# COT 5405 – Analysis of Algorithms

## Programming Project II – Dynamic Programming

2nd Apr 2023

Shreya Shah
Siju Sakaria

## Team members

1. **Shreya Shah (UFID: 6207-8655)**
   Worked on design and analysis of Tasks 2, 4, 5A, 5B, 7A, 7B, generated Makefile, conducted experimental study, added test cases, and worked on report

2. **Siju Sakaria (UFID: 1015-4248)**
   Worked on design and analysis of Tasks 1, 2, 3 and 6, plotted graphs, conducted experimental study, added test cases, tested on Thunder server, and worked on report

## Design and Analysis of Algorithms

### Task 1

#### Design

```
Initialize the bounds(x1, y1, x2, y2) of square to be -1
Keep track of maximum size of square(max_size)

For i in 1 to m:
    For j in 1 to n:
        For s in i to m:
            For t in range j to n:
                Initialize count to 0
                If s + 1 - i != t + 1 - j :  (# Not a square)
                    Continue loop
                Set size = s + 1 - i

                For x in j to i to s + 1:
                    For y in j to t + 1:
                        If matrix[x][y] < h:
                            Increment count

                If count is 0 and size > max_size:
                    max_size = size
                    x1 = i + 1
```

$$y1 = j + 1$$
$$x2 = s + 1$$
$$y2 = t + 1$$

*Return x1, x2, y1, y2*

### Correctness

1. We initialize the bounds (x1, y1, x2, y2) to -1 and the maximum size of the square (max_size) to 0. This ensures that if no valid square area is found during the execution, the result will be consistent with the problem statement.
2. The algorithm iterates over all possible top-left corners (i, j) and bottom-right corners (s, t) of the squares. It checks if the current rectangle formed by the corners (i, j) and (s, t) is a square. If not, it continues to the next iteration.
3. For each valid square formed by corners (i, j) and (s, t), the algorithm counts the number of plots that have a minimum tree requirement of less than h.
4. If the count of invalid plots is 0 and the current square size is greater than max_size, the algorithm updates the max_size and the bounding indices of the square (x1, x2, y1, y2).
5. The algorithm terminates after all possible top-left corners (i, j) and bottom-right corners (s, t) have been considered. At this point, if a valid square area has been found, the algorithm will have updated the max_size and the bounding indices of the square (x1, x2, y1, y2). If no valid square area is found, the max_size will remain 0, and the bounding indices will not be updated.

### Time complexity

Combining all 6 loops, the overall time complexity of the algorithm is Θ(mn * mn * mn) = Θ(m^3n^3).

### Space complexity

The space complexity of the algorithm is $O(1)$ since we are just storing the bounds and not using any other data structure.

## Task 2

### Design

*Initialize the bounds(x1, y1, x2, y2) of square to be -1*
*Keep track of maximum size of square(max_size)*

```
For i in 1 to m:
    For j in 1 to n:
        For sq_size in 1 to min(m - i + 1, n - j + 1):
            Initialize valid_area to True
            For row in i to i + sq_size:
                For col in j to j + sq_size:
                    If matrix[row][col] < h:
                        Set Valid_area to False
                        Break loop
                If valid_area is False:
                    Break loop
            If valid_area found and sq_size > max_size:
                Max_size = sq_size
                x1 = i + 1
                y1 = j + 1
                x2 = i + sq_size
                y2 = j + sq_size

Return x1, x2, y1, y2
```

## Correctness

1. We initialize max_size to 0, and we have not found a valid square area yet. This means that if no valid square area is found during the execution, the result will be consistent with the problem statement.
2. We iterate over all possible top-left corners (i, j) of the square (i from 1 to m, and j from 1 to n), and for each top-left corner, we iterate over all possible square sizes sq_size (from 1 to min(m - i + 1, n - j + 1)). This ensures that we cover all possible squares within the matrix.
3. For each square with top-left corner (i, j) and size sq_size, we check if all enclosed plots require a minimum of h trees. If any plot does not meet the requirement, we mark valid_area as False and break out of the inner loop. If we find a valid square area and its size is greater than the current max_size, we update the max_size and the bounding indices of the square (x1, x2, y1, y2).
4. The algorithm terminates after all top-left corners (i, j) and square sizes sq_size have been considered. At this point, if a valid square area has been found, the algorithm will have updated the max_size and the bounding indices of the square (x1, x2, y1, y2). If no valid square area is found, the max_size will remain 0, and the bounding indices will not be updated.

