COT5405 Algorithms Programming Project II

PROJECT REPORT
Dynamic Programming

Analysis of Algorithms (COT5405)

Team Members:
Name: Aadithya Kandeth
UFID: 6980-2791
Group - 24

Table of Contents

Design and Analysis of Algorithms

Defining the Scenario:

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 Green Hands 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 Problem 1 is a special case of Problem 2 and Problem 3.

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.

Alg 1: Design a Θ(m3n3) time Brute Force algorithm for solving Problem1

Task 1: Give an implementation of Alg1.

Strategy to be implemented:

Iterate over every possible submatrix using loops to go through all rows and columns. If the submatrix is a square (the number of rows is equal to the number of columns), check if the submatrix has any elements that are lesser than h which would make it invalid. Return the bounding indices of the largest valid square submatrix found.

Algorithm:

Function find_p1_m3n3:

1. Take in the parameters p and h.
2. Get the dimensions of the matrix p.
3. Initialize max_size to 0 and top_left and bottom_right to (0, 0).
4. Iterate over all possible submatrices of p that are square, meaning that they have the same number of rows and columns.

5. For each submatrix, check if it is valid by calling the valid_submatrix1 function with the submatrix's top-left and bottom-right coordinates.
6. If the submatrix is valid, calculate its size as (r2 - r1 + 1) * (c2 - c1 + 1) where (r1, c1) and (r2, c2) are the submatrix's top-left and bottom-right coordinates, respectively.
7. If the size is greater than max_size, update max_size, top_left, and bottom_right with the size and the coordinates of the submatrix's top-left and bottom-right corners, respectively.
8. After iterating over all possible submatrices, return the coordinates of the top-left and bottom-right corners of the largest valid submatrix in the form of a tuple (r1, c1, r2, c2).

Function valid_submatrix1:

1. Take in the parameters p, r1, c1, r2, c2, and h.
2. Iterate over the rows from r1 to r2 inclusive.
3. For each row, iterate over the columns from c1 to c2 inclusive.
4. If the element at the current row and column in p is less than h, return False.
5. If the iteration completes without finding an element less than h, return True.

Pseudocode:

```
Function valid_submatrix1(p, r1, c1, r2, c2, h):

  1. Iterate over the rows from r1 to r2 inclusive.
  2. For each row, iterate over the columns from c1 to c2 inclusive.
  3. If the element at the current row and column in p is less than h,
     return False.
  4. If the iteration completes without finding an element less than h,
     return True.

Function find_p1_m3n3(p, h):

  1. Get the dimensions of the matrix p.
  2. Initialize max_size to 0 and top_left and bottom_right to (0, 0).
  3. Iterate over all possible submatrices of p that are square, meaning
     that they have the same number of rows and columns.
  4. For each submatrix, check if it is valid by calling the
     valid_submatrix1 function with the submatrix's top-left and bottom-
     right coordinates.
  5. If the submatrix is valid, calculate its size as (r2 - r1 + 1) *
     (c2 - c1 + 1) where (r1, c1) and (r2, c2) are the submatrix's top-
     left and bottom-right coordinates, respectively.
```

6. If the size is greater than max_size, update max_size, top_left, and bottom_right with the size and the coordinates of the submatrix's top-left and bottom-right corners, respectively.
7. After iterating over all possible submatrices, return the coordinates of the top-left and bottom-right corners of the largest valid submatrix in the form of a tuple (r1, c1, r2, c2).

Proof of Correctness:

The algorithm correctly identifies the largest square submatrix with all elements greater than or equal to h in Θ(m^3n^3) time. It does this by iterating over all possible square submatrices of p and checking if they are valid, i.e., all their elements are greater than or equal to h. This is because the valid_submatrix1 function iterates over all elements of the submatrix and returns false if it finds any element less than h. Otherwise, it returns true. Therefore, if the function returns true, we know that all elements of the submatrix are greater than or equal to h. Therefore, every possible square submatrix of p will be considered by the algorithm.It keeps track of the largest valid submatrix seen so far and returns its top left and bottom right indices at the end. The correctness of the algorithm follows from the fact that it considers all possible square submatrices and correctly identifies valid and largest submatrices.This is of course, not the optimal solution since it is a brute force approach.

Time Complexity:

The valid_submatrix1 function has a time complexity of O((r2 - r1 + 1) * (c2 - c1 + 1)) since it visits every element within the submatrix only once. In contrast, the find_p1_m3n3 function has a time complexity of O(m^3 * n^3) due to its iteration over all possible top-left corner coordinates of submatrices of p. The submatrix is square, meaning its number of rows equals its number of columns, and for each top-left corner coordinate, the function loops through all possible bottom-right corner coordinates while ensuring that r2 >= r1 and c2 >= c1. Consequently, the overall time complexity of the code is O(m^3 * n^3). This is of course, not the optimal solution since there is a better dynamic programming based approach.

Space Complexity:

The space complexity of the code would depend on the size of the matrices being used. The p matrix has a size of m x n, the valid matrix has a size of (m-r+1) x (n-c+1), and the submatrix matrix has a size of (r2-r1+1) x (c2-c1+1). Therefore, the space complexity of the code would be O(mn + (m-r+1)(n-c+1) + (r2-r1+1)(c2-c1+1)), which simplifies to O(mn + m^2n^2).

In the worst case, when m and n are large, the space complexity could become significant. However, it should be noted that the space complexity of this code is still polynomial and not exponential in the size of the input, which means that it can handle reasonably large inputs.

<u>Alg 2: Design a Θ(m2n 2) time algorithm for solving Problem1</u>

<u>Task 2: Give an implementation of Alg2.</u>

Strategy to be implemented:

Preprocess the matrix p to create a 2D matrix count that stores the count of elements greater than or equal to h, and then iterate over all possible square submatrices of p and check their validity using this count matrix. The validity is found by comparing the number of elements in the submatrix formed using the count matrix against the number of elements across the rows and columns in the submatrix. If a valid square submatrix is formed, the indices of the largest one is returned.

Algorithm:

Function preprocess(p, h)

1. Get the dimensions of the input matrix p.
2. Initialize a new matrix count with dimensions (m+1) x (n+1) where each element is 0.
3. Iterate over the rows and columns of count starting from 1.
4. For each cell (i, j) in count, set it to the sum of the cells (i-1, j), (i, j-1), and (i-1, j-1) in count, plus 1 if the corresponding element in p is greater than or equal to h.
5. Return the count matrix.

Function valid_submatrix(count, r1, c1, r2, c2)

1. Calculate the sum of elements in the submatrix defined by (r1, c1) as the top-left corner and (r2, c2) as the bottom-right corner in the count matrix.
2. Subtract the sums of the cells to the left, above, and above-and-to-the-left of the submatrix to get the sum of the submatrix itself.
3. Check if the sum of the submatrix is equal to the expected sum, (r2 - r1 + 1) * (c2 - c1 + 1).
4. Return True if the sum of the submatrix is correct, and False otherwise.

Function find_p1_m2n2(p, h)

1. Get the dimensions of the input matrix p.
2. Initialize variables max_size, top_left, and bottom_right to 0 and (0, 0), respectively.
3. Preprocess the input matrix p using the preprocess function to create a new matrix count.
4. Iterate over all possible submatrices of p that are square, meaning that they have the same number of rows and columns. This is done using four nested loops over the top-left and bottom-right corners of the submatrix.
5. For each submatrix, check if it is valid by calling the valid_submatrix function with the submatrix's top-left and bottom-right coordinates and the count matrix.
6. If the submatrix is valid, calculate its size as (r2 - r1 + 1) * (c2 - c1 + 1) where (r1, c1) and (r2, c2) are the submatrix's top-left and bottom-right coordinates, respectively.
7. If the size is greater than max_size, update max_size, top_left, and bottom_right with the size and the coordinates of the submatrix's top-left and bottom-right corners, respectively.
8. After iterating over all possible submatrices, return the coordinates of the top-left and bottom-right corners of the largest valid submatrix in the form of a tuple (r1, c1, r2, c2).

