COT5405 Analysis of Algorithms Programming Project Report

Ananda Bhaasita Desiraju

[UFID: 40811191]

Instructions to run the code:

1. To compile all the source code files: **make**
2. To run: **java AlgoTowers [1-5]**
   java AlgoTowers 1 – Runs Task1
   java AlgoTowers 2 – Runs Task2
   java AlgoTowers 3 – Runs Task3
   java AlgoTowers 4 – Runs Task4
   java AlgoTowers 5 – Runs Task5
3. Input to the program:
   numOfRows numOfColumns minHeight
   elem01 elem02 elem03 ……
   elem11 elem12 elem13 ……
   …………………….
   …………………….
   ………
4. Another way to give input to the program:
   cat 10x10 | java AlgoTowers 1
5. Expected Output: Top left and bottom right coordinates of the matrix
   x1 y1 x2 y2
6. Cleaning the executables: **make clean**

Algorithm Design Tasks

Problem Statement: Consider a city in Florida named Gridville that has a grid layout of m x n cells. Associated with each cell (i, j) where i = 1,….,m and j = 1,….,n, Gridville architectural board assigns a non-negative number p[i, j] indicating the largest possible number of floors allowed to build on that block. A developer company named AlgoTowers is interested to and the largest possible area (shaped square or rectangle) of blocks within city limits that allows a building of height at least h.

Algorithm 1

To Do: Design a O(mn) time Dynamic Programming algorithm for computing a largest area square block with all cells have the height permit value at least h.

Pseudo code:

1) Construct an auxiliary matrix auxiliaryArray[rows][columns] for a given grid[rows][columns].

a) Copy first row and first column from the grid[][] and compare each element with minHeight. If element >= minHeight then fill in in the auxiliaryArray[][], else fill 0.

b) For the remaining elements of the grid, recursively update the auxiliary matrix starting from row 1 and column 1.

In order to construct the auxiliary matrix, the following expressions are used.

If grid[row][col] is greater than or equal to minHeight then,

auxiliaryArray[row][col] = min(auxiliaryArray[row-1][col-1], auxiliaryArray[row][col-1], auxiliaryArray[row-1][col]) + 1

else //If grid[row][col] is less than minHeight then

grid[row][col] = 0

2) Now, from the auxiliaryArray[rows][columns], find the maximum element, which will be the size of the largest square block with buildings of atleast minHeight.

3) With the help of the maxSize computed from the auxiliaryArray[row], the co-ordinates of the bottom right element in the auxiliary array can be determined and with the help of that and the size of the square, co-ordinates of the top left element can be computed.

Recursive formulation: The mathematical recursive formulation expressing the optimal substructures in this algorithm is

auxiliaryArray[row][col] = min(auxiliaryArray[row-1][col-1], auxiliaryArray[row][col-1], auxiliaryArray[row-1][col]) + 1

Walking through the algorithm to prove the correctness with the help of an example:

Base Cases:

If the given grid consists of only a single row then the size of the largest area square block will also be 1.

If the given grid consists of only a single column then the size of the largest area square block will also be 1.

Approach:

Firstly, we will create an auxiliary array consisting of the same number of rows and columns as the grid that is given as input. We will then fill the auxiliary array and find out the largest square area block in the given grid.

Filling the auxiliary array:

The first row and the first column of the grid are copied into the auxiliary array and while copying, each element is compared with the min height and 1 will be filled if the element in grid is greater than or equal to min height, else zero will be filled.

If the min height is 2 and the values of a given 5x5 grid are as follows:

{{1, 2, 6, 8, 2}, {9, 7, 0, 2, 2}, {7, 0, 4, 2, 9}, {3, 2, 3, 1, 9}, {7, 2, 9, 5, 6}}
The auxiliaryArray after filling the first row and first column will be:

	1	2	3	4	5
1	0	1	1	1	1
2	1				
3	1				
4	1				
5	1				

Now the rest of the auxiliary array will be filled by comparing the element at each position with min height. If the element is less than min height then the corresponding location in the auxiliary array is filled with 0. If the element is greater than or equal to min height then the then the corresponding location is updated with the Minimum (value in the left cell, top cell and left-top diagonal cell) + 1.

