and bottom right corner of the largest square area till now and square_size, end_row and end-col are incremented.

Termination:

The program will terminate when start_row = m or start_col = n i.e., when all possible elements are considered to be the upper left corner of the largest square area. "res" will store top left and bottom right corners of overall largest square are for given matrix. Therefore, the loop invariant holds true after termination of algorithm.

Time Complexity

As we can see there are two loops for traversing through rows and two loops for traversing through columns of the matrix. First two loops are to find the start_row and start_col of the square area. The next two loops are to find the end_row and end_col of the square area.

This gives us a time complexity of O(mn*min(m,n)*(m+n)) -> **$O(m^2n^2)$**

Space Complexity

We do not maintain any previous state here. We are just maintaining the variable square_size. Space complexity here is **O (1).**

##ALG 3 TASK 3 – Dynamic Programming - (O(mn)):

3.1 DESIGN

1. The algorithm starts with initializing the result list to [0,0,0,0] where the upper left corner (starting row and starting column) of the largest possible square area is (result [0], result[1]) and the bottom right corner(ending row and ending column) is (result[3], result[4]).

2. Create the matrix "square_size" of size (m+1) * (n+1) to store the square_size of the square area where the plot under consideration is the bottom right plot. Initialize all elements of this matrix to 0.

3. The algorithm will traverse through all the elements of the matrix, and if the plot has a minimum number of trees required, then try to find the largest possible square area, where the plot under consideration is a right bottom plot of the square area. Here, the idea is that we can increment the size of the square by adding one row and column to a square area whose right bottom corner is either a left diagonal plot or consecutive upper or consecutive left plots.

4. The dp matrix square size is populated using the following recurrence relationship:

   BELLMAN EQUATION
   square_size[end_row] [end_col] = min (square_size [end_row -1] [end_col], square_size[end_row-1] [end_col-1], square_size[end_row] [end_col-1]) + 1

5. The square_size at the current position is compared with the max square area and the max square area is updated accordingly.

Consider following example with m = 4, n =5 and h =3. The dp-matrix "square_size" is as follows:

2	3	4	5	1
1	3	4	5	1
0	6	7	8	2
2	2	1	3	4

Square_size

0	0	0	0	0	0
0	0	1	1	1	0
0	0	1	2	2	0
0	0	1	2	3	0
0	0	0	0	1	1

Taking these pointers in mind we devise the following algorithm

3.2 PSEUDO CODE:

```
TASK3 (m, n, h, matrix)

// Create a (m+1) * (n+1) matrix square_size and initialize every element to 0

max_square ← 0
res ← [0,0,0,0]

FOR end_row = 1 to m:
        FOR start_col = 1 to n:
                IF (matrix[start_row] [start_col] >= h) THEN
                        square_size[end_row] [end_col] ←
                                min (square_size[end_row-1] [end_col],
                                square_size[end_row-1] [end_col-1],
                                square_size[end_row] [end_col-1]) + 1

                        IF (square_size[end_row] [end_col] > max_square) THEN
                                max_square ← square_size [end_row] [end_col]
                                res ← [end_row-max_square+1,
                                        end_col-max_square+1, end_row, end_col]

RETURN res
```

3.3 ANALYSIS

Proof of Correctness

Loop Invariant:

At the start of each iteration **res** contains the upper left corner and bottom right corner of the max square area obtained till now, whereas square_size [i, j] keep tracks of largest possible square area where matrix[i,j] is bottom right plot of square area.

Initialization:

At the start of algorithm, before first iteration every element of square_size is initialized to 0. “max_square” is initialized to 0.

Maintenance:

At every iteration, if the plot under consideration has minimum required trees i.e., then square_size at that position is calculated by using the minimum of square sizes of three adjacent positions, left, top, left-diagonal and then adding 1 to it. Also, the max_square is updated accordingly.

Termination:

The program will terminate when every element of the matrix is traversed, the "square_size" will store the largest possible square area with the current position as right bottom plot. "res" will store overall largest square are for given matrix. Therefore, the loop invariant holds true after termination of algorithm.

Time Complexity

As we can see there is only one loop traversing through rows and one loops traversing through columns of the matrix.

This gives us a time complexity of **O (mn)**

Space Complexity

The algorithm uses "square_size" matrix of size (m+1) * (n+1). Additionally, "res" requires O (1) space to store bounding indices of the largest possible square satisfying required condition. This gives us space complexity of **O(mn).**

PROBLEM 2

Given a matrix p of m × n integers (non-negative) representing the minimum number of trees that must be planted on each plot and an integer h (positive), find the bounding indices of a square area where all but the corner plots enclosed requires a minimum of h trees to be planted. The corner plots can have any number of trees required.

##ALG 4 TASK 4 – Dynamic Programming - ($O(mn^2)$):

4.1 DESIGN

1. Let dp[r][c1][c2] denote the size of largest rectangle with (r,c1) and (r,c2) as bottom vertices. We iterate through all the possible combinations of c1,c2 and find the largest rectangle that satisfy the above relation. Upon find the largest rectangle that is a square, we update the result.

2. We consider each rectangle such that only the bottom corners can have corners plots < h. For every r,c1,c2 we check if the rectangle is a square by taking the width of the plot that is c2-c1. If the width satisfied the height of the largest square for that r,c1,c2 we compare the size of the square with older largest square and update accordingly.