Pseudocode:

```
Function preprocess(p, h)

   1. Get the dimensions of the input matrix `p`.
   2. Initialize a new matrix `count` with dimensions `(m+1) x (n+1)`
      where each element is 0.
   3. Iterate over the rows and columns of `count` starting from 1.
   4. For each cell `(i, j)` in `count`, set it to the sum of the cells
      `(i-1, j)`, `(i, j-1)`, and `(i-1, j-1)` in `count`, plus 1 if the
      corresponding element in `p` is greater than or equal to `h`.
   5. Return the `count` matrix.
```

```
Function valid_submatrix(count, r1, c1, r2, c2)

   1. Calculate the sum of elements in the submatrix defined by `(r1,
      c1)` as the top-left corner and `(r2, c2)` as the bottom-right
      corner in the `count` matrix.
   2. Subtract the sums of the cells to the left, above, and above-and-
      to-the-left of the submatrix to get the sum of the submatrix itself.
   3. Check if the sum of the submatrix is equal to the expected sum,
      `(r2 - r1 + 1) * (c2 - c1 + 1)`.
```

4. Return `True` if the sum of the submatrix is correct, and `False` otherwise.

Function find_p1_m2n2(p, h)

1. Get the dimensions of the input matrix `p`.
2. Initialize variables `max_size`, `top_left`, and `bottom_right` to 0 and `(0, 0)`, respectively.
3. Preprocess the input matrix `p` using the `preprocess` function to create a new matrix `count`.
4. Iterate over all possible submatrices of `p` that are square, meaning that they have the same number of rows and columns. This is done using four nested loops over the top-left and bottom-right corners of the submatrix.
5. For each submatrix, check if it is valid by calling the `valid_submatrix` function with the submatrix's top-left and bottom-right coordinates and the `count` matrix.
6. If the submatrix is valid, calculate its size as `(r2 - r1 + 1) * (c2 - c1 + 1)` where `(r1, c1)` and `(r2, c2)` are the submatrix's top-left and bottom-right coordinates, respectively.
7. If the size is greater than `max_size`, update `max_size`, `top_left`, and `bottom_right` with the size and the coordinates of the submatrix's top-left and bottom-right corners, respectively.
8. After iterating over all possible submatrices, return the coordinates of the top-left and bottom-right corners of the largest valid submatrix in the form of a tuple `(r1, c1, r2, c2)`.

Proof of Correctness:

The find_p1_m2n2 function finds the largest square submatrix of p whose elements are greater than or equal to h. This is achieved by iterating over all possible square submatrices of p and checking whether they are valid using the valid_submatrix function. The function maintains the maximum size and coordinates of the largest valid submatrix found so far.

The correctness of the code follows from the fact that a submatrix of p is a valid solution if and only if it is square and all its elements are greater than or equal to h. The preprocess function efficiently computes the number of elements greater than or equal to h in any submatrix of p, which allows the valid_submatrix function to efficiently check whether a given submatrix is valid. The find_p1_m2n2 function then iterates over all possible square submatrices of p and uses valid_submatrix to check whether they are valid. Since the algorithm checks all possible submatrices, it is guaranteed to find the largest valid submatrix of p.

Time Complexity:

The preprocess function takes O(mn) time to create a 2D list count. The valid_submatrix function takes O(1) time, as it is just performing constant-time arithmetic operations. The nested loops in find_p1_m2n2 iterate over all possible submatrices of p whose dimensions are powers of 2, which is O(m^2n^2) iterations in total. For each submatrix, valid_submatrix is called with the count list and the indices of the top-left and bottom-right corners of the submatrix, which takes O(1) time. Therefore, the time complexity of the nested loops is O(m^2n^2). The preprocess function and the nested loops are called once each, so the overall time complexity of the code is O(mn + m^2n^2) = O(m^2n^2). The time complexity can still be improved using dynamic programming to become O(m*n).

Space Complexity:

The space complexity of the preprocess function is O(mn) because it creates a 2D list of size (m+1) x (n+1). The space complexity of the valid_submatrix function and the find_p1_m2n2 function is O(1) because they use only constant extra space. Therefore, the overall space complexity of the code is O(mn).

Alg 3: Design a Θ(mn) time Dynamic Programming algorithm for solving Problem1

Task 3: Give an implementation of Alg3

Strategy to be implemented:

Use dynamic programming to compute the size of the largest square submatrix in the input matrix p that contains only elements greater than or equal to the threshold h. The algorithm iterates over all cells of p, and for each cell that meets the threshold condition, it computes the size of the largest square submatrix that ends at that cell (The current cell being the bottom left corner of the submatrix). The algorithm updates a dp table with the computed submatrix sizes and keeps track of the largest submatrix found so far. Finally, the algorithm returns the coordinates of the largest submatrix found or None if no such submatrix exists.

Recurrence Relation:

For row and column (i,j),
If i or j = 0
OPT[i][j] = 1
Otherwise,
OPT[i][j] = min(OPT[i-1][j], OPT[i][j-1], OPT[i-1][j-1]) + 1

Algorithm:

1. Get the dimensions of the input matrix `p`.
2. Initialize a new matrix `dp` with dimensions `m x n` where each element is 0.
3. Initialize variables `max_size`, `top`, `left`, `bottom`, and `right` to 0 and -1.
4. Iterate over all cells `(i, j)` in `p`.
5. If the value of `p[i][j]` is greater than or equal to `h`:
6. Set the value of dp of that cell to 1 if i or j equals 0.
7. Otherwise, set the dp value of the cell to the minimum of its neighbours(top, left and top left diagonal) incremented by 1.
8. If the dp value is greater than the current maximum size, update `max_size`, `top`, `left`, `bottom`, and `right` with the size and the coordinates of the bounding indices, respectively.
9. If `max_size` is 0, return `None`.
10. Otherwise, return the bounding indices.

Pseudocode:

```
Initialize variables:
  m, n = dimensions of matrix p
  dp = a new matrix of size m x n, initialized with 0
  max_size = 0, top = left = bottom = right = -1
For i iterating from 0 to m-1:
 For j iterating from 0 to n-1:
   If p[i][j] >= h:
     If i == 0 or j == 0, set dp[i][j] = 1
   Otherwise, set dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1
      If dp[i][j] > max_size, set max_size = dp[i][j], top = i-max_size+1, left = j-max_size+1, bottom = i+1, right = j+1
    If max_size == 0, return None
    Otherwise, return the tuple (top, left, bottom, right)
```

Proof of Correctness:

To prove the correctness of the above program using induction, we need to show that:

- The base case is correct, i.e., the program correctly computes the maximum size of the submatrix with bottom-right corner (0,0).

- If the program correctly computes the maximum size of the submatrix with bottom-right corner (i-1,j-1), then it also correctly computes the maximum size of the submatrix with bottom-right corner (i,j).

<u>Base Case:</u>

When i=0 and j=0, there is only one element in the matrix p. The program initializes dp[0][0] to 1 if p[0][0]>=h, else it is initialized to 0.

This is correct because the maximum size of the submatrix with bottom-right corner (0,0) can be either 1 (if p[0][0]>=h) or 0 (if p[0][0]<h), which is what the program computes.

<u>Inductive Step:</u>

Suppose the program correctly computes the maximum size of the submatrix with bottom-right corner (i-1,j-1) for all i,j such that i+j <= k, where k>0. We need to show that the program also correctly computes the maximum size of the submatrix with bottom-right corner (i,j) for all i,j such that i+j = k.

If p[i][j]>=h, then the maximum size of the submatrix with bottom-right corner (i,j) is the minimum of the sizes of the submatrices with (i-1,j), (i,j-1), and (i-1,j-1) as their bottom-right corners, plus one. Let S be the maximum size of the submatrix with bottom-right corner (i,j) computed by the program, and let S' be the actual maximum size of the submatrix with bottom-right corner (i,j). By the induction hypothesis, the program correctly computes the maximum size of the submatrix with bottom-right corner (i-1,j-1). Therefore, if S is computed correctly, then S' is at least S.

Now, suppose S' > S. But this contradicts the assumption that S is the maximum size of the submatrix with bottom-right corner (i,j) computed by the program, because S would be equal to S'+1 in this case. Therefore, S' cannot be greater than S, and hence the program correctly computes the maximum size of the submatrix with bottom-right corner (i,j). By mathematical induction, the program correctly computes the maximum size of the submatrix with bottom-right corner (i,j) for all i,j, and hence returns the correct output.

Time Complexity:

The time complexity of the above program is Θ(mn), where m and n are the number of rows and columns in the input matrix, respectively. This is because the program loops through each element in the matrix exactly once and performs a constant amount of work (computing the minimum of three values, updating the maximum size, and updating the top, left, bottom, and right indices) for each element. Therefore, the time complexity of the program is proportional to the number of elements in the matrix, which is mn. Hence, the time complexity of the program is Θ(mn).

Space Complexity:

The space complexity of the above program is O(mn), where m and n are the number of rows and columns in the input matrix, respectively. This is because the program creates a 2D list (dp) of size m x n to store the dynamic programming table. Additionally, the program uses a constant amount of extra space to store the variables max_size, top, left, bottom, and right, which do not depend on the size of the input matrix.

