COT 5405 ANALYSIS OF ALGORITHMS
Programming Project II Dynamic Programming
Spring 2023

Team Members
Abdul Samadh - 2213-7268
Sheel Taskar - 2753-4463

Team Members

The programming project is an original work done by Abdul Samadh and Sheel Taskar. Abdul Samadh designed and programmed the brute force algorithm ALG1 for problem 1, optimized O(m*n) algorithm (both top-down ALG 5A and bottom-up ALG 5B) approach for problem 2, and brute force algorithm ALG 6 for problem 3. Sheel Taskar designed and programmed a slightly optimized version of algorithm 1, i.e. O(m^2*n^2) algorithm ALG 2 for problem 2. Sheel also designed dynamic programming solutions ALG 3 and ALG 4 for problems 1 and 2. Both Abdul and Sheel were involved in brainstorming and design of ALG 7 for problem 3. Abdul wrote the program for the top-down ALG 7A approach for problem 3 whereas Sheel wrote the program for the bottom-up ALG 7B approach for problem 3. Abdul and Sheel wrote the design, pseudocode, and analysis of their respective implementations in the report.
Abdul did the comparative studies of all the programming tasks in the report and plotted and compared the runtime graph of tasks. Sheel did the testing of all the algorithms and made the Makefile and final consolidated python file, and tested the setup on the thunder server.

Design and Analysis of Algorithms

Task 1: Design a Θ(m^3*n^3) time Brute Force algorithm for solving Problem1

Design:

1. The brute force approach will try to find out all possible submatrices in the input matrix.
2. To find all possible submatrices we need to fix the four corners of the submatrix.
3. To fix the four corners, we need to fix horizontal corners (either top edge or bottom edge) and vertical corners (either left edge or right edge.)
4. As we have m rows, to fix two vertical corners we need m^2 iteration.
5. As we have n columns, to fix horizontal corners we need n^2 iteration.
6. So we can find all possible submatrices in m^2*n^2 iterations.
7. Once we find the submatrix, we check if each and every element is greater than the threshold or not. This can be done in m*n iterations.
8. If any of the elements is less than the threshold, we discard that particular solution.
9. We keep track of all found solutions and return the solution with the maximum size.

Pseudocode:

1. Initialize i1, j1, i2, j2 to None. These are the indices that will contain the top-left and bottom-right indices of the largest square where all elements in the square are greater than the threshold
2. Initialize maxSquareSize to 0
3. Get the number of rows and columns in the matrix and store them in rows and cols respectively

4. Iterate over each possible sub-square in the matrix using four nested loops, with indices i, j, k, and l where i,j will be the top-left element of the sub-square and j,k will be the bottom-right element of the sub-square
5. Calculate the size of the square by taking squareSize = k-i+1 if k-i == l-j and k-i >= 0, otherwise set squareSize to 0
6. If squareSize is non-zero, check if all elements in the square are greater than or equal to h. If so, update maxSquareSize, i1, j1, i2, j2 if squareSize is greater than or equal to maxSquareSize
7. Return i1, j1, i2, j2

Time and Space Complexity Analysis:

The two outer loops use i,j,k, and l to go through all possible subsquares where i,j represents the top-left element of the subsquare and k,l represents the bottom-right element of the subsquare. Since i and k loop through all rows(m) and j and l loop through all the columns we will be doing Θ(m^2*n^2) operations to generate all possible subsquares. For each subsquare, we traverse all elements to see if each element is greater than the given threshold. Since this operation will require m*n iterations to check all elements of the subsquare the total time complexity will be Θ(m^3*n^3). As our algorithm keeps track of the largest square found in variables, we don't need extra space, hence the space complexity of the algorithm will be O(1).

Proof of Correctness:

Since we are generating all possible subsquares of the given matrix and checking if each element of the subsquare is greater than the threshold, we can be sure that the algorithm is correct.

Task 2: Design a Θ(m^2*n^2) time algorithm for solving Problem1

Design:

1. Unlike the previous solution, instead of finding all possible submatrices, we find all possible squares.
2. To represent a square, we use the top left corner of the square and its size. If we have one corner and size, we can find other four corners of squares as well.
3. So to select the top left corner, we need m*n iterations, as all the elements in the matrix can form their own square.
4. Once a corner is fixed we need to fix its diagonal. To fix its diagonal, we would need a minimum of m or n iterations, as forming a square larger than that is not possible.
5. Hence, for each element, we move the diagonal, one at a time, and with each increment, we check the row and column including whether the diagonal element is greater than the threshold or not. This can be done in the time that is linear in m and n. If greater, we continue our increment, else we discard it and return to the previous solution.

6. We keep track of solutions of every element in the submatrix and return the one with the maximum size.

Pseudocode:

```
1. Iterate over every element in the input matrix A and do steps 2 to 6.
2. If A[i][j] > threshold, SET submatrix Size => 1 and go to step 3,  else skip.
3. Increment i and j to get the diagonal element till you reach the corner element of A.
4. Iterate ith row and jth column, check every element with threshold.
5. If any of the elements < threshold, goto step 6, else SET submatixSize => submatrix Size+1.
6. Get the submatixSize and its index i,j.
7. Return the index with maximum submatrix Size.
```

Time and Space Complexity Analysis:

The solution requires us to iterate over every element in the matrix, which in the worst case requires O(m*n) time. For every element, we need to adjust the value of the diagonal that is the bottom right element. The maximum value the diagonal element can achieve is a minimum of m or n, as we cannot find a bigger square than that in a matrix of size m X n. Hence adjusting the diagonal element will take O(min(m,n)) time. To check the row and column belonging to the diagonal we will need a time liner in m and n, that is O(m+n). Therefore total time complexity will be O(m*n) * O(min(m,n)) * O(m+n) = O(m^2n^2).
As our algorithm keeps track of the largest square found in variables, we don't need extra space, hence the space complexity of the algorithm will be O(1).

Proof of Correctness:

Similar to the previous solution, we are generating all possible Squares of the given matrix and checking if each element of the Square is greater than the threshold, we can be sure that the algorithm is correct.

Algorithm 3: Design a Θ(m*n) time Dynamic Programming algorithm for solving Problem1

Design:

1. Initialize a variable maxSquareSize to 0
2. Initialize variables i1, j1, i2, j2 to None. These are the indices that will contain the top-left and bottom-right indices of the largest square where all elements in the square are greater than the threshold

3. Get the number of rows and columns of the matrix
4. Initialize a 2D array dp with dimensions (rows+1) x (cols+1) with all elements initialized to 0
5. Loop over each element in the matrix with indices (i,j) starting from (1,1):
    - i. If the element is greater than or equal to h, set dp[i][j] to the minimum of dp[i][j-1], dp[i-1][j], and dp[i-1][j-1] plus 1
    - ii. If dp[i][j] is greater than or equal to maxSquareSize, set maxSquareSize to dp[i][j], and set i1, j1, i2, and j2 to the bounding indices of the max square
6. Return i1, j1, i2, j2 as the indices of the largest square of elements greater than or equal to h in the matrix.

Time and Space Complexity Analysis:

For each element in the dp matrix we take the minimum of the values of the cell on the top, top-left and bottom left. This in a Θ(1) operation. We do the same operation for all the elements in the dp matrix. Hence the total time complexity will just be the cost of traversing the whole dp matrix which is Θ(mn). We use extra space to store a dp matrix which has m+1 rows and n+1 columns. Hence space complexity will be Θ(mn).