3. When stumbled upon a corner plot such that (r c2) < h, we stop updating dp[r][c1][c2] for the remaining values of c2 as they are bound to be 0

4. The recurrence relation here is as follows:

   BELLMAN EQUATION
   {
   dp[r][c1] [c2] = ***1***
   } if only **matrix[r-1] [c1-1]<h or matrix[r-1][c2-1]**
   Else
   {
   dp[r][c1] [c2] = ***1*** + ***dp[r-1][c1] [c2]***
   } if **matrix[r-1] [c1-1] to matrix[r-1][c2-1]>=h**
   Otherwise
   {
   dp[r][c1] [c2] = ***0***
   }

Taking these pointers in mind we devise the Algorithm for Task 4.

4.2 PSEUDO CODE

TASK4 (r, c, h, matrix, dp)

```
// Create a (m+1) * (n+1) * (n+1) matrix square_size and initialize every
element to 0

max_square ← 0
res ← [0,0,0,0]

FOR  r  = 1 to m:
        FOR  c1 = 1 to n:
                Set val=1 if matrix[r][c]<h else Set it to dp[r-1][c1][c1]+1
                //start for different c2 values
                FOR  c2 = c1 to n:
                Dp[r][c1][c2] = 1 if val =1 else dp[r-1][c1][c2]+1
        //check if the vertices form a square
                If dp[r-1][c1][c2]+1 == c2-c1+1 Set temp=c2-c1+1
                Else set temp=1
                IF (matrix[r][c] < h) THEN
                   dp[r][c1][c2] =
                IF (temp > max_square) THEN
                        max_square ← temp
                        res ← [end_row-max_square+1,
                        end_col-max_square+1, end_row, end_col]

RETURN res
```

4.3 ANALYSIS

Proof of Correctness

Loop Invariant:

At the start of each iteration **res** contains the upper left corner and bottom right corner of the max square area obtained till now, where as max_square keep tracks of size of largest square area.

Initialization:

For every iteration, first r and c1 is chosen, this is done by traversing through every element of matrix. Then for those r and c1, we iterate through all c2's

starting from c1. Dp is set to 0 for that r,c1,c2 and updated in the loop. Val is initialized based on current matrix element if it is < h, it's initialized to 1 else its initialized to dp(r-1,c1,c2). Temp is also 1 if the vertices do not form a square else it is updated to the size of the square and then values for result are updated based on temp.

Maintenance:

At every iteration, we are checking only newly added (r,c2) pair for give c1, instead of every plot enclosed within a square area. Upon encountering one more plot such that matrix(r,c2)<h, we stop updating other values in the loop for rest of the values between c2 to n as they won't form an acceptable square or rectangle.

Termination:

The program will terminate when we finish traversing for all possible combinations of r,c1 and c2, that is at r=m and c1=n and c2=n. Variable "res" will store top left and bottom right corners of overall largest square are for given matrix. Therefore, the loop invariant holds true after termination of algorithm.

Time Complexity

As we can see there is one loop for traversing through all the rows and two loops for traversing through all combinations of c1 and c2.

This gives us a time complexity of O(m*n*n)) -> **$O(mn^2)$**

Space Complexity

We maintain a 3D array to store our solution for (r,c1,c2) and update other values in O(1) .Space complexity here is **$O(mn^2)$**

##ALG 5 - Dynamic Programming - (O(mn)):

5.1 DESIGN

5. The algorithm requires to construct 3 2-dimentional dp array (dp[i][j]) namely dpD,dpL,dpR where i represent row index, j represent column index and value of D,L and R represent direction of square. Another dp array stores the largest plot with r,c as the bottom right plot with corners that might be less than h.

6. The idea here is to store the size of largest possible square with all but one corner plot in direction of k enclosed requiring minimum of h trees, with its bottom right corner at (i,j) and oriented in direction k, where k is top left, bottom left and top right.

7. We populate our dp matrices, such that if any element is less than h, the entry becomes 1. Also any neighbouring element except for the one in the direction of the given dp, if it is less than h then it becomes 1. Otherwise, the cell is populated with the 1+min of its neighbours.

8. Using this logic we create the three dp matrices dpL,dpD and dpT. While computing the unknown cells recursively we call the TOPLEFTCORNER() from all the cells to make sure we are starting the search similar to what we did for the m,n cell. This generates the subproblems.

9. Using these matrices we calculate our dp solution using the following bellman equation.

   BELLMAN EQUATION
   {
   {
   dp[r][c]] = min (dpD[r-1][c-1], dpT[r-1][c],dpL[r] [c-1]) +1
   } if i>1 and j>1
   else {
   dp[r][c] = 1
   }

10. Here, we are considering 3 directions DIAGONAL, LEFT and TOP to determine the largest square area at (r,c) with all but the corner plots enclosed requiring a minimum of h trees. It takes the minimum of the sizes of the squares in the three directions and adds 1 to it, representing the current square. If the current square size is greater than the maximum square size seen so far, the maximum square size and the corresponding indices are updated.

11. Finally, the algorithm will return the bounding indices of the largest square with all but the corner plots enclosed requiring a minimum of h trees.

Taking these pointers in mind we devise the following 2 tasks, 5A with recursive implementation using memoization of Algo5 and 5B with bottom up iterative approach.