Therefore, the total space used by the program is proportional to the size of the input matrix, which is mn. Hence, the space complexity of the program is 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

Alg4: Design a Θ(mn2) time Dynamic Programming algorithm for solving Problem2

Task4: Give an implementation of Alg4.

Strategy to be implemented:

The objective is to create a function that returns the bounding indices of the largest square submatrix where all elements except the corners have to be greater than or equal to h.. Two dp matrices are used to keep track of the largest square with h condition satisfied and the largest column with h condition satisfied. Then, the function loops through the matrix to calculate the largest square that satisfies all elements greater than h , and uses this information to calculate the largest square that has elements except corners greater than or equal to h. Finally, the function returns the bounding indices of the resultant square.

Recurrence Relation:

The dp matrices are filled using the following recurrence relation.

Main DP matrix:

```
if i - 1 less than  0 or j - 1 less than 0:
        OPT[i][j] = 1
Else:
        OPT[i][j] = min(OPT[i - 1][j - 1], min(OPT[i - 1][j], OPT[i][j - 1])) + 1
```

<u>Columns DP Matrix:</u>

```
if matrix[i][j] >= h:
if i equals 0:
            OPT[i][j] = 1
else:
            OPT[i][j] = OPT[i - 1][j] + 1
```

Algorithm:

1. Define a function that takes a matrix and a height value as input.
2. Initialize rows and cols variables to the number of rows and columns in the matrix, respectively.
3. Create two dynamic programming matrices, colDP and squareDP, of size rows * cols filled with zeros.
4. Loop through the matrix:
5. If the value of the current element is greater than or equal to h, update the current cell in the squareDP matrix with the min of its top left and top left neighbours squareDP value incremented by 1.
6. Similarly, If the value of the current element is greater than or equal to h, update the colDP matrix with the colDP value of its top neighbour incremented by 1 to store the max value for that column.
7. Initialize variables max_side to 2 and max_i and max_j to 1.
8. Loop through the matrix starting from the second row and second column:
9. Compute the length of the bottom row of the largest square that includes the current element.
10. Compute the length of the largest square with minimum height value that includes the current element.
11. Compute the length of the final side of the largest square that has a minimum height value of h and includes the current element.
12. If the final side is greater than max_side, update max_side, max_i, and max_j.
13. Return a tuple of four values: the row and column indices of the top-left corner and the bottom-right corner of the largest square that has a minimum height value of h in the matrix.

Pseudocode:

```
1. Define a function find_p2_dp_mn2(p,h)
2. Initialize two2D lists: colDP and squareDP.
```

```
3. For i from 1 to rows
    For j from 1 to cols
      If value>=h,
        Update squareDP value as min of its neighbors + 1
4. For i from 1 to rows
    For j from 1 to cols
      If value>=h,
        Update colsDP value as its top + 1
5. Initialize maximum side length = 2, maximum i-coordinate = 1, and
   maximum j-coordinate = 1
6. For i from 1 to rows
    For j from 1 to cols
      For each element, calculate the maximum square sub-matrix that
   includes the current element using squareDP and colsDP values.
7. If length of left,bottom side of the square < length of square,
     Final_square_sidelength = 2 + length of square
8. If length of left,bottom side of the square >= length of square,,
   set top side of square based on previous row and set it to final
   side length if the conditions are satisfied
   Else,
     Final_square_sidelength = 2 + length of square.
9. If final side length > current maximum side length
    update maximum side length, maximum i-coordinate, and maximum j-
   coordinate.
10.     Return top left and bottom right coordinates
```

Proof of Correctness:

We can prove the correctness of this algorithm by showing that it satisfies the two properties of dynamic programming:

- Optimal Substructure
- Overlapping Subproblems.

Optimal Substructure:

In the given algorithm, we can see that the maximum size of the largest square submatrix at any given element (i, j) is dependent on the maximum sizes of the largest square submatrices at the elements (i-1, j), (i, j-1), and (i-1, j-1).The squareDP array computes the size of the largest square submatrix ending at (i, j) that includes the element (i, j). It does this by checking the values of its three neighbors (i-1, j), (i, j-1), and (i-1, j-1), and adding 1 to the minimum of these three neighbors if the element (i, j) is greater than or equal to the threshold h. This ensures that the size of the largest square submatrix at (i, j) is optimal, as it is based on the optimal solutions of its subproblems. Similarly, the colDP array computes the length of the longest column of consecutive elements greater than or equal to the threshold h ending at (i, j). This is used to calculate the maximum possible side length of the submatrix by taking the minimum of the longest column and the longest row of consecutive elements greater than or equal to h.

Overlapping Subproblems:

The squareDP array is computed by checking the values of its three neighbors (i-1, j), (i, j-1), and (i-1, j-1) to compute the size of the largest square submatrix ending at (i, j). These three neighbors are subproblems that have already been solved, and their solutions are reused to compute the solution at (i, j). Similarly, the colDP array is computed by checking the value of its neighbor (i-1, j) to compute the length of the longest column of consecutive elements greater than or equal to the threshold h ending at (i, j). This neighbor is a subproblem that has already been solved, and its solution is reused to compute the solution at (i, j).

Hence, the given algorithm satisfies the properties of dynamic programming, and its correctness is proven.

Time Complexity:

The time complexity of the algorithm is dominated by the two nested for loops that iterate over all elements of the matrix p. Therefore, the time complexity is O(mn). For each element, we perform constant time operations, including updating values in the squareDP and colDP arrays, computing the size of the largest submatrix with bottom right corner at the current element, and updating the maximum submatrix found so far. The computation of the largest submatrix involves iterating over the elements of the matrix p in a diagonal direction and taking the minimum of the colDP and bottom_row values, which takes O(min(m, n)) time. Therefore, the overall time complexity of the algorithm is O(mn^2). Since we drop lower order terms and constants in the big O notation, the time complexity of the algorithm is Theta(mn^2).

Space Complexity:

The space complexity of the program is O(mn), since it uses two matrices of size mxn, namely colDP and squareDP. Additionally, it uses constant space for other variables such as rows, cols, max_side, max_i, and max_j. Therefore, the overall space complexity is O(mn).

Alg 5: Design a Θ(mn) time Dynamic Programming algorithm for solving Problem2

Task5a: Give a recursive implementation of Alg5 using Memoization

Strategy to be implemented:
The objective is to find the largest square sub-matrix with elements greater than or equal to a given height value 'h' except the corners. Use dynamic programming with memoization and a top down recursive approach to avoid recalculating the same subproblems. Use a recursive function to to calculate the size of the largest square sub-matrix that includes the current element. Then loop through all elements in the matrix, calculates the size of the largest square sub-matrix that includes each element using the recursive function, and update the maximum square size and the coordinates of the top-left and bottom-right corners of the largest square sub-matrix if a larger square sub-matrix is found that satisfies both the required conditions. Finally, return the coordinates of the top-left and bottom-right corners of the largest square sub-matrix.

Recurrence Relation:

```
If p[i][j]<h:
   OPT[i][j] = min(rec(i-1, j), rec(i, j-1), rec(i-1, j-1))
If p[i][j]>=h and top, left, top left diagonal neighbors >= h
   OPT[i][j] = min(rec(i-1, j), rec(i, j-1), rec(i-1, j-1)) + 1
```

Algorithm:

1. Initialize m and n as the number of rows and columns in the matrix p.
2. Initialize a 2D dp matrix with -1 for all elements.
3. Initialize the maximum square size, and the coordinates of the top-left and bottom-right corners of the largest square sub-matrix with elements greater than or equal to h to 0.
4. Define a nested function that performs recursion and memoization to calculate the maximum size of the largest square sub-matrix with elements greater than or equal to h that includes the element at position (i,j) in the matrix p.
5. Loop through all elements in the matrix, starting from the second row and column.
6. For each element, call the recursive function to calculate the maximum size of the largest square sub-matrix with elements greater than or equal to h that includes the current element.
7. Update the maximum square size and the coordinates of the top-left and bottom-right corners of the largest square sub-matrix if a larger square sub-matrix is found.

8. Return the coordinates of the top-left and bottom-right corners of the largest square sub-matrix with elements greater than or equal to h in the matrix p.