Proof of Correctness:

OPT(i,j): The size of the largest valid subsquare(all elements of subsquare are greater than threshold) whose bottom right index is i,j.

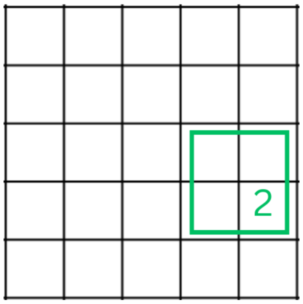
Our goal is to find the maximum of OPT(i,j) for all i,j.

Case 1: If i,j is in the first row or first column, then the largest square with i,j as the bottom right index will be 1 if the element at i,j is greater than threshold, otherwise the size will be 0.

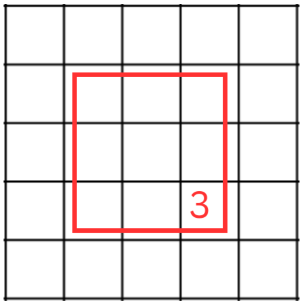
Case 2: For every element the largest square with i,j as the bottom right element will depend upon the size of largest valid squares on its top, top-left and left positions. If the value at i,j is less than threshold then it cannot contribute to the subsquare along with its neighbors. On the other hand if the value at i,j is greater than or equal to the threshold then the size of the largest valid square will be min(OPT(i-1,j-1), OPT(i,j-1), OPT(i-1,j)) +1.

Below is the diagrammatic illustration of the same.

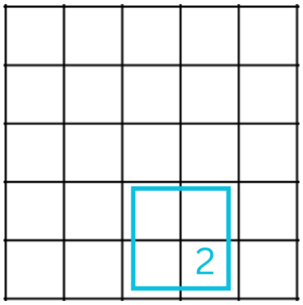
The square that is formed by element(i-1,j) just above our element(i,j) has size 2.



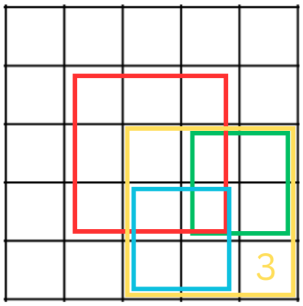
The square that is formed by element(i-1,j-1) just above diagonally our element(i,j) has size 3.



The square that is formed by element(i,j-1) just left our element(i,j) has size 2.



If we try to intersect all the three squares, our resulting square is as below.



Bellman Equation

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

Task 4: Design a Θ(m*n^2) time Dynamic Programming algorithm for solving Problem2

Design:

1. Create a binary matrix where each element is 1 if the corresponding element in the input matrix is greater than or equal to h, and 0 otherwise.
2. Create a prefix sum matrix where each element (i,j) is the sum of all elements in the submatrix from (0,0) to (i,j) in the binary matrix.
3. Initialize variables to keep track of the maximum square size and its coordinates.
4. For each starting position (i,j) in the matrix, iterate over all possible square sizes k that can fit within the remaining submatrix.
5. Compute the sum of all elements in the submatrix from (i,j) to (i+k-1,j+k-1) using the prefix sum matrix.
6. Compute the sum of the four corner elements of the square.
7. Compute the sum of all elements inside the square (excluding the corners).
8. If the sum of the inside elements is greater than or equal to k*k-4, then the square is valid (because it contains at least 4 elements that are greater than or equal to h). Update the maximum square size and its coordinates if necessary.
9. Return the coordinates of the largest square.

Time and Space Complexity Analysis:

We first convert the input matrix into a binary matrix where the element in the binary matrix is 1 if the corresponding element in the input array is greater than threshold, else it is 0. The creation of this binary matrix will take be Θ(mn).

Now we create a prefix sum matrix which will store the sum of all elements in the sub-matrix from (0,0) to (i,j) in (i,j). Using this prefix sum we can find the total sum of any submatrix whose top-left element is i,j and whose size is k in O(1). The creation of prefix matrix will be Θ(mn) since we will iterate the binary matrix two times to calculate cumulative sum for rows and columns.

We traverse all the elements of the binary matrix. For each element at location i,j we will evaluate all possible subsquares whose top-left element is i,j and whose size is k. k will range from 1 to min(m-i,n-j). For each combination of i,j,k using the prefix matrix we can find the sum of all elements of the subsquare of size k with i,j being the top left element of the subsquare. Here the sum of the square is equivalent to the number of elements with value greater than or equal to threshold. We can find the sum of the square and sum of the corner elements in O(1) for a subsquare represented by i,j,k. Using this information we can derive if there is any element in the inner elements (non corner) which is 0(value less than threshold).

Total time complexity will be mn(create binary matrix) + mn(create prefix sum matrix) + mn*min(m,n), hence TC will be Θ(mn*min(m,n)) which is Θ(mn^2) or Θ(m^2n). Since we use two new matrices, which are both of size mXn, the space complexity will be Θ(mn).

Proof of Correctness:

To generate the representation of a square (i,j,k) where i,j is the coordinate of the top-left corner of the subsquare of size k we need to do mn*min(m,n) operations. To iterate through all the possible i,j combinations we will need mn operations and for each i,j value, k (subsquare size) will iterate from 1 to min(m,n) to account for all possible subsquare sizes from i,j.

To make things simpler we convert the input matrix into a binary matrix. The binary matrix will contain 1 if the corresponding element in the input matrix is greater than or equal to threshold and 0 otherwise. With this formulation, the sum of a submatrix will give us the number of elements greater than threshold in the submatrix.

We need to prove that, given a representation i,j,k we can,in constant time, and correctly figure out if it is a valid square ie. all elements but the corner elements have a value greater than or equal to the given threshold.

To do this we need a prefix sum matrix where each cell whose coordinates are i,j contains the sum of elements in the submatrix whose top-left cell is 0,0 and bottom-right cell is i,j.

Let OPT(i,j) = Sum of all elements in the submatrix whose top-left element is 0,0 and bottom-right element is i,j.

We need to find all OPT(i,j) to create the prefix sum matrix. We can use the following bellman equation to find any OPT(i,j)

$$OPT(i,j) = \begin{cases} 0 & \text{if } i==0 \text{ or } j==0 \\ OPT(i-1,j) + OPT(i,j-1) - OPT(i-1,j-1) \\ \quad + binaryMatrix[i-1][j-1] & \text{For other cases} \end{cases}$$

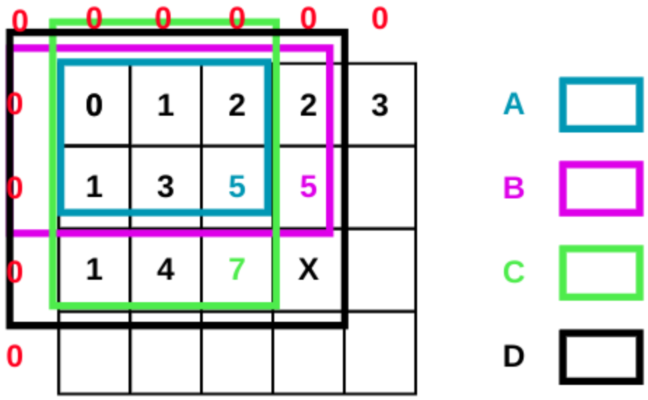
For example, the following binary matrix will be converted to the following prefix sum array:

0	1	1	0	1
1	1	1	0	0
0	1	1	1	0
1	0	1	1	1

→

0	0	0	0	0	0
0	0	1	2	2	3
0	1	3	5	5	6
0	1	4	7	8	9
0	2	5	9	11	13

We prove that this formulation is correct by using the following example,



0	1	1	0	1
1	1	1	0	0
0	1	1	1	0
1	0	1	1	1

Let’s assume we need to find X which is at coordinate OPT(i,j). X represents the sum of all elements in the submatrix from the origin(0,0) to (i,j) which is represented by rectangle D.
Similarly,

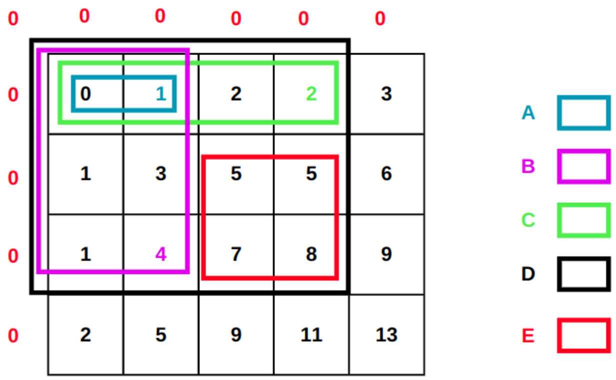
OPT(i,j-1) represents the sum of triangle C and its value is 7
OPT(i-1,j) represents the sum of triangle B and its value is 5
OPT(i-1,j-1) represents the sum of triangle A and its value is 5

Sum of Rectangle D = Sum of Rectangle B + Sum of Rectangle C - Sum of Rectangle A + binaryMatrix[i][j]
We subtract the sum of rectangle A because it is counted twice when adding the sum of rectangles B and C.

From the above equation we can prove OPT(i,j) = OPT(i,j-1) + OPT(i,j-1) + OPT(i,j-1) + binaryMatrix[i][j].

Now we can use this prefix sum matrix to find the sum of any subsquare with top-left coordinates i,j and size k in O(1) using the following equation.
Sum of subsquare = prefixSum[i+k][j+k] - prefixSum[i][j+k] - prefixSum[i+k][j] + prefixSum[i][j]

To prove this we can use the following diagram.



0	1	1	0	1
1	1	1	0	0
0	1	1	1	0
1	0	1	1	1

To find the sum of square E with top-left coordinate i,j and size k, we can use the following using equation,

Sum of E = Sum of D - Sum of C - Sum of B + Sum of A

The rectangles A,B,C and D start from origin so we can use prefix sum matrix to calculate their sums,

Sum of E = prefixSum[i+k][j+k] - prefixSum[i][j+k] - prefixSum[i+k][j] + prefixSum[i][j]

Hence,
SubSquareSum(i,j,k)=prefixSum[i+k][j+k]-prefixSum[i][j+k]-prefixSum[i+k][j] +prefixSum[i][j]

We can find the corner sum of a sub-square represented by i,j,k using
CornerSum(i,j,k)=binaryMatrix[i][j]+binaryMatrix[i+k-1][j+k-1]+binaryMatrix[i][j+k-1] + binaryMatrix[i+k-1][j]

We then have,
InnerSum(i,j,k) = SubSquareSum(i,j,k)-CornerSum(i,j,k)

InnserSum(i,j,k) must be greater than or equal to k^2-4 for it to be a valid square.

We evaluate inner sum for all values of i,j,k and return the coordinates of the largest valid square in Θ(mn*min(m,n)).

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

Design:

Dynamic programming implementation for problem 2 is very similar to dynamic programming implementation for problem detailed in Task 3. Only this time we are allowed to take squares whose corners may be less than the threshold. In Task 3 for each i,j, we considered the value at i,j to be the size of the largest square with all values greater than threshold with i,j being the bottom right square. To find the value at i, j, we used the minimum of the precomputed valid squares with all elements greater than threshold, at the top, top-left and left of the current cell and added 1 to it. In Task 5 we will use the same principle with a slight modification and make use of 4 DP matrices.

- The first DP matrix will be called top-right-dp. For each i,j in this matrix the value at i,j will be the largest side of the square with i,j being the bottom right element of the square, ignoring the value of its top-right most element.
- The second DP matrix will be called top-left-dp. For each i,j in this matrix the value at i,j will be the largest side of the square with i,j being the bottom right element of the square, ignoring the value of its top-left most element.
- The third DP matrix will be called bottom-left-dp. For each i,j in this matrix the value at i,j will be the largest side of the square with i,j being the bottom right element of the square, ignoring the value of its bottom-left most element.
- The final DP matrix will use the following equation to fill values, dp[i][j] = min(top_left[i-1][j-1], top[i-1][j], left[i][j-1])+1

Pseudocode for Top-Down Recursion + Memoization:

```
1. Define a function calculateTopLeft(matrix, top_left, row, col, h) to calculate the value of
   top_left[row][col] using top-left, left, and top values.
    a. If the original element matrix[row][col] is less than h, set top_left[row][col] = 1.
    b. If either the top or left element of the original matrix is less than h, set top_left[row][col]
       = 1.
    c. Otherwise, recursively call calculateTopLeft(matrix, top_left, row, col-1, h),
       calculateTopLeft(matrix, top_left, row-1, col, h), and calculateTopLeft(matrix, top_left,
       row-1, col-1, h) if top_left values are -1.
    d. Set top_left[row][col] to the minimum value of top_left[row][col-1], top_left[row-1][col],
       and top_left[row-1][col-1] plus 1.
```

2. Define a function calculateTopRight(matrix, top_right, row, col, h) to calculate the value of top_right[row][col] using top-right, right, and top values.
    a. If the original element matrix[row][col] is less than h, set top_right[row][col] = 1.
    b. If either the top-right or right element of the original matrix is less than h, set top_right[row][col] = 1.
    c. Otherwise, recursively call calculateTopRight(matrix, top_right, row, col-1, h), calculateTopRight(matrix, top_right, row-1, col, h), and calculateTopRight(matrix, top_right, row-1, col-1, h) if top_right values are -1.
    d. Set top_right[row][col] to the minimum value of top_right[row][col-1], top_right[row-1][col], and top_right[row-1][col-1] plus 1.
3. Define a function calculateBottomLeft(matrix, bottom_left, row, col, h) to calculate the value of bottom_left[row][col] using bottom-left, left, and bottom values.
    a. If the original element matrix[row][col] is less than h, set bottom_left[row][col] = 1.
    b. If either the bottom-left or left element of the original matrix is less than h, set bottom_left[row][col] = 1.
    c. Otherwise, recursively call calculateBottomLeft(matrix, bottom_left, row, col-1, h), calculateBottomLeft(matrix, bottom_left, row-1, col, h), and calculateBottomLeft(matrix, bottom_left, row-1, col-1, h) if bottom_left values are -1.
    d. Set bottom_left[row][col] to the minimum value of bottom_left[row][col-1], bottom_left[row-1][col], and bottom_left[row-1][col-1] plus 1.
4. Initialize variables i1, j1, i2, j2, maxSquareSize to 0.
5. Now we iterate through the final DP array to populate it using the following conditions:
    a. For each i,j in final DP array, its value will be min(top_left[i-1][j-1],left[i][j-1],top[i-1][j])+1