After filling the rest of the auxiliary array:

	1	2	3	4	5
1	0	1	1	1	1
2	1	1	0	1	2
3	1	0	1	1	2
4	1	1	1	0	1
5	1	2	2	1	1

There are 4 different square blocks possible (as highlighted in blue), and the area of the largest square block is 16. (4x4).
All the possible coordinates for the largest area square block are:

- Top left (x1, y1) and bottom right (x2, y2) co-ordinates are: 1 4 2 5
- Top left (x1, y1) and bottom right (x2, y2) co-ordinates are: 2 4 3 5
- Top left (x1, y1) and bottom right (x2, y2) co-ordinates are: 4 1 5 2
- Top left (x1, y1) and bottom right (x2, y2) co-ordinates are: 4 2 5 3

Time and Space Complexity Analysis:
Time complexity : O(mn) where m in the number of rows in the grid and n is the number of columns

Space complexity : O(mn) Another matrix of same size as the grid can be used. (or) O(n) Another matrix of the size number of columns in the grid can be used.

In the above mentioned algorithm for computing auxiliaryArray element of *ith* row we are only using the previous element and the (*i*−1)*th* row. So, inorder to simplify this we can use a 1D array instead of a 2D matrix.

To fill the 1D auxiliary array, we start with the array containing all zeroes in the beginning. And as we go along scanning the elements in the grid across a row, we update the auxiliary matrix based on the equation:

auxiliaryArray[col]=min(auxiliaryArray[col-1], auxiliaryArray[col], prev)

where prev refers to the old auxiliaryArray[col-1]. The same process is repeated for all rows in grid. This approach reduces the space complexity from O(mn) to O(n).

Algorithm 2

To Do: Design a O(m3n3) time Brute Force algorithm for computing a largest area rectangle block with all cells have the height permit value at least h.

Pseudo Code:

```
for lowerLeft.x = 0 .. n
   for lowerLeft.y = 0 .. m
     for upperRight.x = lowerLeft.x .. n
       for upperRight.y = lowerLeft.y .. m
         if checkBuildingHeight(grid, upperRight.x, upperRight.y, lowerLeft.x, lowerLeft.y,
minHeight))
           maxArea = Math.max(maxArea, calculateAreaOfRect(upperRight.x, upperRight.y,
lowerLeft.x, lowerLeft.y));
           //if maxsize is less than max area, compute the coordinates
           if (maxSize < maxArea) {
             maxSize = maxArea;
             x1 = lowerLeft.x + 1;
             y1 = upperRight.x + 1;
             x2 = lowerLeft.y + 1;
             y2 = upperRight.y + 1;


//compute area of the rectangle
calculateAreaOfRect(int x1, int x2, int y1, int y2)
  return (x2 - x1 + 1) * (y2 - y1 + 1);


//if building height is less than the min height, return false else return true
checkBuildingHeight(int[][] grid, int x1, int x2, int y1, int y2, int minHeight)
  for (i = y1; i <= y2; i++) {
    for (j = x1; j <= x2; j++) {
      if (grid[i][j] < minHeight)
        return false;
  return true;
```

Walking through the algorithm to prove the correctness:

For this brute force approach, there are clearly a finite number of sub arrays that are distinct for a given grid. So, all the possibly enumerate all the possible sub rectangles and check if they all consist of 1's. The brute force algorithm mentioned above, enumerates all the sub rectangles of the grid by selecting each element of the grid as a possible lower-left corner of the largest area rectangle block. And then for each lower left, every possible upper-right corner is also computed. When the lower left and upper right coordinates are finalized, the algorithm the checks whether the selected rectangle consists of only 1's and also checks whether it is the largest possible rectangle. If the selected rectangle does not have the max area, it will not be scanned for 1's. This approach sure does give the largest area rectangle block but takes a huge amount of time to do so.

Time and Space Complexity Analysis

Time complexity: O(m^3n^3) where m is the number of rows and n is the number of columns.

For selecting a lower left element, there are mxn possibilities. And for each of those lower left elements, there are mxn possible choices for an upper right corner element. So, on average for every lower left, there are O(mn) possibilities for an upper right corner. The total sub rectangles possible will be O(m2n2) and the for each of these sub rectangles, the average number of elements present in them is O(mn), making the worst case time complexity of this algorithm to be O(m3n3).

Space complexity: O(1)

This algorithm does not require any additional storage.

Algorithm 3

To Do: Design a O(mn) time Dynamic Programming algorithm for computing a largest area rectangle block with all cells have the height permit value at least h.