Pseudocode:

```
1. Set m and n to be the number of rows and columns in the matrix p.
2. Create a 2D list called dp, initialize all elements to -1.
3. Initialize max_size, top, left, bottom, and right to be 0.
4. Define a recursive function called "rec" that takes two parameters
   i and j, representing the current position in the matrix.
    If i = 0 or j = 0, return 1
    If dp[i][j] != -1, return dp[i][j].
    Recursively call function rec on elements (i-1,j), (i,j-1), and
   (i-1,j-1), set their values to up, left, and up_left, respectively.
    If p[i][j] >= h, and all adjacent elements >= h, set dp[i][j] to
   the minimum of up, left, and up_left plus 1.
    Otherwise, set dp[i][j] to the min(up, left, up_left)
5. Return dp[i][j].
6. For i from 2 to rows, for j from 2 to cols
7. Call the rec function on the current element, and store the result
   in "size".
8. If size>max_size,
    update "max_size", "top", "left", "bottom", and "right"
9. return bounding indices.
```

Proof of Correctness:

To prove the correctness of the algorithm, we will use induction. Let n be the size of the input grid (i.e., the number of rows times the number of columns).

Base case: n = 1

When n = 1, there is only one cell in the grid, so the algorithm will simply return that cell if its height is greater than or equal to h, and none otherwise. This is correct since the maximum submatrix with height h in a 1x1 grid is either the cell itself (if its height is >= h) or an empty submatrix (if its height is < h).

Induction hypothesis:

Assume that the algorithm is correct for all grids of size k <= n.

Inductive step:

For a cell in the ith row and jth column, if the height of that cell and its four adjacent cells is at least h, then it is possible to form a square with the current cell as the top right corner. In this case, the size of the square is one more than the minimum of the sizes of the squares that can be formed from the three adjacent cells. If the height of the current cell and its four adjacent cells is less than h, then it is not possible to form a square with the current cell as the top right corner. In this case, the size of the square is equal to the minimum of the sizes of the squares that can be formed from the three adjacent cells. By following this approach, the algorithm correctly computes the size of the largest square in the matrix with a height greater than or equal to h.

Time Complexity:

The time complexity of the above program is indeed Θ(mn).The program first initializes a 2D array of size mxn, which takes O(mn) time. Then, for each cell in the matrix, it makes a constant number of checks and recursive calls. Since there are mn cells in the matrix, the total number of checks and recursive calls made is also O(mn). Therefore, the overall time complexity of the program is O(mn) + O(mn) = Θ(mn).

Space Complexity:

The space complexity of the above program is O(mn), as it uses a 2D array of size m x n to store the values of the subproblems. Additionally, the recursive function uses the call stack, which has a space complexity of O(m + n) in the worst case (when the entire matrix needs to be traversed). Therefore, the overall space complexity is O(mn + m + n) = O(mn).

Alg5: Design a Θ(mn) time Dynamic Programming algorithm for solving Problem2

Task5b: Give an iterative BottomUp implementation of Alg5

Strategy to be implemented:

Implement an iterative Bottom-Up dynamic programming algorithm to find the largest square of cells all of which have values greater than or equal to h except the corner elements. The algorithm initializes a two-dimensional DP array to store the solution to subproblems and iteratively fills it in by considering the values of the top, left and top-left neihbouring cells. Finally, the algorithm returns the bounding indices of the largest square submatrix.

Recurrence Relation:

```
if p[i][j] >= h and top, left, bottom and right neighbors >= h:
    OPT[i+1][j+1] = min(OPT[i][j], OPT[i+1][j], OPT[i][j+1]) + 1
```

Else:
    OPT[i+1][j+1] = min(OPT[i][j], OPT[i+1][j], OPT[i][j+1])

Algorithm:

1. Read input matrix p and the minimum height value h.
2. Initialize a 2D array dp of size mxn with all elements set to 0.
3. Initialize the maximum submatrix size max_size to 0 and the indices top, left, bottom, and right to -1.
4. For each row i in p, and for each column j in p:
   If i is the first row or j is the first column, set dp[i][j] to 1.
   If i is the second row or j is the second column, set dp[i][j] to 2.
   Else, if all the top, left, bottom, right neighbors of current element are greater than or equal to h, set dp[i+1][j+1] to the minimum of dp of neighbors incremented by 1; otherwise, set dp[i+1][j+1] to the minimum of dp of neighbors..
5. If dp[i+1][j+1] is greater than max_size, set max_size to dp[i+1][j+1], and set top, left, bottom, and right to the indices of the submatrix.
6. If max_size is 0, return None; otherwise, return bounding indices.

Pseudocode:

```
1. Initialize dp[m*n] = 0
2. initialize max_size, top, left, bottom, and right = 0.
3. For i = 0 to m-1 and j = 0 to n-1, do:
      a. If i = 0 or j = 0, set dp[i][j] to 1.
      b. Else if i = 1 or j = 1, set dp[i][j] to 2.
4. For i = 1 to m-2 and j = 1 to n-2, do:
      a. If p[i][j] >= h, p[i+1][j] >= h, p[i-1][j] >= h, p[i][j-1]
   >= h, and p[i][j+1] >= h, set dp[i+1][j+1] to min(dp[i][j],
   dp[i+1][j], dp[i][j+1]) + 1.
      b. Otherwise, set dp[i+1][j+1] to min(dp[i][j], dp[i+1][j],
   dp[i][j+1]).
      c. If dp[i+1][j+1] > max_size, set max_size to dp[i+1][j+1]
   and set top, left, bottom, and right accordingly.
5. Otherwise, return (top , left, bottom, right) if maxSize is not 0
```

Proof of Correctness:

The correctness of the program can be shown by induction. First, the initialization step of the dp array is correct, as it correctly sets the sizes of the squares that can be formed with the cells in the first row and first column of the matrix. Second, the update step of the dp array is correct, as it correctly computes the sizes of the squares that can be formed with the cells in the remaining rows and columns of the matrix. Specifically, if the height of the current cell and its four adjacent cells is greater than or equal to h, then the size of the square that can be formed is the minimum of the sizes of the squares that can be formed from the three adjacent cells plus one, as the current cell can be the top right corner of the square. Otherwise, the size of the square that can be formed is the minimum of the sizes of the squares that can be formed from the three adjacent cells, as the current cell cannot be the top right corner of a square. The program updates the dp array correctly based on these rules, and the final dp array correctly stores the sizes of the largest squares that can be formed with each cell as the top right corner. Finally, the program correctly finds the maximum size of the square found so far and returns the indices of the corresponding submatrix.

Time Complexity:

The time complexity of the above program is Θ(mn), where m and n are the dimensions of the input matrix p. This is because the program has two nested for-loops that iterate over all elements in the matrix, so the time complexity is proportional to the number of elements in the matrix. The first for-loop initializes the dp array, which takes Θ(mn) time as it iterates over all elements in the array. The second for-loop updates the dp array by checking the adjacent elements of each cell in the matrix, which also takes Θ(mn) time as it iterates over all elements in the array. Finally, the program determines the largest square with height greater than or equal to h by scanning the dp array, which takes Θ(mn) time as it iterates over all elements in the array. Therefore, the total time complexity of the program is Θ(mn), which is linear in the size of the input matrix.

Space Complexity:

The space complexity of the program is O(mn), since we are using a 2D array of size m x n to store the dynamic programming table. Additionally, we are using a constant amount of space to store the values of max_size, top, left, bottom, and right, so their contribution to the space complexity is negligible. Therefore, the dominant term in the space complexity is the size of the dp array, which 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 where only up to k enclosed plots can have a minimum tree requirement of less than h.

Alg6 Design a Θ(m3n 3) time Brute Force algorithm for solving Problem3

Task6: Give an implementation of Alg6.

Strategy to be implemented:

The objective is to find the largest square submatrix where all elements are greater than or equal to h but at most k elements can be less than h, given a matrix p and integers h and k. Use a brute force approach that iterates through all possible submatrices of p that are square and calculate the side length of each submatrix. Then count the number of cells in each submatrix that have a value less than h. If the count of such cells is less than or equal to k, the area of the submatrix is updated as the new maximum square and the indices of the submatrix are saved. Return the bounding indices of the largest submatrix that meets the criteria.

Algorithm:

1. Define a function that takes three arguments p, h and k.
2. Set max_area to 0 and indices to None.
3. Loop through all possible starting coordinates of the submatrices:
4. a. For each starting coordinate, loop through all possible ending coordinates of the submatrices
   b. If the ending coordinate is such that the submatrix is square, calculate its area.
   c. If the area is less than or equal to max_area, skip to the next iteration.
   d. Otherwise, count the number of elements in the submatrix that are less than h.
   e. If the count is less than or equal to k, update max_area and indices to reflect the new maximum area.
5. Return the indices of the submatrix with the largest area that satisfies the conditions.