##TASK 5A- recursive implementation of Alg5 using Memoization – O(mn)

1. PSEUDO CODE

TopLeftCorner (r, c, h, matrix, dp)

```
IF (matrix(r,c)<h or matrix(r)(c-1)<h or matrix(r-1,c)<h) THEN dpD[r][c] ← 1

ELIF
        IF dpD(r,c-1) == -1 THEN
            dpD(r,c-1) ←  TopLeftCorner (r,c-1,h,matrix,dpD)

        IF dpD(r-1,c-1) == -1 THEN
            dpD(r-1,c-1) ←  TopRightCorner(r-1,c-1,h,matrix,dpD)

        IF dpD(r-1,c) == -1 THEN
            dpD(r-1,c) ←  TopRightCorner(r-1,c,h,matrix,dpD)


        dpD(r,c)←  min(dpD(r,c-1),dpD(r-1,c-1), dpD(r-1,c))+1
```

TopRightCorner (r, c, h, matrix, dp)

```
IF (matrix(r,c)<h or matrix(r)(c-1)<h or matrix(r-1,c-1)<h) THEN dpR(r,c) ← 1

ELIF
        IF dpR(r,c-1) == -1 THEN
            dpR(r,c-1) ←  TopLeftCorner (r,c-1,h,matrix,dpR)

        IF dpR(r-1,c-1) == -1 THEN
            dpR(r-1,c-1) ←  TopRightCorner(r-1,c-1,h,matrix,dpR)

        IF dpR(r-1,c) == -1 THEN
            dpR(r-1,c) ←  TopRightCorner(r-1,c,h,matrix,dpR)


        dpD(r,c)←  min(dpD(r,c-1),dpD(r-1,c-1), dpD(r-1,c))+1
```

BottomLeftCorner (r, c, h, matrix, dp)

IF (matrix(r,c)<h or matrix(r-1)(c-1)<h or matrix(r-1,c)<h) THEN dpL(r,c) ← 1

ELIF

IF dpL(r,c-1) == -1 THEN
dpL(r,c-1) ← **TopLeftCorner** (r,c-1,h,matrix,dpL)

IF dpL(r-1,c-1) == -1 THEN
dpL(r-1,c-1) ← **TopRightCorner**(r-1,c-1,h,matrix,dpL)

IF dpL(r-1,c) == -1 THEN
dpL(r-1,c) ← **TopRightCorner**(r-1,c,h,matrix,dpL)

dpD(r,c)← min(dpD(r,c-1),dpD(r-1,c-1), dpD(r-1,c))+1

To calculate dp from this

1. **Loop over rows:**
   a. **Loop over cols:**
      i. **If row==0 or col==0**
         1. **Set dpD,dp,dpR,dpL for row and col to 1**

      Else if dpD(row-1,col-1) =-1
      TopLeftCorner(matrix,dpD,row-1,col-1,h)
      //Similarly do this for dpR(row-1,col) and dpL(row,col-1)

```
Dp(r,c)= min(dpD[i-1][j-1], dpR[i-1][j],
dpL[i][j-1])+1
```

To calculate result from this

1. Initialize `maxSquareSize` to 0

2. Loop over the rows of `dp`:

   a. Loop over the columns of `dp`:

      i. If `dp[r][c]` is greater than or equal to `maxSquareSize`:

         - Set `maxSquareSize` to `dp[r][c]`

         - Set `max_square` to `dp[r][c]`

         - Set `result` to the coordinates of the current square:

- The top left corner is `(r - max_square + 1, c - max_square + 1)`

- The bottom right corner is `(r, c)`

3. Return `result`

##TASK 5B- Iterative Bottom-Up implementation of Alg5– O(mn)

1. PSEUDO CODE

TASK5B (m, n, h, matrix)

// Create 3 2D matrix dpD, dpL, dpT of size (m+1) * (n+1) and initialize all elements with 0

// Generate dpT, dpL and dpD using the following logic
//let K denote the orientation in dpK
FOR r = 1 to m:
    FOR c = 1 to n:
        IF r=1 or c=1 or matrix(r,c)<h:
            dpK(r,c)=1
        ELSE
        //check the neighbours except for the K direction one
        //if found < h then set it to 1
        IF any neighbouring elements except for the Kth direction <h
        DpK(r, c) = 1
        Else
        dpK[r][c] = min(dpK[r][c-1], dpK[r-1][c], dpK[r-1][c-1])+ 1

// After running this loop for the 3 matrices we generate the solution

Max_square = 0
 FOR r = 1 to m:
    FOR c = 1 to n:
        IF r=1 or c=1 :
            dpK(r,c)=1
        ELSE;
            dp[r][c] = min(dpD[r-1][c-1], dpT[r-1][c], dpL[r][c-1])+1
        IF dp(r,c) > max_square:
            Max_square = dp(r,c)
            Update result

2. ANALYSIS

Proof of Correctness

Loop Invariant:

At the end of each iteration of the outer loop (traversing through rows), the variable 'max_square' contains the size of the largest square area of the matrix that satisfies the condition of having all but corner plots with a minimum of h trees to be planted, and 'res' contains the indices of the bounding square area.

Initialization:

Bottom up -

At the start of algorithm, 3 'dp' matrix of size (n+1) x (m+1) are created and every element is initialized to 0. Dp(i, j) represents the size of the largest square area ending at (i, j) with its bottom-right corner at (i, j).Before the first iteration 'max_square' is initialized to 0.

Recursive –

In this approach the arrays are initialized to -1 except for the dp array which will store the length of the largest rectangle. Similar to iterative approach the max_saquare will be 0 and updated as and when we get square of size> max_square.

Maintenance:

In each iteration of the outer loop, algorithm iterates through each element (r, c) of the matrix. It computes 'dp' as the size of the largest square area ending at (r, c) with its bottom-right corner at (r, c) and updates 'max_square' and 'result' if 'curr_square' is greater than 'max_square'. If the value of the element matrix[r-1][c-1] is greater than or equal to h, the algorithm updates ''dp' matrix at position (r, c) for each direction (DIAGONAL, LEFT, and TOP) by taking the minimum value from the adjacent elements in that direction and adding 1 to leeway the plot <h at the end. Otherwise, sets 0 for all directions.