Pseudo Code:

```
1. Given an mxn grid matrix, compute the largest area rectangle by keep track of height,
   left and right of each element of grid rows.
2. height, left and right matrices are of size cols (number of columns in the grid) and all
   are initialized to zero.
3. The height matrix is updated using the following equation:
   if row[col] >= minHeight
           height[col] = height[col-1] + 1
   else
           height[col] = 0
4. The left matrix, which is the left bound of the rectangle is updated using the equation:
           left[col] = max(left[col - 1], currentLeft)
   The value of current left is one greater than rightmost occurrence of zero
   encountered in our rectangle.
5. The right matrix can be updated using:
           right[col] = min(right[col - 1], currentRight)
   The value of current right is the leftmost occurrence of zero that was encountered.
6. Area of the largest rectangle block is computed using:
           maxArea = height[col] * (right[col] - left[col])
7. Once the area and the left, right and height are computed, the top left and bottom
   right coordinates of the rectangle can be computed.
```

Recursive formulation: The mathematical recursive formulation expressing the optimal substructures in this algorithm are:

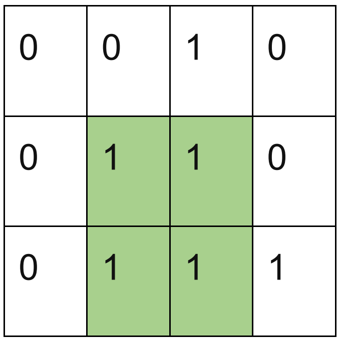
height[col] = height[col-1] + 1

left[col] = max(left[col - 1], currentLeft)

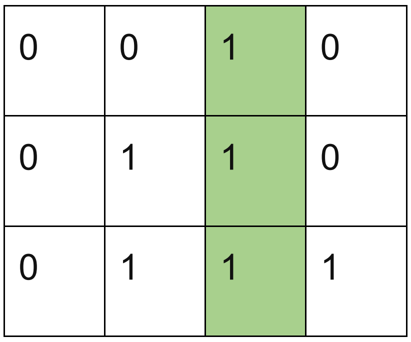
right[col] = min(right[col - 1], currentRight)

Walking through the algorithm to prove the correctness:

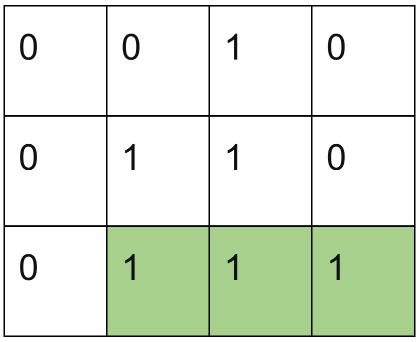
Given a matrix, a rectangle can be computed from it by computing height, left and right boundaries. Consider the following examples:



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



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



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

For the first rectangle, height is 2, left boundary is 1 and the right boundary is 2

For the second rectangle, height is 3, left boundary is 2 and the right boundary is 2

For the third rectangle, height is 1, left boundary is 1 and the right boundary is 3

These above examples are based on the truth that first rectangle is the largest rectangle of height 2, second rectangle is the largest rectangle with height 3 and the third rectangle is the largest rectangle with height 1. The largest area rectangle block will be one of these rectangles.

Time and Space Complexity Analysis:
Time complexity: O(mn) where m in the number of rows in the grid and n is the number of columns

Space complexity: O(m) where m is the length of the additional arrays.