Pseudocode for Bottom-Up Dynamic Programming:

1. Initialize rows and cols to the number of rows and columns in the given matrix.
2. Pad the matrix with a row of zeros at the top and a column of zeros on the left.
3. Initialize four 2D arrays dp, top, left, and top_left with zeros, where each array has rows + 1 rows and cols + 1 columns.
4. Use the top array to calculate the size of the largest square submatrix in the given matrix that ends at each position in the top row or leftmost column, ignoring the top-right corner of each submatrix. Specifically, for each i from 1 to rows and each j from 1 to cols, set top[i][j] to:
    a. 1 if i == 1 or j == 1.
    b. 1 if the element in original matrix at position (i, j) is less than h.
    c. min(top[i][j-1], top[i-1][j], top[i-1][j-1]) + 1 otherwise.
5. Use the left array to calculate the size of the largest square submatrix in the given matrix that ends at each position in the leftmost column or top row, ignoring the bottom-left corner of each submatrix. Specifically, for each i from 1 to rows and each j from 1 to cols, set left[i][j] to:
    a. 1 if i == 1 or j == 1.
    b. 1 if the element in the original matrix at position (i, j) is less than h.
    c. min(left[i][j-1], left[i-1][j], left[i-1][j-1]) + 1 otherwise.

6. Use the top_left array to calculate the size of the largest square submatrix in the given matrix that ends at each position in the top row or leftmost column, ignoring the top-left corner of each submatrix. Specifically, for each i from 1 to rows and each j from 1 to cols, set top_left[i][j] to:
    a. 1 if i == 1 or j == 1.
    b. 1 if the element in the original matrix at position (i, j) is less than h.
    c. min(top_left[i][j-1], top_left[i-1][j], top_left[i-1][j-1]) + 1 otherwise.
7. Initialize maxSquareSize to 0 and i1, j1, i2, j2 to None.
8. Use the dp array to calculate the size of the largest square submatrix in the given matrix that ends at each position, considering all corners of the submatrix. Specifically, for each i from 1 to rows and each j from 1 to cols, set min(top_left[i-1][j-1],left[i][j-1],top[i-1][j])+1

Time and Space Complexity Analysis:

For task 5 we make use of 4 new matrices of size (m+1, n+1). So the space complexity will be Θ(mn).

Time Complexity Analysis for Top-Down Memoization:
We iterate through all i,j in the DP matrix whose size is (m,n). For each i,j in DP matrix we make 1 each call to fetch top-left[i-1][j-1], top[i-1][j] and left[i][j-1].

In a top-down recursive approach these 3 auxiliary matrices are not precomputed. All elements in these matrices are initialized to -1 indicating that the value for that cell is yet to be computed. In all the matrices the value of a cell is derived from the values of cells on top, top-left and bottom-left. In the worst case if we require the value of top_left[m][n] (bottom right most cells of top-left auxiliary matrix) there will be m*n recursive calls since all the values from 1,1 to m,n will need to be computed. But every time a value is computed it is stored in the cell, thereby ensuring when the same cell value is sought again it will be returned in O(1) time.

Hence while iterating through the final DP, for each i,j we will make recursive calls to fetch values from the three auxiliary matrices. At most for each of the auxiliary matrices there will me m*n calls, after that all values would have been precomputed and will be returned in O(1). We will need to iterate i,j in the final DP from 1 to m,n. Hence total time complexity will be Θ(mn(iterate through final DP)+3*mn(at most m*n recursive calls for the 3 auxiliary matrices)), therefore Θ(mn).

Time Complexity Analysis for Bottom Up Dynamic Programming:
When iterating through the final DP, at each i,j we need the value at i-1,j-1 from top-left DP, i-1,j from top DP and i,j-1 from left DP. So before iterating through the final DP we need to precompute values for the 3 auxiliary matrices. Once we have the precomputed auxiliary matrices we can find the final DP matrix in Θ(mn).

For each of the auxiliary matrices M(top-left, left and top), to calculate value at M(i,j) we need values at M(i-1,j-1), M(i,j-1) and M(i-1,j). We can do this in one single iteration for all i,j. Hence the cost to precompute each auxiliary matrix is Θ(mn).

Hence, total time complexity will be Θ(mn(iterate through final DP)+3*mn(For auxiliary matrix precomputation), therefore Θ(mn).

Proof of Correctness:

We have already proved in Task 3 that the size of the largest square with its bottom-right most cell being i,j will be the minimum of sizes at valid squares with the bottom left corners at (i-1,j-1), (i-1,j) and (i,j-1) + 1. We will use this result in the following bellman equations to generate the auxiliary matrix and then the final DP matrix.

Top-Left DP Matrix:

$$OPT(i,j) = \begin{cases} 1 \text{ if } i==0 \text{ or } j==0 \\ 1 \text{ if } Matrix(i-1,j) \text{ or } Matrix(i,j-1) \\ \quad \text{or } Matrix(i,j) < h \\ MIN(OPT(i-1,j-1), OPT(i-1,j), OPT(i,j-1)) + 1 \\ \quad \text{for all other cases} \end{cases}$$

Here OPT(i,j) is the side of the maximum square whose bottom right-most element is i,j such that only the top-left most cell of the square may be less than threshold.
For generating top-left DP matrix, for each i,j where i,j is the bottom right most element of square of side with the value at i,j:

- All the elements in the first row or first column are individual squares of size 1 and hence are valid squares since a single square can be considered a top-left corner and hence be ignored.
- For other elements:
    - If top or left cells have values less than threshold in the original matrix then the maximum subsquare size with the current element at i,j can be only 1 since we can only allow the top-left element to be less than threshold.
    - If the value of the current cell is less than threshold in the original matrix then the maximum subsquare size with the current element at i,j can be only 1 since we can only allow the top-left element to be less than threshold and the current element i,j is the bottom-right most element.
    - For all other cases, we can use this equation,
      top-left(i,j) = min(top-left(i-1,j-1),top-left(i-1,j-1),top-left(i-1,j-1)) + 1

Top DP Matrix:

$$\mathrm{OPT}(i,j) = \begin{cases} 1 \text{ if } i==0 \text{ or } j==0 \\ 1 \text{ if Matrix}(i-1,j-1) \text{ or Matrix}(i,j-1) \text{ or Matrix}(i,j) < h \\ \mathrm{MIN}(\mathrm{OPT}(i-1,j-1), \mathrm{OPT}(i-1,j), \mathrm{OPT}(i,j-1)) + 1 \text{ for all other cases} \end{cases}$$

Here OPT(i,j) is the side of the maximum square whose bottom right-most element is i,j such that only the top-right most cell of the square may be less than threshold.