Pseudocode:

```
1. Initialize max_area, indices = 0 ,None
```

```
2. Nested Loops
   a. row index i from 0 to m-1.
   b. column index j from 0 to n-1.
   c. row index x from i to m-1.
   d. column index y from j to n-1.
   e. If (x-i)==(y-j) it is a square.
   f. If not valid square:
      continue
   g. area of the submatrix = (x-i+1) * (y-j+1).
   h. If area <= max_area:
        continue
   i. Count the number of cells in the submatrix that have a value
   less than h.
   j. If count > k:
       continue
   k. Update max_area = area of the submatrix,
      Update bounding indices.
3. Return indices
```

Proof of Correctness:

To prove the correctness of this program, we need to show that it always returns the bounding indices of the largest square submatrix where all elements are greater than or equal to h, and at most k elements are less than h. First, the program iterates through all possible submatrices that are square, which ensures that no square submatrix is missed.

Next, the program checks whether the count of cells with a value less than h in each submatrix is less than or equal to k. This ensures that the program only considers submatrices that have at most k elements less than h. Then, the program updates the maximum area of the submatrix that meets the criteria and saves the indices of the submatrix. This ensures that the program returns the largest square submatrix where all elements are greater than or equal to h, and at most k elements are less than h. Finally, the program returns the bounding indices of the largest submatrix that meets the criteria. This ensures that the program returns the correct indices of the submatrix.

Therefore, the program correctly finds the bounding indices of the largest square submatrix where all elements are greater than or equal to h, and at most k elements are less than h.

Time Complexity:

The time complexity of the brute force algorithm is O(m^3 * n^3), since it involves iterating over all possible submatrices, and for each submatrix, iterating over all of its elements to count the number of elements less than h. The outer loop iterates over m * n elements, and the next two loops iterate over m * n elements again to find all possible submatrices. The innermost loop iterates over (x-i+1)*(y-j+1) elements for each submatrix to count the number of elements less than h.

Therefore, the total number of iterations is mn * mn * m*n, resulting in a time complexity of O(m^3 * n^3).

Space Complexity:

The space complexity of the above algorithm is O(1) because the algorithm only uses a constant amount of extra space to store the maximum area and the indices of the submatrix with the maximum area. The input matrix is not modified in place and no extra data structures are created during the execution of the algorithm.

Alg7 Design a Θ(mnk) time Dynamic Programming algorithm for solving Problem3

Task7a Give a recursive implementation of Alg7 using Memoization.

Strategy to be implemented:

The objective is to find the largest square submatrix in the given matrix p that contains at most k elements less than h but every other element greater than h. Use a recursion and memoization based top down dynamic programming approach to precompute the count of elements less than h in each submatrix of p and then recursively explores all possible submatrices that could contain at most k elements less than h using a memoized helper function helper. The function computes the area of a submatrix (i.e., the number of elements it contains), which is used to compare submatrices in the main loop to find the one with the largest area. The final output is the indices of the submatrix

Recurrence Relation:

To create the dp array:
dp[i][j] = dp[i-1][j] + dp[i][j-1] - dp[i-1][j-1] + (p[i][j] < h)

Recurrence function call:
helper(i, j, k) =

A. if k==0 then (i, j, i, j)
B. if dp[i+k][j+k] - dp[i+k][j-1] - dp[i-1][j+k] + dp[i-1][j-1] <= k then (i, j, i+k, j+k)
C. else, then helper(i, j, k-1)

Algorithm:

1. Initialize dp as a matrix of size m x n.
2. Populate dp using the above mentioned recurrence relation: This computes the count of elements less than h in the submatrix that spans from (0,0) to (i,j).
3. Define a recursive function helper(i, j, k) that takes the starting indices i and j, and the maximum number of allowed elements less than h, k.
4. Use memoization to store previously computed submatrices and their areas.
5. If k == 0, return the indices of the submatrix (i, j, i, j).
6. Otherwise, check if the submatrix (i, j, i+k, j+k) has at most k elements less than h. If so, return (i, j, i+k, j+k).
7. Otherwise, recursively call helper(i, j, k-1) to explore smaller submatrices.
8. Store the result in the memoization table and return it.
9. Initialize res to None.
10. Loop over all possible submatrix sizes from 1 to k.
11. For each submatrix size k1, loop over all possible starting indices (i, j) such that the submatrix fits within p.
12. Use the helper recursive function to find the submatrix with the largest area that contains at most k elements less than h for the current (i, j, k1) combination.
13. If the result is better than the previous best result, update res.
14. Convert the indices of res to one-indexed format and return them.

Pseudocode:

```
1. Define a function named that takes three arguments, p, h, and k
2. Initialize dp array
3. Fill dp array using using (dp[i-1][j] if i > 0 else 0) +
   (dp[i][j-1] if j > 0 else 0) - (dp[i-1][j-1] if i > 0 and j > 0
   else 0) + (p[i][j] < h)
4. Initialize memo = {}
5. Define function helper(i,j,k1)
6. If (i, j, k1) is in memo,return the corresponding value
7. if k1 equals 0, set res equal to (i, j, i, j)
8. If k1 is not 0, check count of elements < h in submatrix (i, j,
   i+k1, j+k1) is <= k
   ->If yes, set res equal to (i, j, i+k1, j+k1)
   ->If no, call helper with k1-1 and set res = recursive call
   result
9. Store res in memo as (i, j, k1)
10.     Return res
11.     Initialize res = {} outside
12.     For k1 from 1 to k+1
        For i from 1 to m-k1
           For j from 1 to n-k1
              cur_res = Call helper( i, j, k1)
13.     If cur_res != None and func(cur_res) is greater than
   func(res), set res = cur_res
14.     Return res
```

Proof of Correctness:

The algorithm aims to find the largest submatrix in an m x n matrix, satisfying the condition that the number of elements in the submatrix that are less than a given value h is at most k. The algorithm employs dynamic programming and recursion to achieve this objective.

The correctness of the recursion step relies on the observation that if a submatrix with top-left corner (i, j) and bottom-right corner (i+k, j+k) contains at most k elements less than h, any submatrix contained within it also has at most k elements less than h. By recursively searching for the largest submatrix with at most k elements less than h over all submatrices with top-left corner (i, j) and bottom-right corner (i+k, j+k) for all values of i, j, and k and tracking the largest submatrix found so far, the algorithm correctly searches for the largest submatrix satisfying the condition.

Thus, the algorithm's correctness follows from correctly computing the count of elements less than h in each submatrix of the given matrix using dynamic programming and then searching for the largest submatrix with at most k elements less than h using recursion. The algorithm always returns a valid solution satisfying the problem statement.

Time Complexity:

The time complexity of the find_p3_memo algorithm is $\Theta(mnk)$, where m, n are the dimensions of the matrix p and k is the maximum number of elements less than h allowed in a submatrix.

The initialization of the dp array takes $\Theta(mn)$ time, as it involves iterating over all elements of the matrix p. The helper function is called $\Theta(mnk)$ times in the worst case, as it is called for each possible submatrix with at most k elements less than h. The computation of the count of elements less than h in each submatrix using dynamic programming takes constant time per submatrix, so the total time spent on dynamic programming is also $\Theta(mnk)$. The comparison of submatrices to find the one with the largest area takes constant time, so the overall time complexity of the algorithm is $\Theta(mnk)$.

Space Complexity:

The space complexity of the algorithm is dominated by the dp array used to store the count of elements less than h in each submatrix. This array has dimensions m by n, so it uses O(mn) space. The space used by the memo dictionary is proportional to the number of recursive calls made by the helper function, which is at most O(mnk) because there are O(mn) possible submatrices to consider and each recursive call reduces the size of the submatrix by at most one row and one column. Therefore, the space complexity of the algorithm is O(mn + mnk), which simplifies to O(mnk) because k is less than or equal to mn.

Alg7: Design a Θ(mnk) time Dynamic Programming algorithm for solving Problem3

Task7b: Give an iterative BottomUp implementation of Alg7

Strategy to be implemented:

The code finds the largest submatrix in a given matrix p such that the count of elements less than a given h is less than or equal to a given integer k but every other element is greater than h. It uses dynamic programming to optimize the time complexity. The cnt list is used to store the count of elements less than h in each submatrix, and the dp list is used to store the largest submatrix for each combination of starting index, size, and threshold count. The function func calculates the size of the submatrix, and the function find_p3_dp returns the one-based indices of the largest submatrix that meets the given criteria.