### Time complexity

Time complexity of the algorithm is Θ(m * n * min(m, n) * mn) = Θ(m^2n^2).

### Space complexity

The space complexity of the algorithm is O(1) since we are just storing the bounds and not using any other data structure.

## Task 3 (Dynamic Programming)

### Design

*Initialize the bounds(x1, y1, x2, y2) of square to be -1*
*Create a dp array of size (m + 1) * (n + 1)*
*Keep track of maximum size of square(max_size)*

```
For i in 1 to m + 1:
    For j in 1 to n + 1:
        If matrix[i - 1] [j - 1] >= h:
                Set dp[i][j] to min(dp[i- 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1
                If dp[i][j] > max_size:
                        Max_size = dp[i][j]
                        x1 = i - max_size + 1
                        y1 =j - max_size + 1
                        x2 = i
                        y2 = j

Return x1, x2, y1, y2
```

### Recursive Formulation

$$
\begin{cases}
dp[i][j] = 0 & \text{if } i = 0 \text{ or } j = 0 \\
dp[i][j] = \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1 & \text{if } matrix[i-1][j-1] >= h \\
dp[i][j] = 0 & \text{otherwise}
\end{cases}
$$

### Correctness

Base case:

When i=0 or j=0, dp[i][j] is set to 0, which corresponds to the boundary of the matrix.

Inductive step:
For each cell (i, j), if matrix[i-1][j-1] >= h, then it means the current plot requires a minimum of h trees, and we can try to extend the square. We take the minimum of the neighboring cells dp[i-1][j], dp[i][j-1], and dp[i-1][j-1] and add 1 to it. This operation ensures that we find the largest square that can be formed using the current cell as its bottom-right corner while satisfying the h trees requirement.

If matrix[i-1][j-1] < h, we set dp[i][j] to 0, as we cannot form a valid square using the current cell as its bottom-right corner.

### Time complexity

Iterating through the plots to fill the dp array takes Θ(mn) time.

### Space complexity

The space complexity of the algorithm is O(mn) which is used by the dp array

## Task 4 (Dynamic Programming)

### Design

```
Initialize the bounds(x1, y1, x2, y2) of square to be -1
Create a dp array of size (m + 1) * (n + 1)
Keep track of maximum size of square(max_size)

For i in 1 to m + 1:
    For j in 1 to n + 1:
        If matrix[i - 1] [j - 1]< h:
                Set temp to 1
        Else:
                Set temp to 0
        Set dp[i][j] to dp[i- 1][j] + dp[i][j - 1] - dp[i - 1][j - 1]) + temp

        For x in 0 to min(i, j):
                ans = dp[i][j] + dp[i - x + 1][j - x - 1] - dp[i - x-1][j] - dp[i][j - x
        -1]

                If ans <= 4 and x+1 > max_size:
```

```
Set corner_points to [(i, j), (i - x, j - x), (i - x, j), (i, j - x)]
Set count to 0
For a, b in corner_points:
        if matrix[a - 1][b - 1] < h:
                count += 1
If count == ans:
        max_size = x + 1
        x1 = i - x
        y1 =j - x
        x2 = i
        y2 = j
```

*Return x1, x2, y1, y2*

### Recursive Formulation

$$dp[i][j] = 0, \quad \text{if } i = 0 \text{ or } j = 0$$

$$dp[i][j] = dp[i-1][j] + dp[i][j-1] - dp[i-1][j-1] + temp, \text{ otherwise, where}$$

$$temp = 1, \text{ if } matrix[i-1][j-1] < h$$

$$temp = 0, \text{ otherwise}$$

### Correctness

Base case:
When i=0 or j=0, dp[i][j] is set to 0, which corresponds to the boundary of the matrix.