For generating top DP matrix, for each i,j where i,j is the bottom right most element of square of side with the value at i,j:

- All the elements in the first row or first column are individual squares of size 1 and hence are valid squares since a single square can be considered a top-right corner and hence be ignored.
- For other elements:
    - If top-left or left cells have values less than threshold in the original matrix then the maximum subsquare size with the current element at i,j can be only 1 since we can only allow the top-right element to be less than threshold.
    - If the value of the current cell is less than threshold in the original matrix then the maximum subsquare size with the current element at i,j can be only 1 since we can only allow the top-right element to be less than threshold and the current element i,j is the bottom-right most element.
    - For all other cases, we can use this equation, top(i,j) = min(top(i-1,j-1),top(i-1,j-1),top(i-1,j-1)) + 1

Left DP Matrix:

$$\mathrm{OPT}(i,j) = \begin{cases} 1 \text{ if } i==0 \text{ or } j==0 \\ 1 \text{ if Matrix}(i-1,j-1) \text{ or Matrix}(i-1,j) \text{ or Matrix}(i,j) < h \\ \mathrm{MIN}(\mathrm{OPT}(i-1,j-1), \mathrm{OPT}(i-1,j), \mathrm{OPT}(i,j-1)) + 1 \text{ for all other cases} \end{cases}$$

Here OPT(i,j) is the side of the maximum square whose bottom right-most element is i,j such that only the bottom-left most cell of the square may be less than threshold.

For generating top DP matrix, for each i,j where i,j is the bottom right most element of square of side with the value at i,j:

- All the elements in the first row or first column are individual squares of size 1 and hence are valid squares since a single square can be considered a bottom-left corner and hence be ignored.
- For other elements:
    - If top-left or top-right cells have values less than threshold in the original matrix then the maximum subsquare size with the current element at i,j can be only 1 since we can only allow the bottom-left element to be less than threshold.
    - If the value of the current cell is less than threshold in the original matrix then the maximum subsquare size with the current element at i,j can be only 1 since we can only allow the bottom-left element to be less than threshold and the current element i,j is the bottom-right most element.
    - For all other cases, we can use this equation,
      left(i,j) = min(left(i-1,j-1),left(i-1,j-1),left(i-1,j-1)) + 1

For the final DP matrix we can use the precomputed values in the three auxiliary matrices to find the side of the largest square with bottom-right element at i,j by allowing corners to be less than threshold. For each i,j in the final DP matrix we need to pick the minimum of the three largest valid squares ending the cells on top, top-left and left and add 1 as proven in Task 3.

$$OPT(i,j) = \begin{cases} 1 \text{ if } i==0 \text{ or } j==0 \\ \\ MIN(OPT(i-1,j-1), OPT(i-1,j), OPT(i,j-1)) + 1 \\ \text{for all other cases} \end{cases}$$

Here OPT(i,j) is the side of the maximum square whose bottom right-most element is i,j such that the top-left most, top-right most and bottom-left most cells of the square may be less than threshold.

For generating final DP matrix, for each i,j where i,j is the bottom right most element of square of side with the value at i,j:

- All the elements in the first row or first column are individual squares of size 1 and hence are valid squares since a single square can be considered as any one of the four corners and hence be ignored.
- For other elements we can find the value using this equation,
  dp(i,j) = min(dp(i-1,j-1),dp(i-1,j-1),dp(i-1,j-1)) + 1

Task 6: Design a Θ(m^3*n^3) time Brute Force algorithm for solving Problem3

Design:

1. The brute force approach will try to find out all possible submatrices in the input matrix.
2. To find all possible submatrices we need to fix the four corners of the submatrix.
3. To fix the four corners, we need to fix horizontal corners (either top edge or bottom edge) and vertical corners (either left edge or right edge.)
4. As we have m rows, to fix two vertical corners we need m^2 iteration.
5. As we have n columns, to fix horizontal corners we need n^2 iteration.
6. So we can find all possible submatrices in m^2*n^2 iterations.
7. Once we find the submatrix, we need to check if it is a valid sub square with up to k elements which are allowed to be less than threshold.
8. If any of the sub squares has more than k elements less than threshold, we discard that particular solution.
9. We keep track of all found solutions and return the solution with maximum size.

Pseudocode:

```
1. 1.Initialize i1, j1, i2, j2 to None. These are the indices which will contain the top-left and
   bottom-right indices of the largest square where all elements in the square are greater than
   threshold
2. Initialize maxSquareSize to 0
3. Get the number of rows and columns in the matrix and store them in rows and cols
   respectively
4. Iterate over each possible sub square in the matrix using four nested loops, with indices i,
   j, k, and l where i,j will be the top-left element of the sub square and j,k will be the
   bottom right element of the sub square
5. Calculate the size of the square by taking squareSize = k-i+1 if k-i == l-j and k-i >= 0,
   otherwise set squareSize to 0
6. If squareSize is non-zero, count all the elements in the sub square which are less than the
   threshold. If count is less than or equal to k, update maxSquareSize, i1, j1, i2, j2 if
   squareSize is greater than or equal to maxSquareSize
7. Return i1, j1, i2, j2
```

Time and Space Complexity Analysis:

The two outer loops use i,j,k and l to go through all possible subsquares where i,j represents the top-left element of the subsquare and k,l represent the bottom right element of the subsquare. Since i and k loop through the all rows(m) and j and l loop through all the columns we will be doing Θ(m^2*n^2) operations to generate all possible subsquares. For each subsquare we traverse all elements to count the number of elements which are less than or equal to the threshold. Since this operation will require m*n iterations to check all elements of the subsquare the total time complexity will be Θ(m^3*n^3). As our algorithm keeps track of the largest square found in variables, we don't need extra space, hence the space complexity of the algorithm will be O(1).

Proof of Correctness:

Since we are generating all possible subsquares of the given matrix and checking if the number of elements less than threshold is less than or equal to k, we can be sure that the algorithm is correct.

Task 7: Design a Θ(m*n*k) time for solving Problem3

Task 7 has two parts, part A and part B. One implementation is using dynamic programming and the other is using recursion. Hence design of task 7A and 7B would be the same except for a few changes.

Design:

1. For finding the solution for Problem 3 in m*n*k time, the big giveaway is that for each element in our input matrix, we are allowed to spend k iterations to find the maximum square containing k values.
2. If we consider the approach similar to Task 1, 2 or 6, we will be required to check all elements, for that we can’t achieve our objective of locating k values in O(k) time.
3. Similarly if we consider the checking for subsquares by moving a diagonal element similar to task 4, we still need O(min(m,n)) which may or may not be close to O(k).
4. To complete the task in O(k) time, we need to consider only elements that are less than the threshold.
5. For the simplicity of explanation and solving the problem, we convert all the elements greater than or equal to threshold as 1 and less than the threshold as 0. This can be done in single pass in O(m*n) time.
6. Along with converting to 0s and 1s, we also single out the location of all 0s but in a separate list. We will be required to make k comparisons to get the desired solution from the separated list.
7. For each element, we iterate through the list, we first transform the list according to the perspective of the current element index. This can be done in a linear pass of k.
8. We form a square with k nearest 0s present in our list of 0s. If the square can be constructed, we calculate the number of 0s in the square in the submatrix.
9. The process of finding the zero can if we find the number of 1s and then subtracting it with maximum number of 1s possible in that square (which is side multiplied by side)
10. We discard the accommodated zeros in the square. In our current subsquare, we accommodate more than k 0s, we discard the solution and pick the previous solution, else we try to form a square with next 0. This can be done in the worst case of O(k) time. We store the size of solution in a variable.
11. We repeat the above steps for every element.can return the index of the element with largest size.