Termination:

When the iterations complete, the 'res' variable contains the indices of the bounding square area with all but corner plots enclosed requiring a minimum of h trees to be planted. Therefore, loop invariant holds true after termination of algorithm

Time Complexity

The nested loops iterate through each element of the matrix, and the computations within each iteration take constant time. For computing these loops, we need O(4mn) time.

In recursive code, each loop will call three functions in nested fashion, however because of memorization, we will avoid visiting elements which are already computed. Hence the algorithm will therefore run till 3mn elements for the 3 dp matrices. Once that the computation of largest square will take constant time. Therefore, the time complexity is **O (mn).**

Space Complexity

The space complexity of this algorithm is O(mn), where m and n are the dimensions of the matrix. This is because the algorithm uses a 4 2-dimensional dp matrix of size (n+1) * (m+1) to store the largest square area for each element in the matrix, and each of these matrix has mn entries. Additionally, the algorithm uses a constant amount of extra space for the 'max_square', result etc variables. Therefore, the overall space complexity is **O(mn).**

PROBLEM 3

Given a matrix p of m × n integers (non-negative) representing the minimum number of trees that must be planted on each plot and an integer h (positive), find the bounding indices of a square area where only up to k enclosed plots can have a minimum tree requirement of less than h.

##Algo 6 TASK 6- Brute Force– $O(m^3n^3)$

6.1 DESIGN

1. The variables start_row, start_col, end_row, and end_col determine the beginning and ending points of the sub-square that is being examined.

2. To implement the Brute force approach, the algorithm use a nested loop to traverse the matrix, where each element matrix [i,j] represents the minimum number of trees that must be planted on that plot. When we come across a plot with p[i,j]>=h, the algorithm marks that plot as the starting point of our square area.

3. Next, the algorithm uses a while loop to traverse all the squares that start from the plot identified above until we reach the end of the matrix (end_row < m and end_col < n).

4. Within the while loop, two nested loops are used for i and j, ranging from start_row to end_row and start_col to end_col respectively, to traverse through the square area. Here the variable 'count' is initialized to 0.

5. For each plot within the square area, the algorithm checks whether p[i,j] < h and increment the count if it is true.

6. If the count<=k, which means we come across at most k plots with p[i,j] < h, here the algorithm calculate the area of the sub-square and compare it with the maximum area obtained so far and updates the maximum area accordingly.

7. End while loop, when each sub-square is considered. Return the bounding upper left corner and right bottom right corner of the max square area found. The time complexity of this brute force approach is $O(m^3n^3)$.

Taking these pointers in mind we devise the following algorithm

6.2 PSEUDO CODE

```
TASK6 (m, n, h, k, matrix)

res ← [0,0,0,0]

FOR start_row = 1 to m:
        FOR start_col = 1 to n:
                IF (matrix[start_row] [start_col] >= h) THEN
                        end_row ← start_row
                        end_col ← start_col
                        WHILE (end_row <= m and end_col <= n )
                                count ← 0
                                FOR i = start_row to end_row:
                                        FOR j = start_col to end_col:
                                                IF (matrix[i][j] >= h) THEN CONTINUE
                                                ELSE
                                                        count ← count+1
                                IF (count <= k) THEN
                                        temp ← [start_row, start_col, end_row, end_col]
                                        temp_area ← (end_row-start_row) * (end_col-start_col)
                                        max_area ← (res[2] – res[0]) * (res[3] – res[1])
                                        IF (max_area < temp_area) THEN
                                                res ← temp
                                end_row ← end_row + 1
                                end_col ← end_col +1
                        END WHILE
RETURN res
```

6.3 ANALYSIS

Proof of correctness:

Loop Invariant:

At any point during the execution of the loops, res contains the indices of the largest square area that satisfies the condition of having at most k plots with a minimum tree requirement of less than h, out of all the square areas examined so far.

Initialization:

The algorithm initializes res to [0, 0, 0, 0] at the start. Res represents indices of top left corner and bottom right corner of largest possible square area satisfying the given condition.

Maintenance:

The algorithm iterates through all possible square areas in the matrix, starting from each plot. For each square area, it counts the number of plots with a minimum tree requirement of less than h, and if this count is less than or equal to k, it compares the area of this square to the area of the largest square found so far. If the new square has a larger area, it updates result with the new square's indices. This process continues until all possible square areas have been examined.

Termination:

When the loops finish iterating through all possible square areas, result contains the indices of the largest square area that satisfies the condition of having at most k plots with a minimum tree requirement of less than h, out of all the square areas examined. Therefore, loop invariant holds true after termination of algorithm.

Time Complexity

As we can see there are three loops for traversing through rows and three loops for traversing through columns of the matrix. First, two loops are to find the start_row and start_col of the square area. The next two loops are to find the end_row and end_col of the square area. And last two loops are to check each plot enclosed within a square area.