Programming Tasks

Programming Task 1

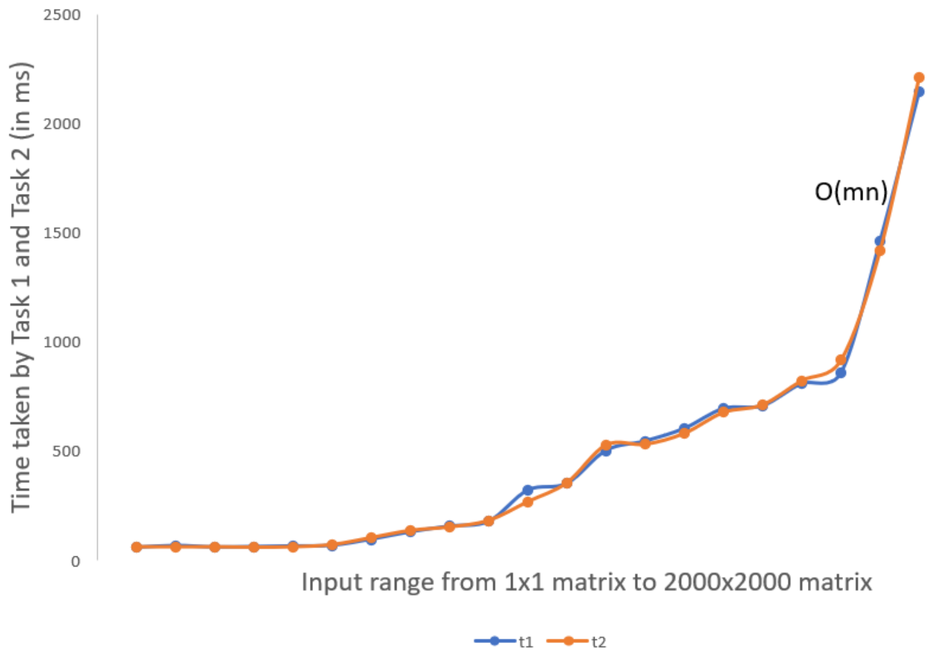
The first task was to give a recursive implementation of Algorithm1 using memoization and O(mn) extra space. This was similar to a maximal square DP problem with a slight modification. In the maximal square problem, the matrix is filled with zeroes and ones. Here the grid we have consists of heights (integers) between 0-9. The implementation was slightly modified to first convert the problem into a maximal square problem and then I was able to solve it. I also had to incorporate recursion into the problem I did follow bottom up approach even for the first task. The auxiliary array took an extra space O(mn) and this problem was solved in O(mn) time.

Programming Task 2

The second task was to implement an iterative BottomUp implementation of Alg1 using O(n) extra space. The previous task was given a space limit of O(mn). Task1 can be optimised to task2. This is also similar to maximal square DP problem with a similar modification. The implementation was slightly modified to first convert the problem into a maximal square problem. I used an iterative bottom up approach for this task. The auxiliary array took an extra space O(n) and this problem was solved in O(mn) time.

Experimental Comparative Study of Task1 vs Task2:

Both task1 and task2 take O(mn) time and hence while comparing the performances, I didn’t notice much difference in the time taken by each of these tasks to complete execution. However the only change is the space utilized. The below plot shows the comparision of execution times for task 1 (in blue) and task2 (in orange) and we observe that both the plottings overlap. The graph below shows execution times upto matrices of size 2000x2000.



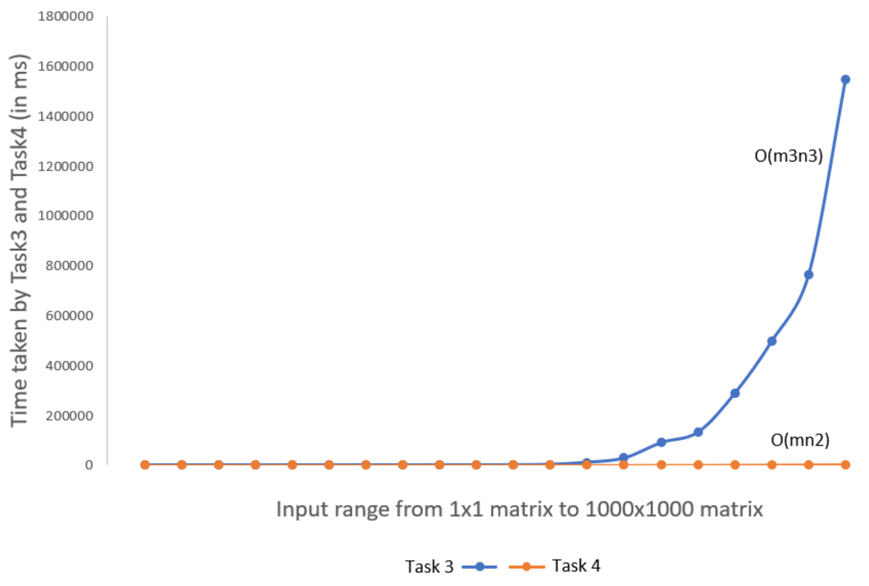
Programming Task 3

The third task was to implement Alg2 using O(1) extra space. This was similar to a maximal rectangle DP problem with a slight modification. In the maximal rectangle problem, the matrix is filled with zeroes and ones just like the maximal square problem. Here the grid we have consists of heights (integers) between 0-9. The implementation was slightly modified to first convert the problem into a maximal rectangle problem and then I was able to solve it. This was a brute force approach. The implementation was slightly modified to first convert the problem into a maximal rectangle problem. This task was given a space limit of O(1) and a time limit of O(m3n3) and I was able to solve the problem within those space and time constraints.
.