Recurrence Relation:

For cnt matrix:
  cnt[i][j] = (cnt[i-1][j] if i > 0 else 0) + (cnt[i][j-1] if j > 0 else 0) - (cnt[i-1][j-1] if i > 0 and j > 0 else 0) + (p[i][j] < h)

For dp matrix:
A. If k1 = 0, then dp[i][j][k1] = (i, j, i, j).
B. If i + k1 < m and j + k1 < n and cnt[i+k1][j+k1] - (cnt[i+k1][j-1] if j > 0 else 0) - (cnt[i-1][j+k1] if i > 0 else 0) + (cnt[i-1][j-1] if i > 0 and j > 0 else 0) <= k,
Then dp[i][j][k1] = (i, j, i+k1, j+k1).

C. Else, dp[i][j][k1] = dp[i][j][k1-1].

Algorithm:

1. Initialize cnt as a two dimensional array.
2. For each element, initialize cnt[i][j] based on the recurrence relation mentioned above.
3. Initialize empty 3d dp array as empty.
4. Nested loops:
   Iterate k1 through k+1
   Iterate i through m
   Iterate j through n
   Create dp array based on the recurrence relation mentioned above.
5. Initialize res = none
6. Nested loops:
   Iterate k1 through k + 1
   Iterate i through m - k1
   Iterate j through n - k1
   Current res = dp[i][j][k1]
   If current res is not None and function(Current res) > function(res)
   Res = Current res
7. Return res

Pseudocode:

```
1. Initialize 2D array cnt of size m x n,
2. Compute cnt as follows. For each row i and column j, initialize
   cnt[i][j] to zero.
3. For each row i and column j, compute cnt[i][j] as the sum of the
   following values:
   i. cnt[i-1][j] if i > 0
   ii. cnt[i][j-1] if j > 0
   iii. cnt[i-1][j-1] if both i > 0 and j > 0
   iv. 1 if p[i][j] < h, else 0.
4. Initialize 3D array dp of size m x n x (k+1),
5. Compute using dp as follows:
   For each value of k1 from 0 to k, and each row i and column j,
   initialize dp[i][j][k1] as follows:
   i. If k1 == 0, set dp[i][j][k1] to (i,j,i,j).
   ii. Otherwise, if i+k1 < m and j+k1 < n and the number of
   elements in the submatrix from (i,j) to (i+k1,j+k1) that are less
   than h is at most k, set dp[i][j][k1] to (i,j,i+k1,j+k1).
   iii. Otherwise, set dp[i][j][k1] to dp[i][j][k1-1].
6. Initialize res to None.
```

```
7. For each value of k1 from 0 to k, and each row i and column j,
   compute cur_res as dp[i][j][k1].
8. If cur_res is not None and either res is None or func(cur_res) >
   func(res), set res to cur_res.
9. Return res
```

Proof of Correctness:

The correctness of the algorithm is based on two observations:

- Given a submatrix with top-left corner (i,j) and bottom-right corner (i+k,j+k), if the number of elements less than h in this submatrix is at most k, then any submatrix contained within this submatrix also has at most k elements less than h. Therefore, the largest submatrix with at most k elements less than h can be found by iterating over all submatrices with top-left corner (i,j) and bottom-right corner (i+k,j+k) for all values of i, j, and k.

- The count of elements less than h in a submatrix can be calculated efficiently using dynamic programming by precomputing the count of elements less than h for each submatrix with the top-left corner (0,0) to (i,j).

Based on these observations, the algorithm first precomputes the count of elements less than h in each submatrix using dynamic programming. It then creates a 3D DP array dp where dp[i][j][k1] represents the submatrix with maximum size k1 that has at most k elements less than h. The algorithm iterates over all submatrices with top-left corner (i,j) and bottom-right corner (i+k,j+k) and checks if the number of elements less than h in the submatrix is at most k. If it is, it updates dp[i][j][k1] to the current submatrix size. If not, it keeps the previous value of dp[i][j][k1]. This process populates the dp array with the largest submatrix for each combination of starting index, size, and threshold count. Finally, the algorithm finds the maximum size submatrix by iterating over all possible combinations of submatrices and comparing their sizes. It returns the one-based indices of the largest submatrix that meets the given criteria.

Therefore, the algorithm always returns a valid solution that satisfies the problem statement.

Time Complexity:

The time complexity of the above algorithm is Θ(mnk), where m is the number of rows in the input matrix, n is the number of columns, and k is the threshold count. This is because the algorithm requires iterating over all possible submatrices of the input matrix, which takes Θ(m^2 * n^2) time. However, the use of dynamic programming to compute the count of elements less than h in each submatrix reduces the time complexity to Θ(mn). This allows the algorithm to complete in Θ(mnk) time.

Space Complexity:

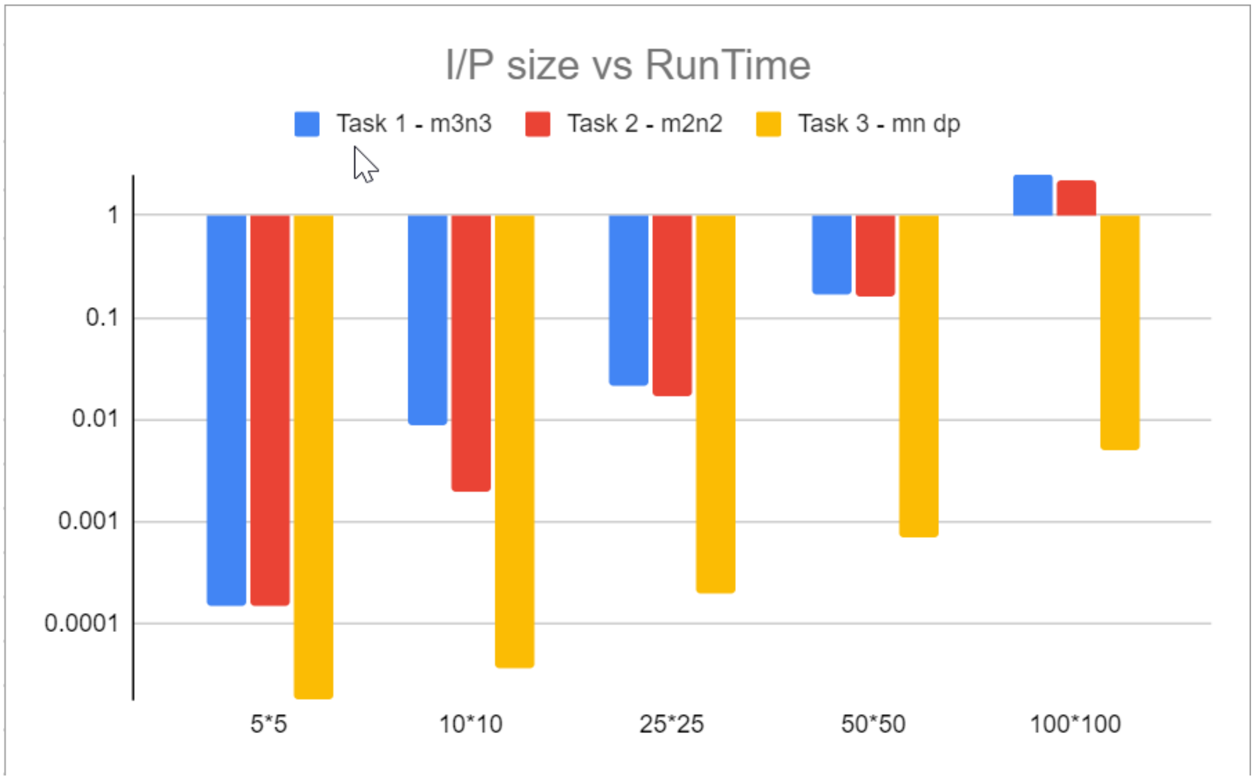
The space complexity of the above algorithm is O(mnk), as it uses a 3D list to store the largest submatrix for each combination of starting index, size, and threshold count. The size of this list is (m x n x k), which is proportional to the input size (m x n) and the maximum count threshold (k). The cnt list also uses O(mn) space, and the other variables used in the algorithm (m, n, h, k) are constant space. Therefore, the overall space complexity is O(mnk).