This gives us a time complexity of $\mathbf{O(m^3n^3)}$

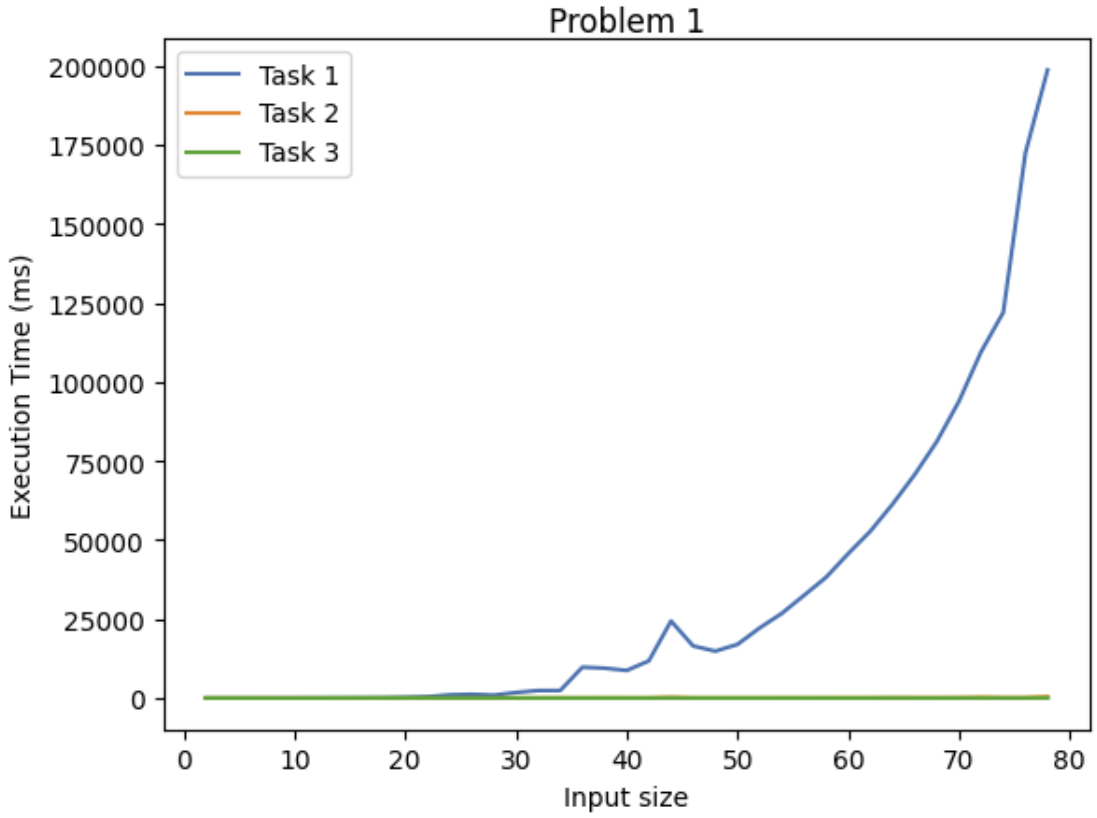
Space Complexity

We do not maintain any previous state here. The 'res', 'temp' variables each require a constant amount of space. Therefore, Space complexity here is **O (1).**