Inductive step:
For each cell (i, j), the algorithm updates the DP table dp[i][j] based on the values of its neighboring cells (dp[i-1][j], dp[i][j-1], and dp[i-1][j-1]). This ensures that the DP table correctly counts the number of cells with a minimum tree requirement of less than h in the submatrix formed by the top-left corner (0, 0) and the bottom-right corner (i-1, j-1).

Finding the largest square area:
For each cell (i, j), the algorithm iterates through all possible square sizes (x) and checks if the current square has at most 4 cells with a minimum tree requirement of less than h. If so, and the square size is greater than the current maximum size, it updates the maximum size and the bounding indices of the square (x1, y1, x2, y2).

The algorithm terminates after all cells (i, j) have been considered. At this point, if a valid square area has been found, the algorithm will have updated the maximum size and the bounding indices of the square (x1, x2, y1, y2). If no valid square area is found, the maximum size will remain 0, and the bounding indices will not be updated.

### Time complexity

Iterating through the plots to fill the dp array takes Θ(mn) time and for each cell, it iterates through all possible square sizes (min(m, n)). So, total time complexity is Θ(mn^2)

### Space complexity

The space complexity of the algorithm is O(mn) which is used by the dp array

## Task 5A (Dynamic Programming - Recursive)

### Design

*Initialize the bounds(x1, y1, x2, y2) of square to be -1*
*Create a dp array of size (m + 1) * (n + 1)*
*Keep track of maximum size of square(max_size)*

*Create a recursive function(helper) with parameters i, j (size of plot)*
*If i is 0 or j is 0:*
*Set dp[i][j] to 0*
*Return dp[i][j]*
*If dp[i][j] contains value:*
*Return dp[i][j]*
*If plot[i - 1] [j - 1] < h:*
*Set dp[i][j] to 0*
*Else:*
*Set dp[i][j] to min(helper(i- 1, j), helper(i, j - 1), helper(i- 1, j - 1)) + 1*
*Set l to min(helper(i- 1, j), helper(i, j - 1), helper(i- 1, j - 1))*
*If l + 2 >= max_size:*
*Max_size = l + 2*
*x1 = i - max_size + 1*
*y1 = j - max_size + 1*
*x2 = i*
*y2 = j*
*Return dp[i][j]*

*Call recursive fn helper(m, n)*

*Return x1, y1, x2, y2*

### Recursive Formulation

Let's define a function f(i, j) that returns the maximum square size with a minimum tree requirement of at least h, for a submatrix of size i x j, where the bottom-right corner of the submatrix is at the cell (i, j).

*Base case:*
$f(0, ) = 0$
$f(, 0) = 0$

*Recursive case:*
*if* $p[i-1][j-1] < h$: $f(i, j) = 0$
*Else:* $f(i, j) = \min(f(i-1, j), f(i, j-1), f(i-1, j-1)) + 1$

### Correctness

1. The recursive function f(i, j) evaluates the maximum square size that ends at the current cell (i, j). It considers three possible ways to extend the square size: horizontally, vertically, or diagonally. By taking the minimum of these three options and adding 1, it ensures that the cell (i, j) is included in the square.

2. When p[i-1][j-1] < h, it means the current cell does not meet the minimum tree requirement, so the function returns 0, indicating that no valid square can be formed that includes this cell.

3. The algorithm iterates through all cells in the matrix and computes f(i, j) for each cell. It also keeps track of the maximum square size and the indices of its corners. The memoization table guarantees that each subproblem is computed only once, ensuring an efficient solution.

4. By recursively evaluating f(i, j) for all cells in the matrix, the algorithm eventually finds the largest square area that meets the minimum tree requirement.

### Time complexity

The time complexity is determined by the number of unique subproblems (m * n) multiplied by the time taken for each subproblem. Since each subproblem takes constant time (O(1)) outside of the recursive calls, and memoization ensures that each subproblem is only computed once, the time complexity of the algorithm is $\Theta(m * n)$.