NOTE:

1. Task 7b requires another step at the start to generate the DP matrix of prefix sum. We have discussed the generation of Prefix Sum DP in detail in Algorithm 4 and have also proved its correctness.
2. Step 9 in design requires finding the number of 0s in a square. For Task 7a, we make a recursive function call to get the sum of all elements in the square, while for Task 7b, we refer to the DP matrix we generated earlier.

Pseudocode:

The method here is valid for both tasks: Task 7a and 7b except for step 2 and step 8.

Method

```
1. Change the representation of the input matrix to 1/0 matrix and store the 0s in a separate list using horizontal plus vertical traversal.
2. Initialize memoization DP for PrefixSum Method.
3. Iterate through each element in the matrix and Repeat step 3 to 13
4. Initialize maxSize=0, size = 0 and numZeros = n*m
5. Iterate through list of k 0s, for every element Repeat step 4 to 13
6. Adjust index of 0 according to index of element (such that element appears at origin 0,0) => x,y=> xi-x0,yi-y0.
7. If  a square can be formed from the element to the location of current 0. Do step 8-11, else step 12
8. Call PrefixSum(x,y) to get the Number of 1s in the square.
9. Number of 0s = size*size - Number of 1s
10. If number of 0s < k, update size, update numZeros  move to next 0 and Do step 9, Else Step 8.
11. Discard the solution, and return size => size - 1
12. If square cannot be formed, return maximum size of the square that can be formed excluding current 0 and update size
13. Check if size found for current element > maxSize, if yes update maxSize
14. Return maxSize with the index of the element.
```

PrefixSum Method for Task 7a

```
1. If x == 0 or y == 0 return 0
2. Else if PrefixSumDP[x][y] is not uninitialized, return PrefixSumDP[x][y]
3. Else return PrefixSumDP[x][y] = prefixSum(i - 1,j) + prefixSum(i,j-1) - prefixSum(i-1,j-1) + matrix[i - 1][j - 1]
```

PrefixSum Method for Task 7b

```
1. Return PrefixSumDP[x][y]
```

NOTE:

1. In step 2, Task 7a initializes DP with all the elements as -1. (PrefixSum recursive method populates the values in DP on the run).
2. In step 2, Task 7b initializes DP by using below steps.
    a. Initialized DP matrix of size m+1*n+1
    b. For i from 0 to m+1, Do c
    c. For j from 0 to n+1, Do d
    d. DP[i][j] = DP[i - 1][j] + DP[i][j - 1] - DP[i - 1][j - 1] + A[i - 1][j - 1]

Time and Space Complexity Analysis:

First step required us to change the representation of the matrix to 0s and 1s. This can be done in one pass i.e O(m*n) time. Similarly, separating the zeros will take an independent pass of O(m*n) time. We also need to compute PrefixSumDP which can be done in an independent pass of O(m*n). If we compute PrefixSumDP using recursion, it required The solution requires us to iterate over every element in the matrix, which in worst case requires O(m*n) time. For every element, we need the first k 0s after that element that are separated out in the list of 0s which take O(k) time. For every iteration in k, we are required to find the sum of all the elements in the formed subsquare from current element to current 0. This can be done in O(1) time. Therefore total time complexity will be O(m*n) + O(m*n) *O(k)*O(1) = O(m*n*k).
As our algorithm keeps track of all the 0s found in matrix that can max O(m*n), and we also need to compute prefixSumDP and storing it will require O(m*n) space. Hence the space complexity of the algorithm will be O(m*n).

Proof of Correctness:

The proof of correctness of the algorithm requires us to prove the correctness of PrefixSum generation as well as the proof of correctness of the rest of the steps. We have already argued the proof of correctness of PrefixSum generation in Section 4. The bellman equation for PrefixSum is as below:

$$\mathrm{OPT}(i,j) = \begin{cases} 0 & \text{if } i{=}{=}0 \text{ or } j{=}{=}0 \\ \mathrm{OPT}(i-1,j) + \mathrm{OPT}(i,j-1) - \mathrm{OPT}(i-1,j-1) \\ \quad + \mathrm{binaryMatrix}[i-1][j-1] & \text{For other cases} \end{cases}$$

Below formulation is given according to the 1-base index.

Definition

OPT(i,j): Sum of submatrix whose bottom right element is i,j

Goal
OPT(m,n): Sum of submatrix whose bottom right element is m,n, i.e. the entire matrix

Case 0:
If i or j =0, no element exists, so the sum would be 0.

Case 1a:
If i=1, no elements exist above, all the elements that are present w.r.t the current element(i,j) are to the right. Hence OPT(i,j) = A[i][j] + optimal solution of left element(i,j-1)

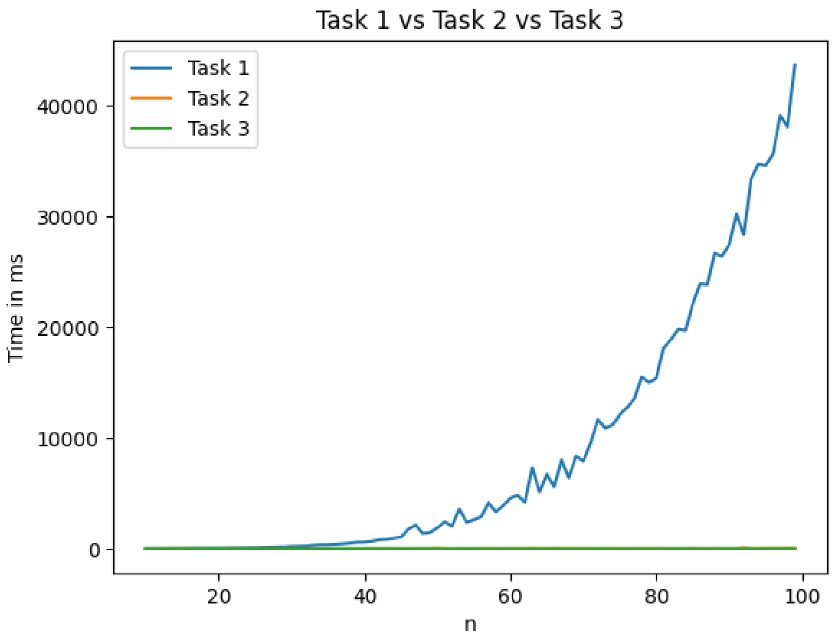
Case 1b:
If j=1, no elements exist right, all the elements that are present w.r.t the current element(i,j) are above it. hence OPT(i,j) = A[i][j] + optimal solution of above element(i-1,j)

Case 2:
If both i>1 and j>2, it means we have elements at the top, left as well as diagonally. We need to include the left and top element with the current element and exclude the top left diagonal element, (reasoning already given in Task 4). Hence, OPT(i,j) = A[i][j] + optimal solution of left element(i-1,j) + + optimal solution of above element(i-1,j) + optimal solution of left element(i,j-1) - optimal solution of above-left-diagonal element(i-1,j-1)