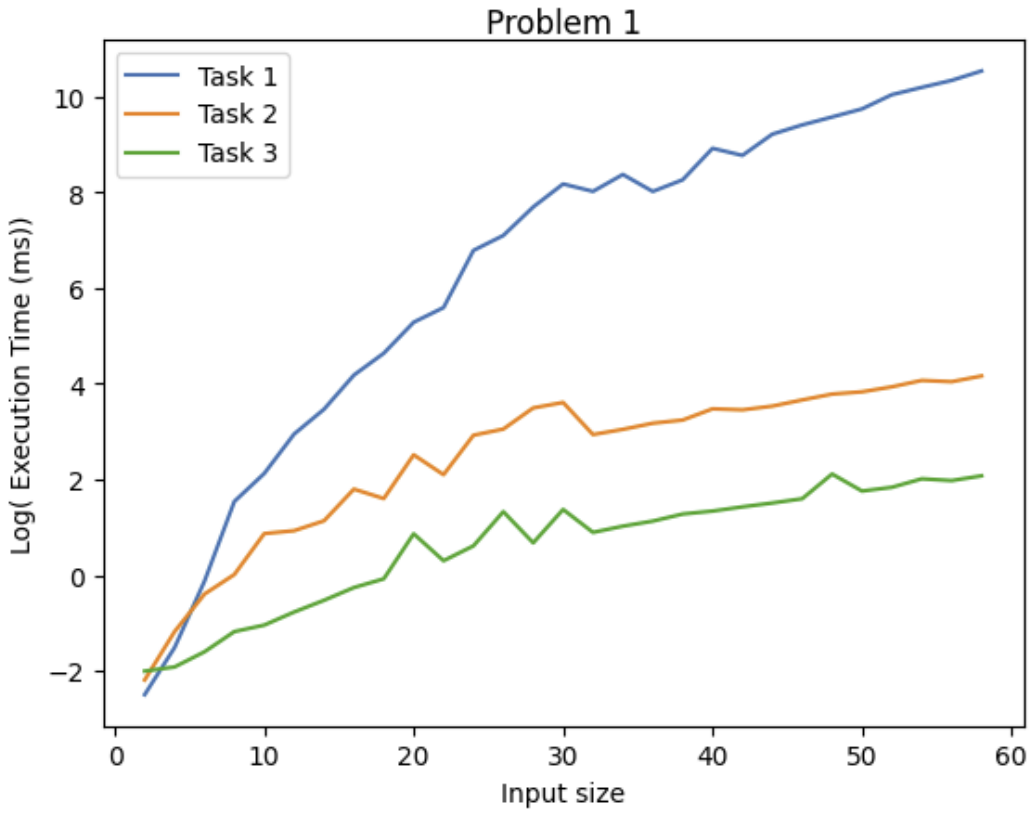
EXPERIMENTAL COMPARITIVE STUDY

PROBLEM 1

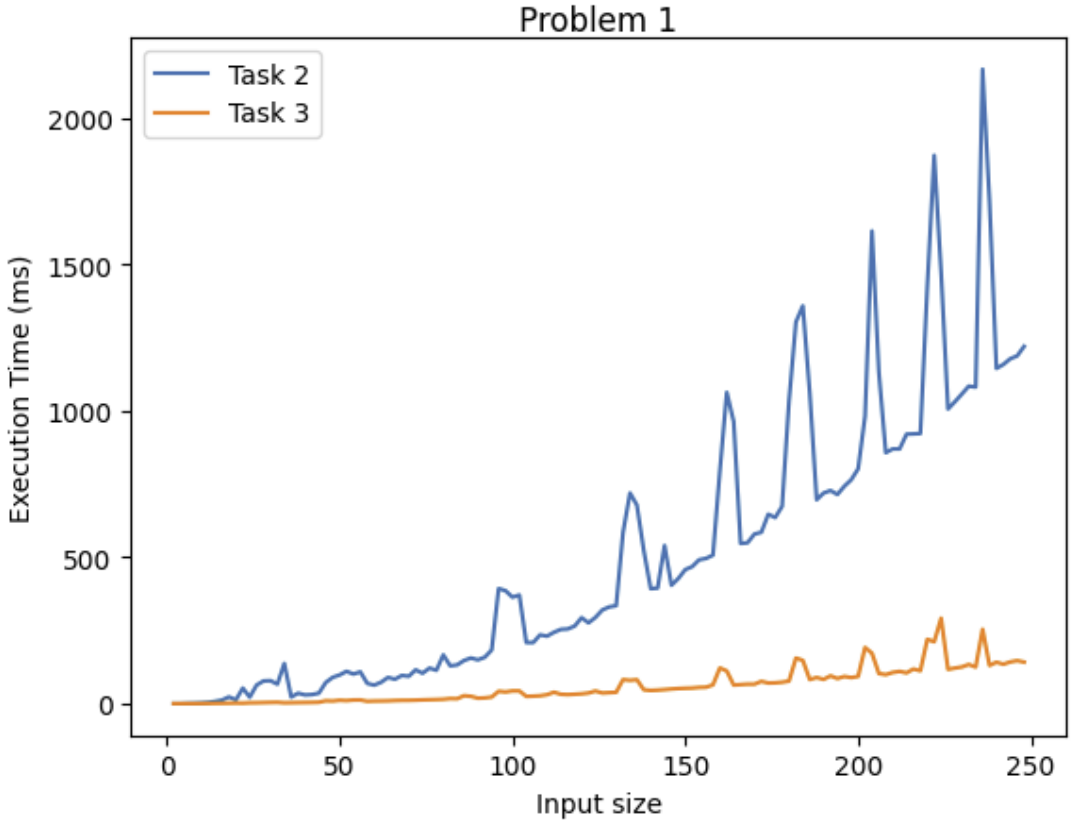
For Problem 1, we executed our implementation for TASK1, TASK2, and TASK3 with the following input size, where m=n, h, and matrix values are randomly generated. Following graph is visual representation of the execution time required for each task for different input sizes.



For Task1, as the input size increases, we can clearly see the graph trends to be like $O(m^3n^3)$, i.e., a cubic function. Following graph is logarithmic representation of execution times, which gives better understanding.



Also, In first graph, as we can see, TASK 2 and TASK3 are almost overlapping, that means the execution times for TASK2 and TASK 3 is very small as compared to TASK1. In following graph, we compare TASK 2 and TASK3 for much larger input size.

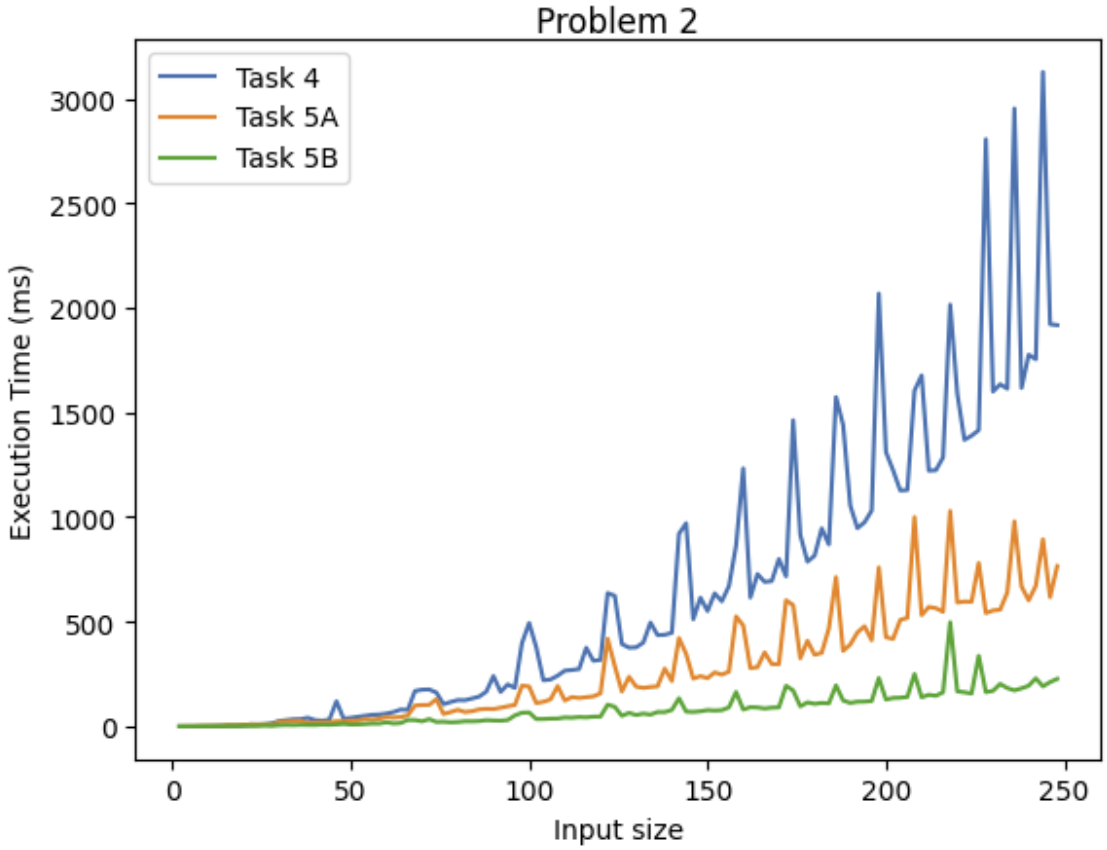


Task 3 is dynamic programming implementation in O(mn) time complexity and the curve is slowing increasing. However, TASK2 have time complexity of $O(m^2n^2)$ and curve is increasing faster as compared to TASK 3.