<u>Experimental Comparative Study</u>

As a part of this, I generated random matrices p, and integers h and k. I ran each task for these sets of inputs. The histograms for this case study is shown below. All histograms show running time on y axis vs the input size on the x axis

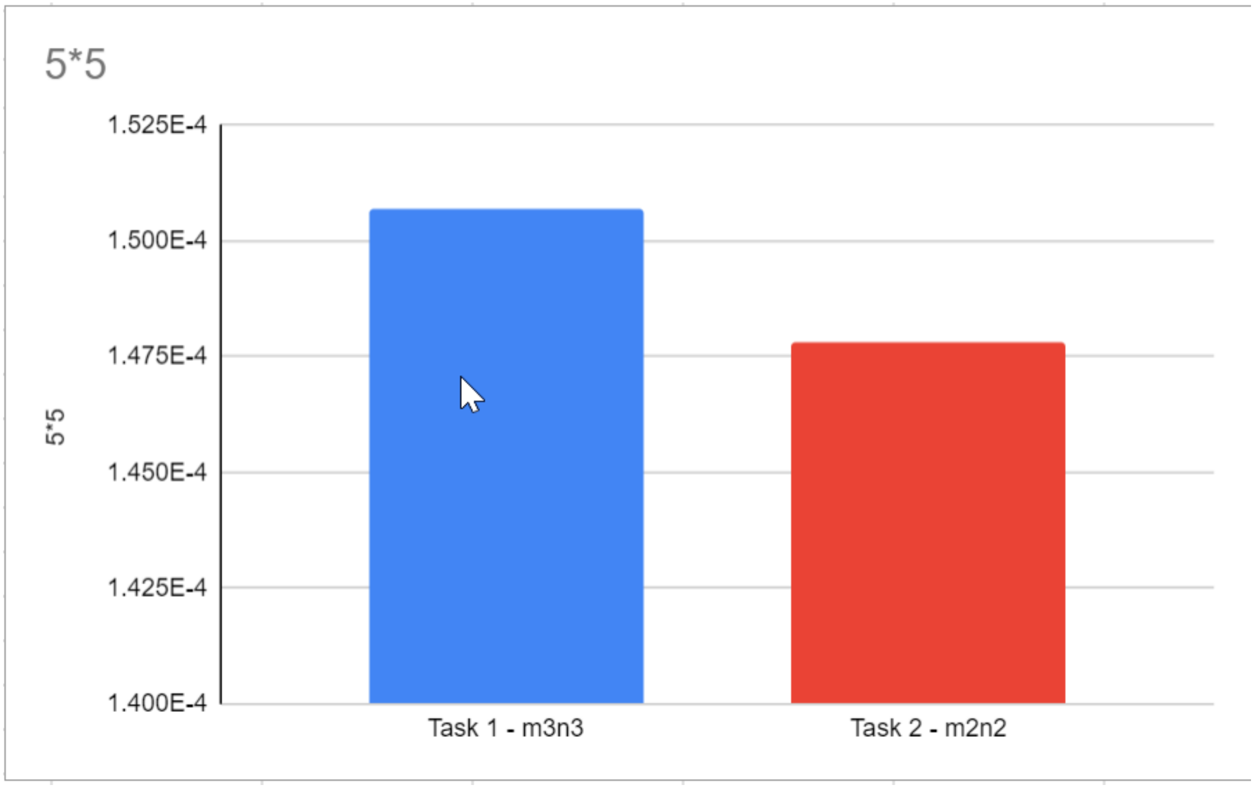
Plot 1: This shows the comparison for each task in problem 1

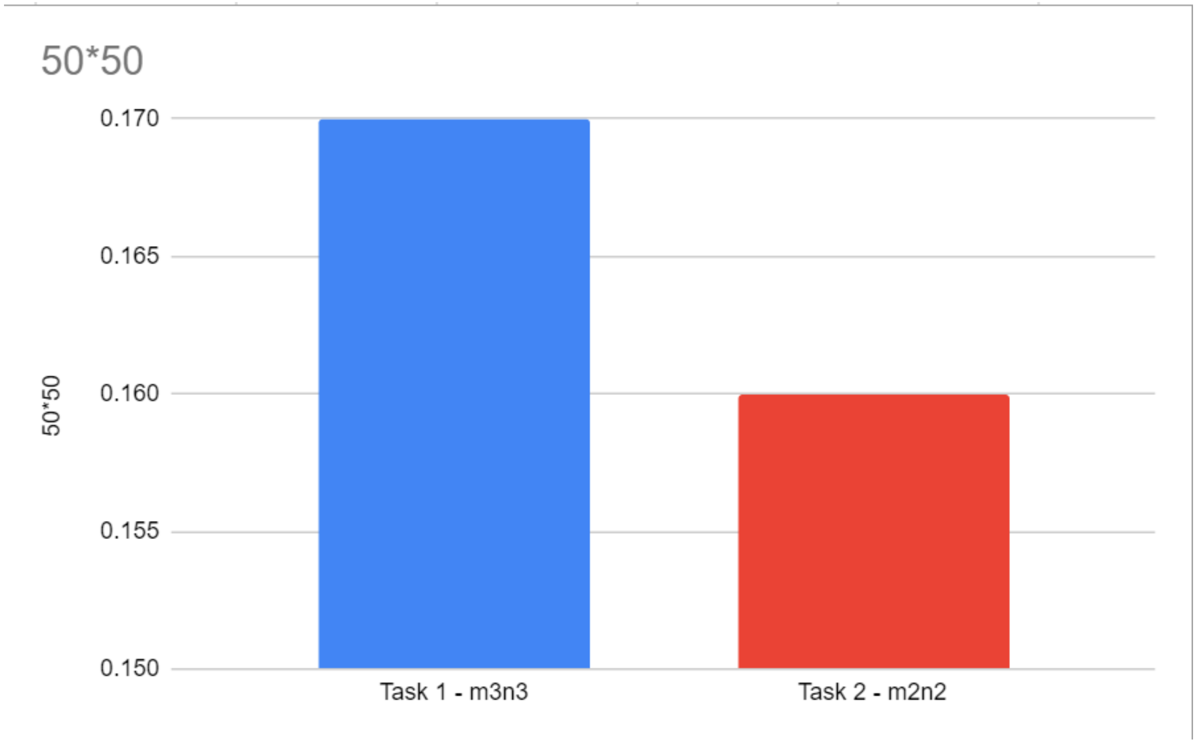
Task 1 vs Task 2 vs Task 3



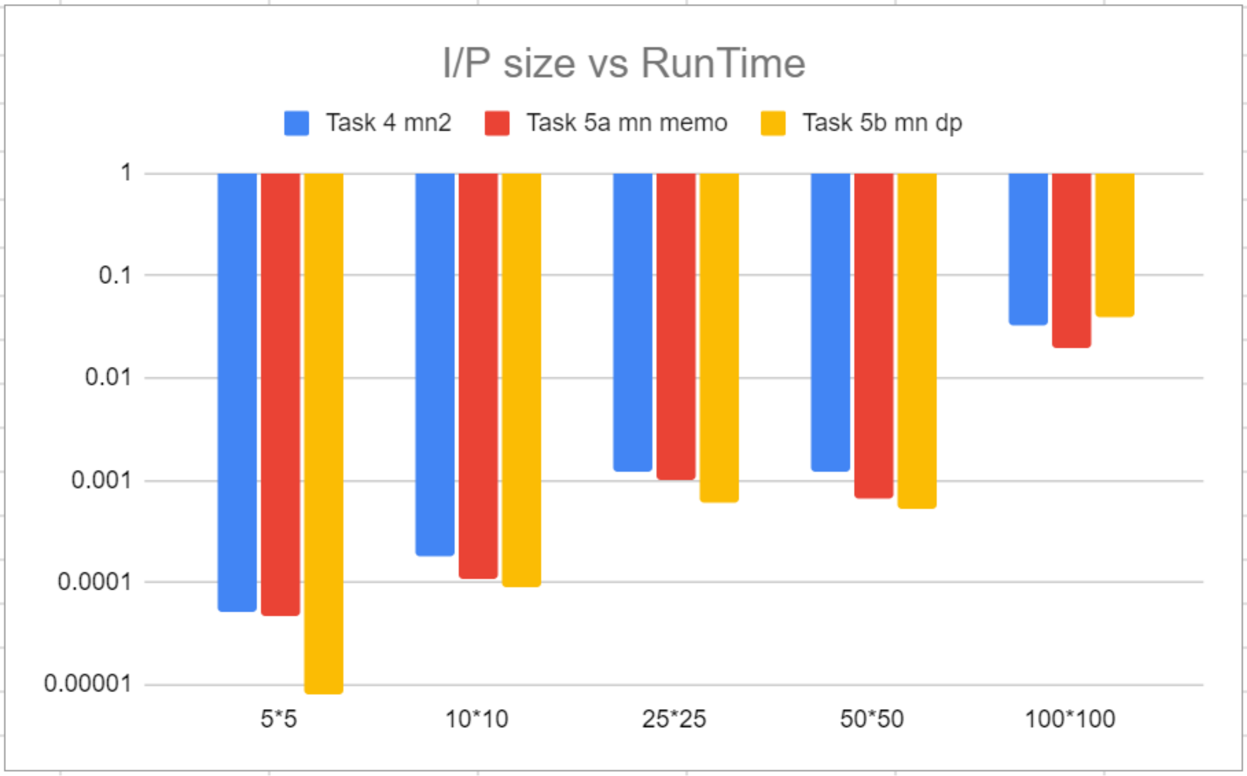
Task 3 with *mn* is the fastest, followed by Task 2 with *m^2n^2*. Task 1 with *m^3n^3* is the slowest