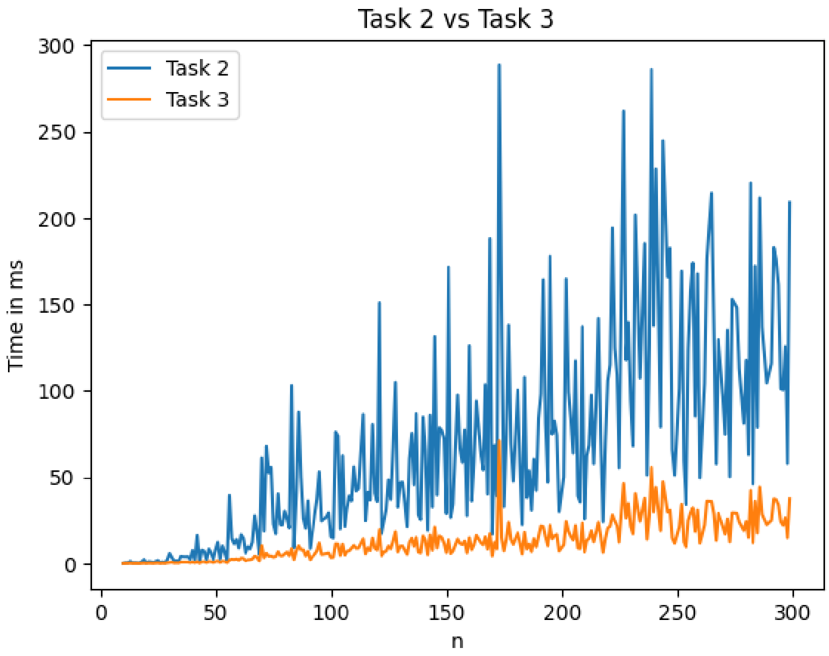
As we already argued the proof of correctness of the above equation in Task 4, we move to the next part, that is the algorithm yields the biggest subsquare of any element i,j. For any element i,j, we find the closest 0 that is available. The separation of 0s we did in 1st step gives us 0s in increasing order of that element. Once we find the 1st 0, we try to form a square with it, and check how many 0s are present in the square. We define the 1st 0 added and all the 0s added with that 0 as Layer 1 0s. We keep increasing the Layers of 0s, and keep track of total 0s in our solution. (We automatically skip Layer with no 0 as it won't be present in our list of separated 0). Consider, the number of 0s were valid for Layer K (0 in kth row or kth column) and invalid for Layer L (0 in lth row or lth column). Which means we can form a square from i,j to (max(i,j) of Layer K 0) but cannot form a square from i,j of (max(i,j) of Layer l 0). If L-K>1, it means the layers in the middle have all 1s. We can include those layers without checking them as they dont have any 0s. Hence, our solution returns the maximum possible square i.e square from i,j of size one less that max(vertical distance from i to Layer L 0, horizontal distance from i to Layer L 0).
If 0 count doesn't exceed k, it means k > all the 0s present in the matrix, which means we can consider the entire matrix , our solution will also return the max possible size i.e square from 1,1 to min((m,m),(n,n)).
We are iterating over every element, Hence the two for loops will always terminate. If 0 count exceeds k, then we terminate the solution before k iteration. Hence, the inner for loop will also terminate. Then the algorithm will always terminate.

Experimental Comparative Study

Problem 1:

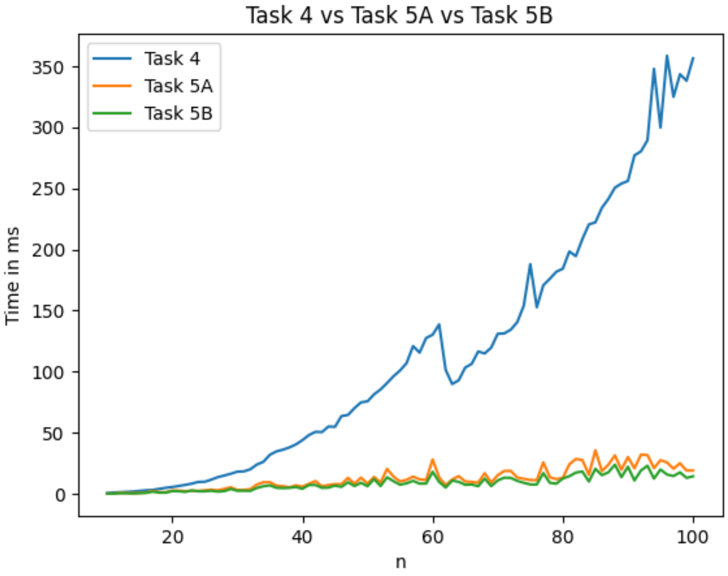


This result is a simulation of running times between Task 1, Task 2 and Task 3. We assumed the random input matrix to be of size nXn and plotted the graph with running time in ms in the Y axis and n from 1 to 100 in the x-Axis. Since the time complexity in terms of n for Task 1 is O(n^6) which is exponentially higher when compared to Task 2 O(n^4) and Task 1 O(n^2) we can see that Task 1 overshadows Task 2 and Task 3. To better analyze the difference between Task 2 and Task 3, we plot the below graph:

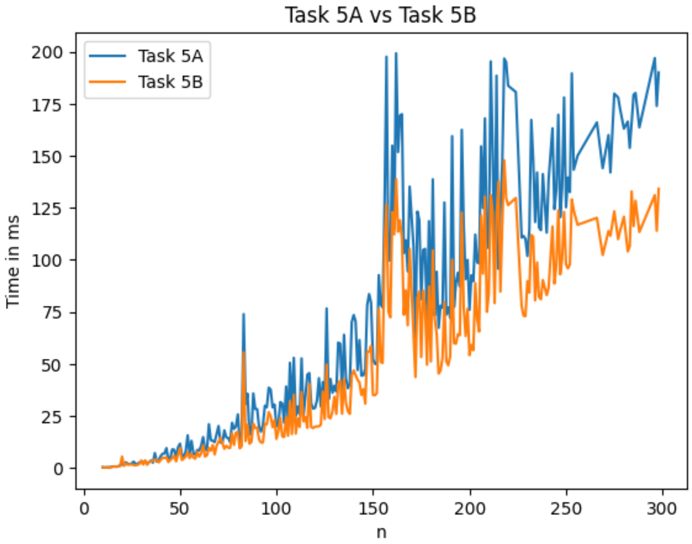


In this graph we can clearly see that Task 2 O(n^4) is much slower than Task 3 O(n^2) for all the iterations for n = 1 to 300.

Problem 2:

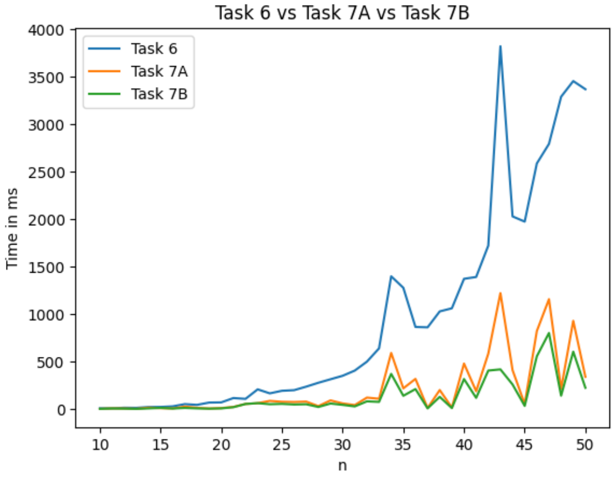


In this plot we see the comparison between Task 4 which is O(n^3) vs Task 5A and 5B which are both O(n^2) solutions. Here we can observe that Task 4 takes exponentially more time with increasing size of the matrix. For the sake of observation we used only square matrices in the inputs to better appreciate the comparison between different tasks. Another important observation in this graph is that the time taken by Task 5A is slightly higher than Task 5B for all values of n. Even though both the tasks have the same time complexity O(n^2). For a better comparison we plot the same graph but this time just for Task 5A and 5B below.