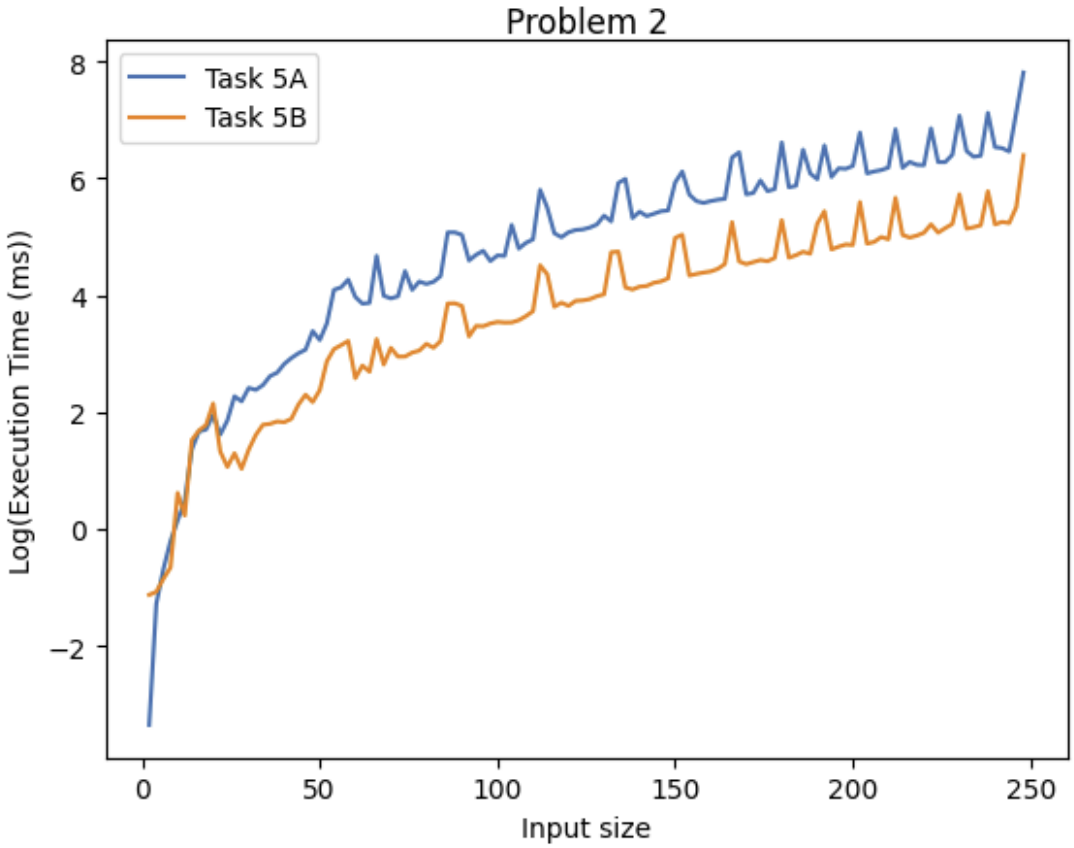
ALGO3, dynamic implementation proves to be fastest among all other algorithms to solve problem 1.

PROBLEM 2

Similar data set is used for Problem 2. Following graph represents running of the TASK4, TASK 5A and TASK 5B. TASK 4 running at $O(mn^2)$ is having increasing curve above both TASK5A and TASK5B which have time complexity of O(mn)



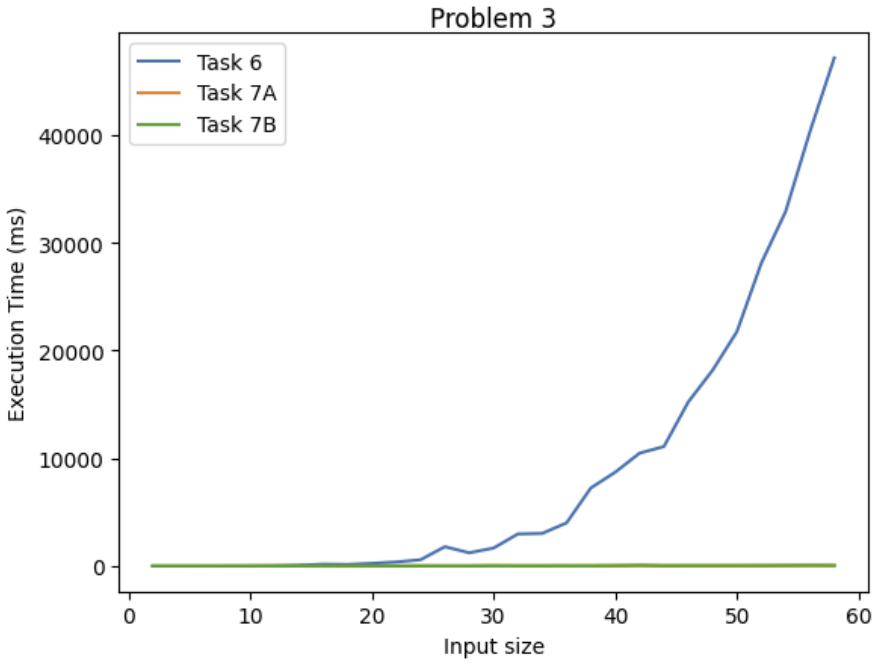
Though both 5A and 5B have same running time complexity O(mn), we can see that the TASK 5A takes longer time than 5B. 5A is recursive implementation of the same algorithm, recursive implementation adds extra overhead due to internal function calling and stack handling. 5A is still running in O(mn) and its graph is close to 5B. Following graph of log (execution time) gives better understanding of that.