### Space complexity

The space complexity of the algorithm is O(mn) which is used by the dp array

## Task 5B (Dynamic Programming - Iterative)

### Design

```
Initialize the bounds(x1, y1, x2, y2) of square to be -1
Create a dp array of size (m + 1) * (n + 1)
Keep track of maximum size of square(max_size)

For i in 1 to m + 1:
    For j in 1 to n + 1:
        If plot[i - 1] [j - 1] >= h:
            Set dp[i][j] to min(dp(i- 1, j), dp(i, j - 1), dp(i- 1, j - 1)) + 1

        Set l to min(dp(i- 1, j), dp(i, j - 1), dp(i- 1, j - 1))
        If l + 2 >= max_size:
            Max_size = l + 2
            x1 = i - max_size + 1
            y1 = j - max_size + 1
            x2 = i
            y2 = j
Return x1, y1, x2, y2
```

### Correctness

1. The given algorithm aims to find the largest square area in a grid where each plot enclosed requires a minimum of h trees to be planted. It uses a dynamic programming approach to achieve this goal.

2. We initialize variables x1, y1, x2, y2 to store the bounding indices of the square area, and maximum_size to keep track of the maximum square size found so far.

3. We construct a 2D DP table called dp with dimensions (m + 1) x (n + 1). This table is used to store the size of the largest square with a lower-right corner at position (i, j) and containing only cells with at least h trees.

4. Then, the algorithm iterates through all cells in the matrix from (1, 1) to (m, n). For each cell (i, j), it checks if the value in the matrix at position [i-1][j-1] is greater than or equal to h. If it is, the cell meets the minimum tree requirement, and the

algorithm updates dp[i][j] to be the minimum of dp[i-1][j], dp[i][j-1], and dp[i-1][j-1] plus 1. This step ensures that the square is extended in either a horizontal, vertical, or diagonal direction while taking the minimum size to ensure all plots enclosed have the minimum tree requirement.

5. The algorithm then calculates the minimum value of l among dp[i-1][j], dp[i][j-1], and dp[i-1][j-1]. If l + 2 is greater than or equal to maximum_size, it updates the maximum_size, x1, y1, x2, and y2. This step is used to keep track of the largest square found so far.

6. By iterating through all cells and updating the dp table accordingly, the algorithm ensures that it finds the largest square area with the required minimum tree count. The final result will be stored in the variables x1, y1, x2, and y2, representing the bounding indices of the square area.

### Time complexity

There is only 1 loop that iterates through the plot size (m * n). So, the time complexity of the algorithm is Θ(m * n).

### Space complexity

The space complexity of the algorithm is O(mn) which is used by the dp array

## Task 6

### Design

```
Initialize the bounds(x1, y1, x2, y2) of square to be -1
Keep track of maximum size of square(max_size)

For i in 1 to m:
    For j in 1 to n:
        For s in i to m:
            For t in range j to n:
                Initialize count to 0
                If s + 1 - i != t + 1 - j :  (# Not a square)
                    Continue loop
                Set size = s + 1 - i

                For x in j to i to s + 1:
```

```
                For y in j to t + 1:
                    If matrix[x][y] < h:
                        Increment count

            If count <= k and size > max_size:
                max_size = size
                x1 = i + 1
                y1 = j + 1
                x2 = s + 1
                y2 = t + 1

Return x1, x2, y1, y2
```

### Correctness

1. We initialize the bounds (x1, y1, x2, y2) to -1 and the maximum size of the square (max_size) to 0. This ensures that if no valid square area is found during the execution, the result will be consistent with the problem statement.
2. The algorithm iterates over all possible top-left corners (i, j) and bottom-right corners (s, t) of the squares. It checks if the current rectangle formed by the corners (i, j) and (s, t) is a square. If not, it continues to the next iteration.
3. For each valid square formed by corners (i, j) and (s, t), the algorithm counts the number of plots that have a minimum tree requirement of less than h.
4. If the count of invalid plots is less than equal to k and the current square size is greater than max_size, the algorithm updates the max_size and the bounding indices of the square (x1, x2, y1, y2).
5. The algorithm terminates after all possible top-left corners (i, j) and bottom-right corners (s, t) have been considered. At this point, if a valid square area has been found, the algorithm will have updated the max_size and the bounding indices of the square (x1, x2, y1, y2). If no valid square area is found, the max_size will remain 0, and the bounding indices will not be updated.