Programming Task 4

The fourth task was to implement Alg3 using O(mn) extra space. This was also a maximal rectangle DP problem with a slight modification. The implementation was slightly modified to first convert the problem into a maximal rectangle problem and then I followed an iterative bottom up approach to solve the problem. This task was given a space limit of O(mn) and a time limit of O(mn). However I was able to solve this algorithm in O(mn2) time and O(mn) extra space.

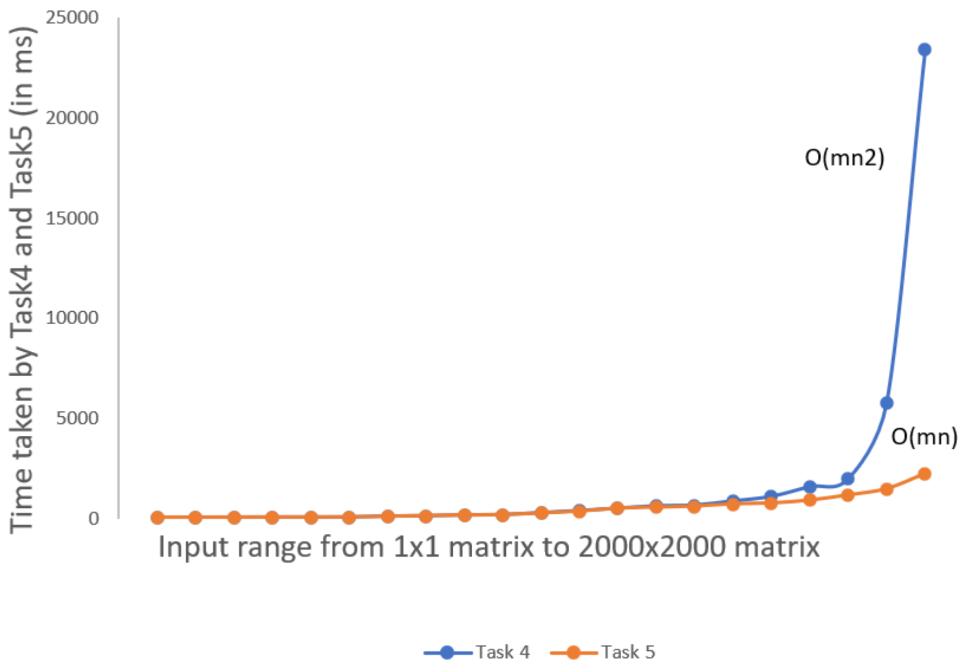
Experimental Comparative Study of Task3 vs Task4
Since task3 takes O(m3n3) time and task4 take O(mn2) time, while comparing the performances, I noticed difference in the time taken by each of these tasks to complete execution as the input size kept increasing. The below plot shows the comparision of execution times for task 3 (in blue) and task4 (in orange) and we observe that both the plottings overlap during the initial phases but as the input size increases, the execution time of task3 starts increasing exponentially. I was able to collect the execution times of matrices upto 1000x1000 for task3. After that point, it was taking forever to execute for input size of 1500x1500. It will probably be a little faster for input sizes between 1000x1000 and 1500x1500. Task4 was however faster compared to task3 and I was able to collect data upto the size 2000x2000 without any isuues.



Programming Task 5

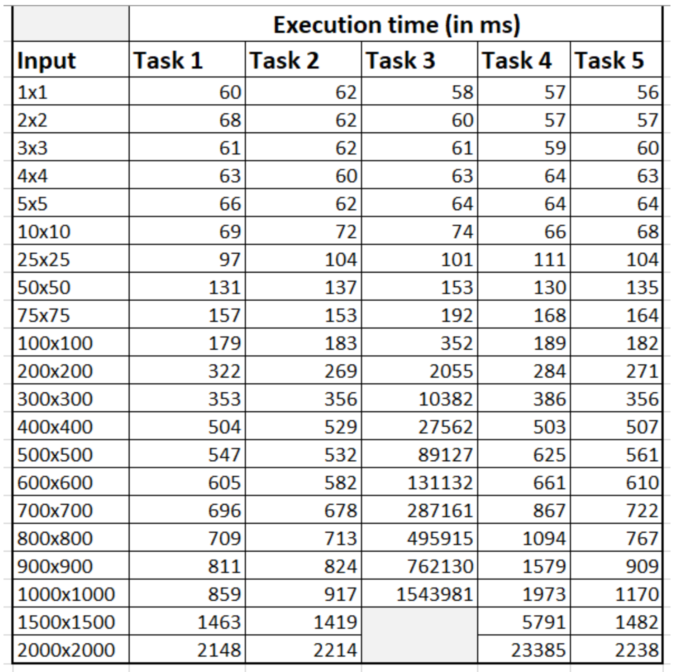
The fifth task was to implement Alg3 using O(n) extra space. Task4 can be further optimised to task5. This was also a maximal rectangle DP problem with a slight modification. The implementation was slightly modified to first convert the problem into a maximal rectangle problem and then I followed an iterative bottom up approach to solve the problem. This task was given a space limit of O(n) and a time limit of O(mn). I was able to solve this algorithm in the given time and space constraints.