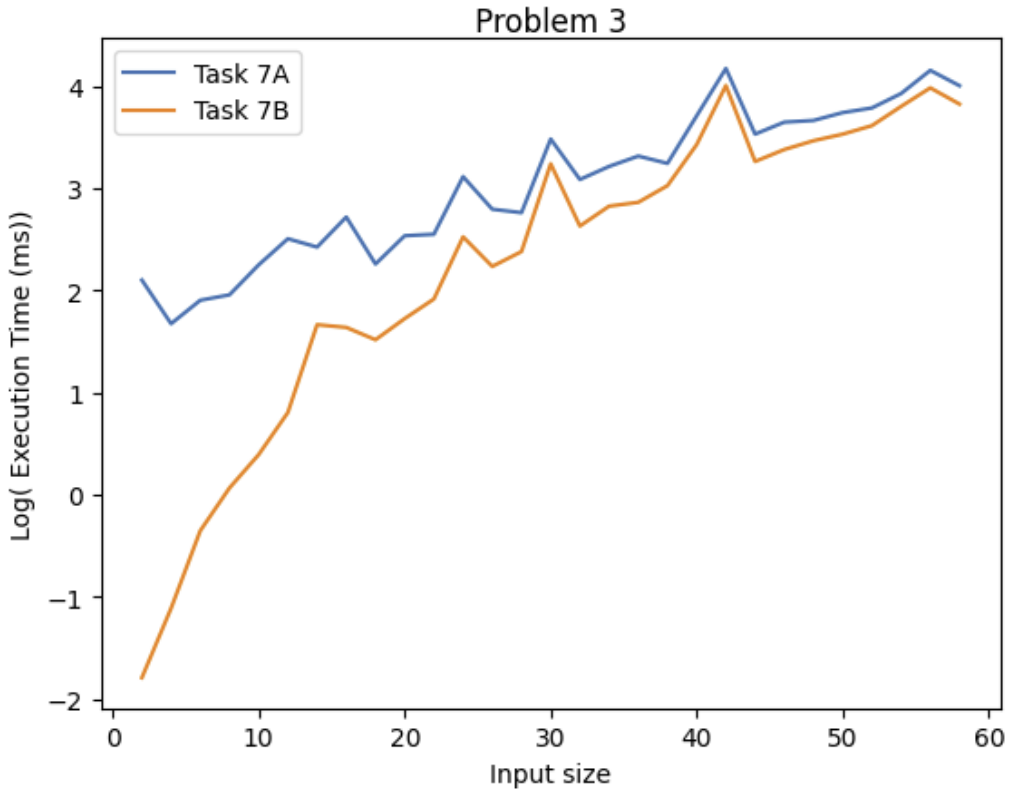
The TASK 5B, bottom up implementation of dynamic programming algo 5 proves to be fastest in experimental analysis.

##PROBLEM 3

Similarly following graph compares TASK6, TASK7A, TASK7B



For better understanding of 7A and 7B time complex, logarithm of execution time vs input size graph



CONCLUSION

- The programming assignment helped us in understanding the dynamic programming in depth. We were able to understand bottom-up approach and top-down approach with memoization clearly.

- This assignment leads us to understand how dynamic programming can reduce time complexity of brute force approach significantly from like $O(m^3n^3)$ to O(mn).

- We were able to grasp concept of overlapping subproblems. And implementing it made it more concrete.

- We were able to visualize running time of each task and how time complexity affects the execution times of algorithms.

Ease of Implementation:

- We chose python for the implementation. Implementation with python is pretty straight forward. We defined two functions get_input1() for problem 1 and problem2 and get_input2() for problem 3.

- The appropriate implementation of each task was based on input and output format. Though output format is same for all tasks, input format was little different. For example: get_input1() takes m, n, h and matrix, whereas get_input2() takes m, n, h, k, matrix as input.

- Also, the data structures like dp matrix and temporary variables needed for each task are defined within the function definition instead of defining it globally.

- To test the implementation of all tasks for various inputs and to analyse the execution time, we wrote python scripts to generate randomized data sets.

- Implementation of brute force approaches were pretty straight forward. The trickiest part was to figuring out how to define dp matrix and recursive function calls for top-down approach with memoization. Compared to recursive implementation, iterative implementation was easy to implement, there we can clearly see corner cases.

Observation:

- Brute force approaches are very easy to implement. Also, space complexity of brute force approaches is less compared to other tasks. But drawback, was high time complexity. In experimental analysis, we saw the evident time different for large input sizes. The tasks took 100-200 times more compared to other tasks

- Dynamic Programming implementations are not that intuitive and are complex to understand and implement. Sometimes dp matrix takes more space than the actual input size. But advantage here is efficient time complexity. DP tasks requires considerably lower time than the brute force tasks.

- Also, as mentioned in the experimental, iterative implementation of DP is faster, since there is no overhead of internal function calling and stack handling.