### Time complexity

Combining all 6 loops, the overall time complexity of the algorithm is $\Theta(mn * mn * mn) = \Theta(m^3n^3)$.

### Space complexity

The space complexity of the algorithm is $O(1)$ since we are just storing the bounds and not using any other data structure.

Task 7A (Dynamic Programming)

### Design

*Initialize the bounds(x1, y1, x2, y2) of square to be -1*
*Create a dp array of size (m + 1) * (n + 1)*
*Keep track of maximum size of square(max_size)*

*Create a recursive function(helper) with parameters i, j (size of plot)*
*If i is 0 or j is 0:*
*Set dp[i][j] to 0*
*Return dp[i][j]*
*If dp[i][j] contains value:*
*Return dp[i][j]*
*If plot[i - 1] [j - 1] < h:*
*Set temp to 1 else 0*

*Set dp[i][j] to helper(i- 1, j) + helper(i, j - 1) - helper(i- 1, j - 1)) + temp*

*For x in 0 to min(i, j):*
*Set ans to dp[i][j] + helper(i - x - 1, j - x - 1) - helper(i - x - 1, j) - helper(i, j - x - 1)*

*If ans <= k and x + 1 > max_size:*
*max_size = x + 1*
*x1 = i - x*
*y1 = j - x*
*x2 = i*
*y2 = j*
*Return dp[i][j]*

*Call recursive fn helper(m, n)*
*Return x1, y1, x2, y2*

### Recursive Formulation

*The recurrence relation for the given algorithm can be defined as follows:*

*helper(i, j) = 0* *if i <= 0 or j <= 0*

*helper(i, j) = helper(i - 1, j) + helper(i, j - 1) - helper(i - 1, j - 1) + temp if i > 0 and j > 0*

*where temp = 1 if p[i-1][j-1] < h, and temp = 0 otherwise.*

### Correctness

The algorithm uses a memoization table to store the number of plots that have a minimum tree requirement of less than h within a given rectangular area up to (i, j). The memoization table is filled using the previously computed values, ensuring that the optimal substructure property is met. The algorithm iterates over all possible square sizes for each (i, j) and checks if the number of such plots within the square is less than or equal to k. If a larger square with the given constraints is found, the maximum size and the bounding indices are updated. This guarantees that the largest square satisfying the conditions will be found.

### Time complexity

The time complexity is determined by the number of unique subproblems (m * n) multiplied by the time taken for each subproblem. Since each subproblem calls a loop which is min(i, j) ~k outside of the recursive calls, and memoization ensures that each subproblem is only computed once, the time complexity of the algorithm is Θ(m * n * k).

### Space complexity

The space complexity of the algorithm is O(mn) which is used by the dp array

## Task 7B (Dynamic Programming - Iterative)

### Design

*Initialize the bounds(x1, y1, x2, y2) of square to be -1*
*Create a dp array of size (m + 1) * (n + 1)*
*Keep track of maximum size of square(max_size)*

```
For i in 1 to m + 1:
    For j in 1 to n + 1:
            If plot[i - 1] [j - 1] < h:
                    Set temp to 1
```

```
        Else:
                Set temp to 0

        Set dp[i][j] to  dp(i- 1, j) + dp(i, j - 1) - dp(i- 1, j - 1)) + temp
        Set t to min(i, j)

       For x in 0 to t:
            Set ans to dp[i][j] + dp[i - x-1][j - x-1] - dp[i - x-1][j] - dp[i][j -
    x-1]
            If ans <= k and x + 1 > max_size:
                    max_size = x + 1
                    x1 = i - x
                    y1 = j - x
                    x2 = i
                    y2 = j

Return x1, y1, x2, y2
```

## Correctness

1. The algorithm aims to find the largest square area within a given matrix, such that the number of elements less than h within the square is less than or equal to k. The algorithm uses dynamic programming to store the number of elements less than h for each submatrix ending at (i, j). The dynamic programming relation is:

   dp[i][j] = dp[i - 1][j] + dp[i][j - 1] - dp[i - 1][j - 1] + temp

   where temp = 1 if matrix[i - 1][j - 1] < h, and temp = 0 otherwise.