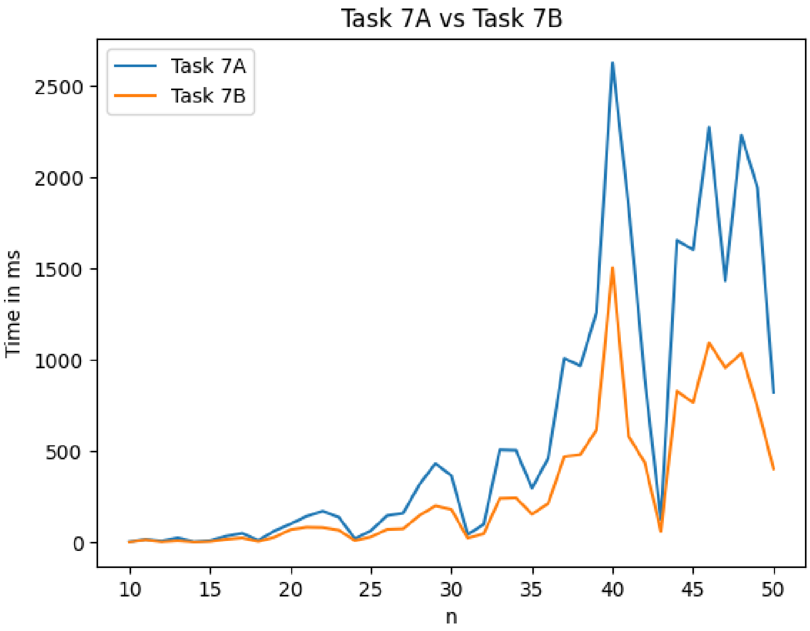
The only difference between Task 5A and 5B is that 5A uses recursion and memoization to solve the optimum sub problem whereas 5B uses bottom up dynamic programming. Even though both implementations perform the exact same number of operations and use the same space, recursion is slightly slower because every time a recursion call is made, there is some overhead in calling the new function like creating a stack frame and saving data in the registers which consumes CPU cycles.

Problem 3:



This graph shows the plot of Time vs n where nXn is the size of the input random matrix for Task 6, 7A and 7B. We used randomly generated matrices of size nXn with n ranging from 10 to 50 and random values of h and k. Task 6 is again exponentially slower since it is an O(n^6) solution which goes through all possible squares and finds the largest one which has upto k elements less than threshold. Task 7A and

7B are much better improvements since they utilize dynamic programming to reduce the time complexity to O(n^2k).



Just like in the case of Task 5A and 5B, even though Task 7A and 7B use the exact same algorithm and perform the exact same number of operations, Task 7A is slightly slower than 7B because 7A used top-down recursion and memoization to compute the solution whereas 7B uses bottom up dynamic programming. Multiple recursive calls(even though there will be at maximum m*n calls in a matrix of size mXn) will add additional overhead because of maintaining and pushing data on the recursion stack and context switching time associated with recursion. This causes the recursion + memoization approach to be slightly slower than bottom-up dynamic programming solution for the same problem.

Conclusion

We started our implementation with Algorithm 1 which is the brute force approach of Problem 1. We were asked to implement the algorithm in m^3*n^3 time. This time complexity was a big hint of what was needed to be done in this task. We straightaway realized that we need to solve the solution by considering all possible submatrices in the input matrix. Hence, we implemented algorithm 1 with no difficulty.

Then we moved to implementation of Algorithm 2 which requires to implement the Problem 1 in m^2*n^2 time. Now the time given was less, so considering all possible submatrices was not an option. But we don't need to find all possible submatrices, we just need to find all possible squares. Unlike a submatrix, which needs 4 parameters to represent (all four corners), a square can be represented using only 2 parameters, i.e. one corner and its size. Hence, as we were able to reduce 2 parameters, we were also able to reduce the runtime by the factor of m*n. In this case also, required runtime gave a big hint regarding implementation and we didn't face any challenge while implementing it.

Next, we moved onto implementation of Algorithm 3. The algorithm mentioned dynamic programming, hence we knew that if we figured out overlapping subproblems we would figure out the bellman equation and designing the algorithm would be straightforward. While implementing Algorithm 1 and 2, we already saw the patterns that were evolving while the valid subsquare was being formed. We have even given the diagrammatic representation of what we visualized in mind. Once we did this, writing the bellman equation and algorithm was easy. Figuring out the overlapping subproblem required a bit of thinking but the rest was simple.

Subsequently, we commenced with Algorithm 5. We temporarily suspended the resolution of Algorithm 4 and switched to Algorithm 5, as we had a hunch that it resembled Algorithm 3. As it turned out, our intuition was correct. Instead of borrowing the height of top, left and diagonal squares, we have to consider squares which allow one corner to have any value. So we had the idea but implementing it was somewhat challenging, as we had to maintain 3 separate DP matrices. We tried to trace sample examples on paper, and then wrote a program for it. First we wrote a bottom-up program, i.e. task 5b and then implemented task 5a. Implementation of task 5a was the most difficult programming task to implement in this assignment for us, as we missed out a few edge cases as we had to debug to find the problem. Hence, figuring out the algorithm was easy but converting it to code was challenging.

Then, we came back to Algorithm 4. Algorithm 4 required quite a bit of thinking. We tried various representations of OPT(i,j), finally we figured out a solution that was working. The space complexity of the algorithm was m*(m*n) as for every value of row index, we had to maintain a separate DP. Later we move to solving other problems. But later we came back the this same task as we figured out a better solution with less space complexity O(m*n). Hence we discarded the previous algorithm and modified the new one. We had to come back twice to solve this problem. Surprisingly, even though the runtime of the algorithm is worse than its successor, designing the algorithm was much more difficult.

Next, we started Algorithm 6 which took no time. With minor changes in Algorithm 1 and we got ourselves a brute force algorithm for Task 6. Hence, we implemented algorithm 6 with no difficulty.

Lastly, we arrived at Algorithm 7. By this stage, we had acquired considerable familiarity with the methodology of resolving such issues. We had previously tackled a recursive top-down task and derived the Bellman equation. Algorithm 7 also demanded significant contemplation. Nonetheless, the given time complexity proved to be a crucial hint in finding a solution. Both team members worked jointly to brainstorm potential solutions, and eventually, we succeeded in finding one. Our prior experience of solving Algorithm 4 and 5 proved to be immensely useful in resolving this problem. The most significant challenge was discerning the subtle nuances and similarities between this problem and the previous ones. Once we grasped the approach, implementing the top-down and bottom-up algorithms was a straightforward process.

Overall, it was a fantastic learning experience, and employing multiple techniques to solve the same problem helped us comprehend the methodology of tackling dynamic programming issues.