Additional Plots: This shows a closer view of the difference

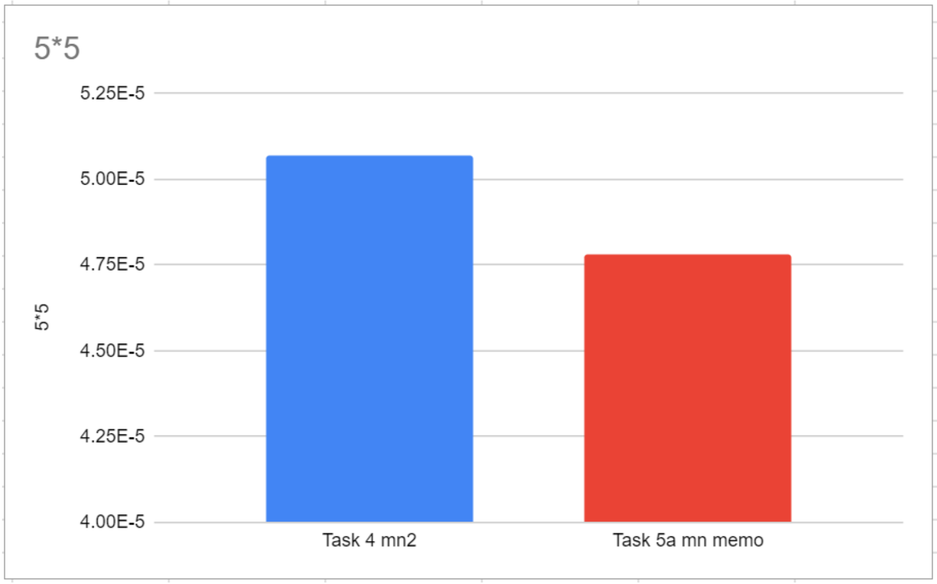




Plot 2: This shows the comparison for each task in problem 2 Task 4 vs Task 5A vs Task 5B

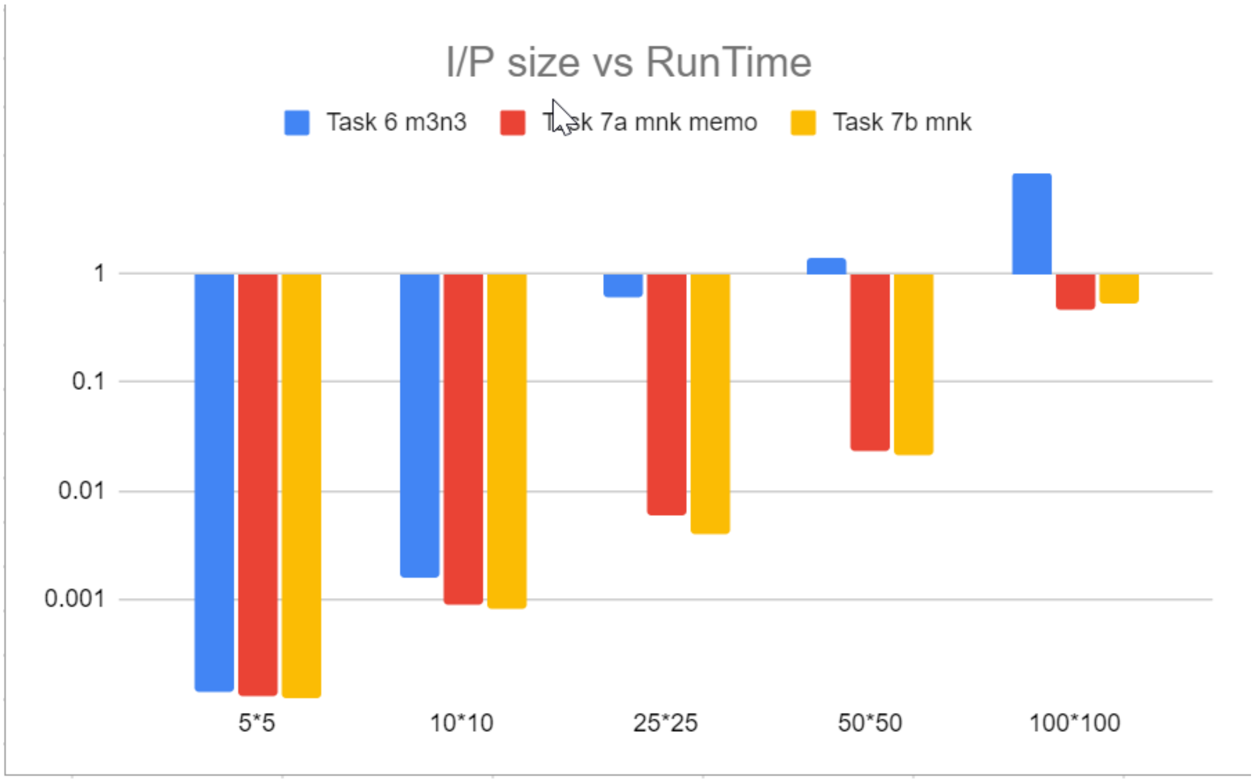


Task 5a and Task 5b with *mn* perform similarly . Task 4 with *m^2n^2* is the slowest Additional Plots: This shows a closer view of the difference



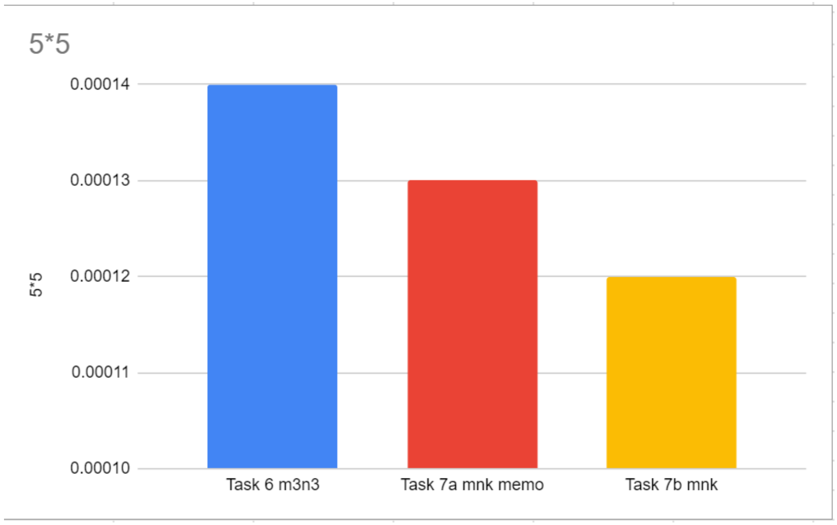
Plot 3: This shows the comparison for each task in problem 3

Task 6 vs Task 7A vs Task 7B



Task 7a and Task 7b with *mnk* perform similarly . Task 6 with *m*^3*n*^3 is the slowest

Additional Plots: This shows a closer view of the difference



Conclusion

In each of the programming tasks, I experienced a varying level of difficulty and complexity. Trying to understand both top down and bottom up approaches to dynamic programming for each problem while maintaining the tightly bound time complexities was a challenge in itself. During the implementation of the programming project, I made a few observations. Firstly, it is observed that the brute force approach consistently produces slower outputs. The most optimal solution across all cases is the m*n. This is further evidenced at higher values of m and n and indicated by the histograms . In addition the dp algorithm is faster than any other algorithm. There was not significant difference between bottom up and top down implementations. I also faced a few challenges during the implementation. Coming up with an algorithm that ran in the required time complexity for Task 7A and 7B was extremely challenging. Finding a solution was a lot easier when there was no constraint on time complexity. Another challenge faced was to find appropriate test cases for all the tasks.