2. The algorithm iterates over each cell of the matrix and then iterates over all possible square sizes (from 1 to min(i, j)) ending at the current cell (i, j). It calculates the number of elements less than h within the square using the DP table and checks if it's less than or equal to k. If a larger valid square is found, the maximum size and the bounding indices are updated. This ensures that the largest square satisfying the given constraints will be found.

## Time complexity

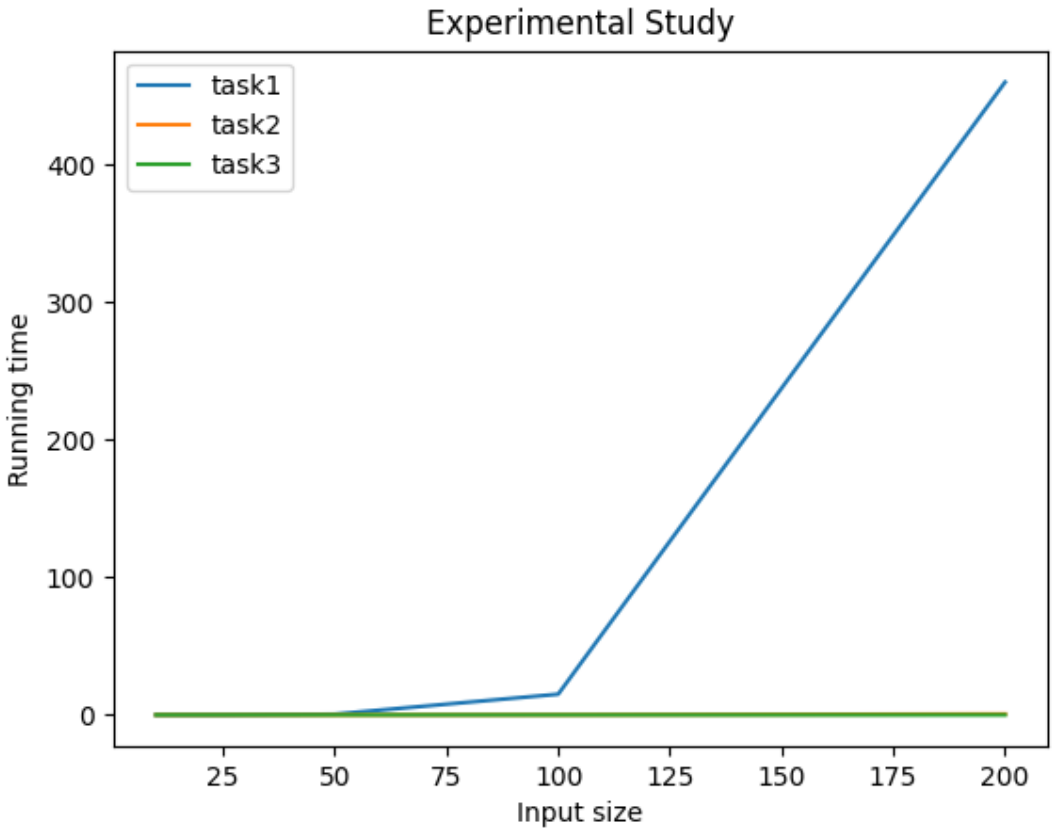
Outermost loop iterates through the plot size (m * n). Inner loop iterates through min(i, j) which is equivalent to k. So,  the time complexity of the algorithm is Θ(m * n * k).

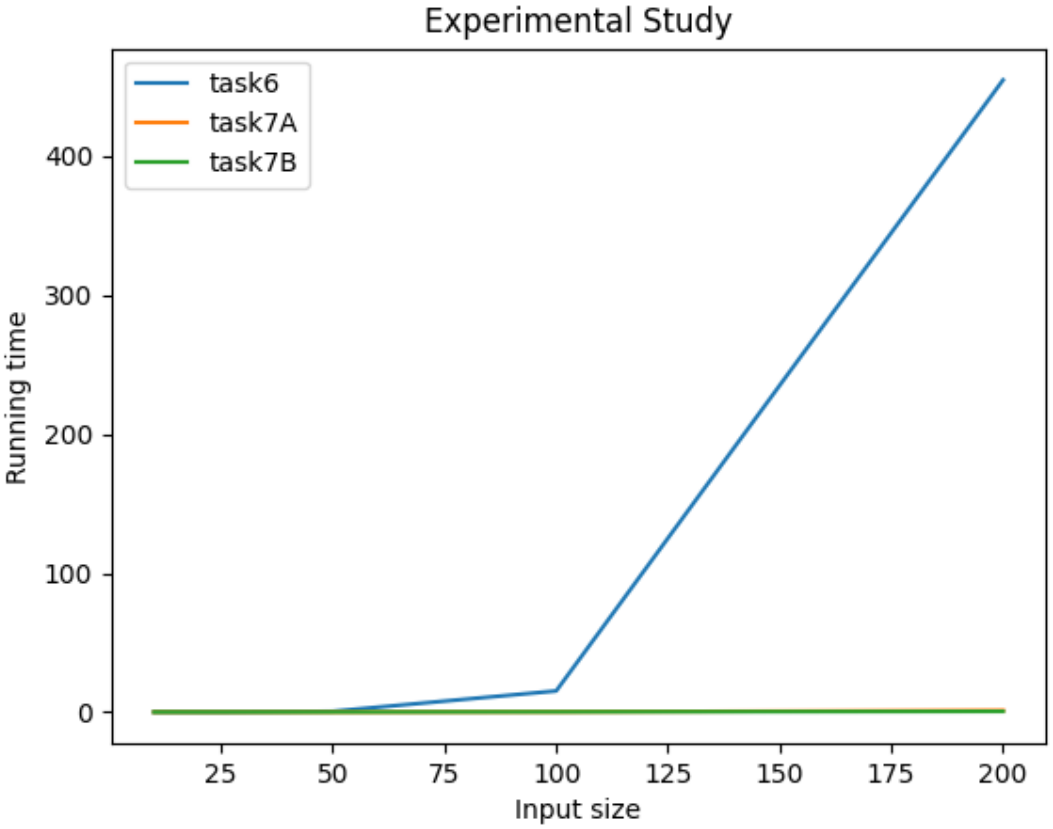
### Space complexity

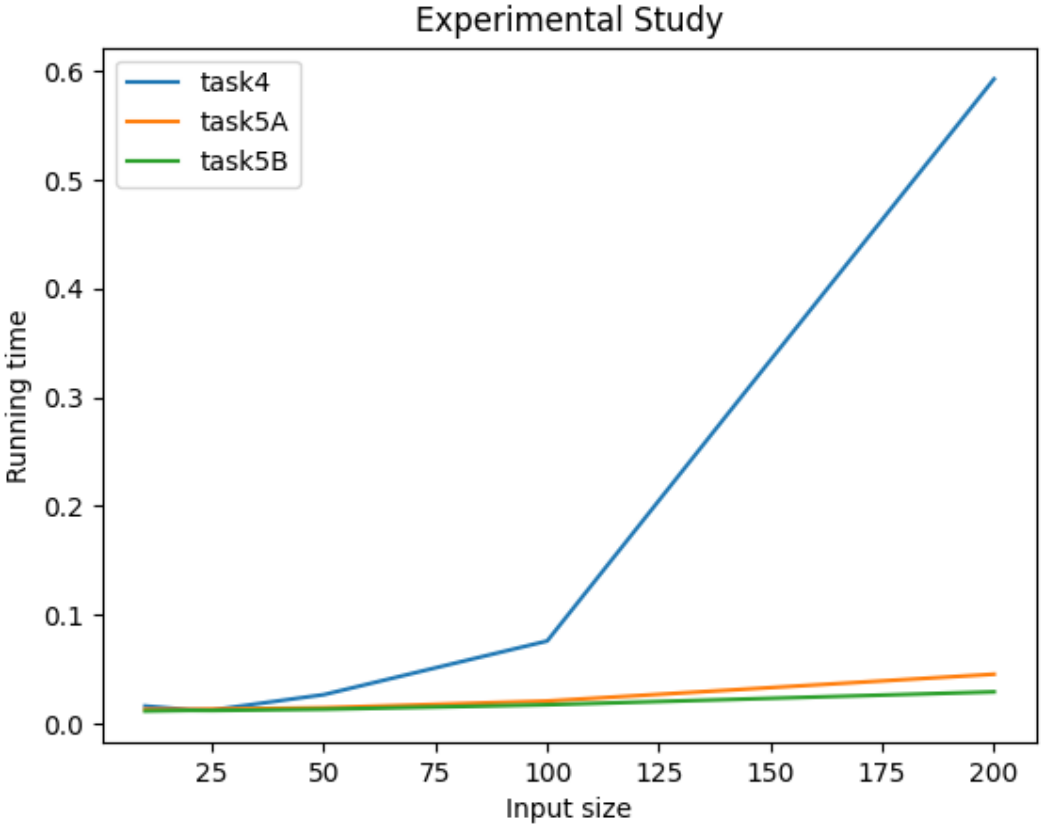
The space complexity of the algorithm is O(mn) which is used by the dp array

# Experimental Comparative Study

We have randomly generated test files using input size as 10, 25, 50, 100. We use these input sizes to generate random values of m, n and plot matrix. Here are the plots showing the comparisons between (Task 1, Task 2, Task 3), (Task 4, Task 5A, Task 5B) and (Task 6, Task 7A, Task 7B)




Experimental Study
task6
task7A
task7B
Running time
0
100
200
300
400
25
50
75
100
125
150
175
200
Input size


Experimental Study
task4
task5A
task5B
Running time
0.0
0.1
0.2
0.3
0.4
0.5
0.6
25
50
75
100
125
150
175
200
Input size

**Data**

| Task | I/P Size(10) | I/P Size(25) | I/P Size(50) | I/P Size(100) | I/P Size (200) |
|---|---|---|---|---|---|
| Task 1 | 0.0118 | 0.035 | 0.569 | 15.074 | 460.13 |
| Task 2 | 0.0115 | 0.01274 | 0.0220 | 0.0927 | 0.69614 |
| Task 3 | 0.0123 | 0.0125 | 0.01237 | 0.0142 | 0.0219 |

| Task | I/P Size(10) | I/P Size(25) | I/P Size(50) | I/P Size(100) | I/P Size (200) |
|---|---|---|---|---|---|
| Task 4 | 0.016 | 0.0124 | 0.0265 | 0.0758 | 0.5927 |
| Task 5A | 0.0130 | 0.01317 | 0.0145 | 0.0206 | 0.0453 |
| Task 5B | 0.0114 | 0.0119 | 0.0130 | 0.01723 | 0.02910 |

| Task | I/P Size(10) | I/P Size(25) | I/P Size(50) | I/P Size(100) | I/P Size(200) |
|---|---|---|---|---|---|
| Task 6 | 0.0147 | 0.0359 | 0.5604 | 15.2066 | 454.9062 |
| Task 7A | 0.0116 | 0.0132 | 0.0304 | 0.1567 | 1.1761 |
| Task 7B | 0.0142 | 0.0219 | 0.0124 | 0.1370 | 0.5013 |

Data and graph implies that among (Task1, Task2 and Task 3), Task 1 takes the longest time due to O(m^3n^3) time complexity. Among tasks (Task4, Task5A and Task 5B), Task 5B takes the shortest time(O(mn)) while Task 4 takes the longest. Task 5B takes less time than 5A since it doesn't use recursion.

Among (Task6, Task7A and Task 7B), Task 7B takes the shortest time (O(mn)) and task 6 the longest O(m^3n^3).

# Conclusion

To summarize the learning experience, we both learned a lot while working on this project. One of us is proficient in Java while the other is proficient in C++, so we chose a middle ground and decided to choose Python as our main language to allow both of us to learn and benefit from the process. While implementing the tasks, Task 1 was straightforward to implement while other tasks required us to do a bit of research to figure out the algorithm. It took us a significant amount of time to come up with the algorithm for tasks 5 and 7.