Experimental comparative study of Task4 vs Task5:



Since task4 takes O(mn2) time and task5 takes O(mn) time, while comparing the performances, I noticed difference in the time taken by each of these tasks to complete execution as the input size kept increasing. The above plot shows the comparision of execution times for task 4 (in blue) and task 5 (in orange) and we observe that both the plottings overlap during the initial phases but as the input size increases, the execution time of task4 starts increasing exponentially. Task4 was slower than task5. I was able to collect data upto the size 2000x2000 without any isuues.

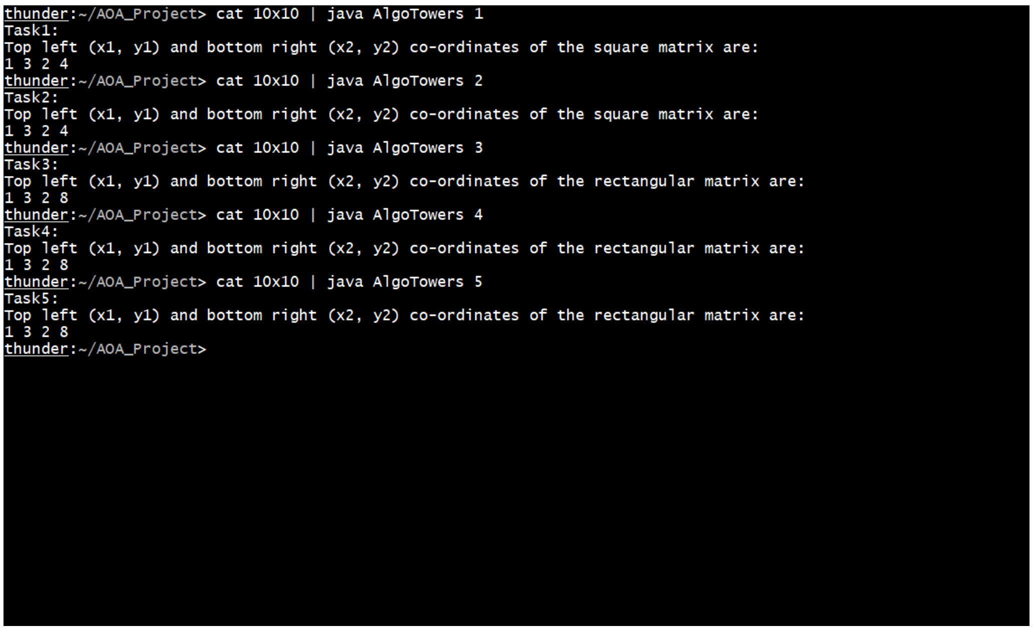
Execution Time Data Collected for each task:



	Execution time (in ms)				
Input	**Task 1**	**Task 2**	**Task 3**	**Task 4**	**Task 5**
1x1	60	62	58	57	56
2x2	68	62	60	57	57
3x3	61	62	61	59	60
4x4	63	60	63	64	63
5x5	66	62	64	64	64
10x10	69	72	74	66	68
25x25	97	104	101	111	104
50x50	131	137	153	130	135
75x75	157	153	192	168	164
100x100	179	183	352	189	182
200x200	322	269	2055	284	271
300x300	353	356	10382	386	356
400x400	504	529	27562	503	507
500x500	547	532	89127	625	561
600x600	605	582	131132	661	610
700x700	696	678	287161	867	722
800x800	709	713	495915	1094	767
900x900	811	824	762130	1579	909
1000x1000	859	917	1543981	1973	1170
1500x1500	1463	1419		5791	1482
2000x2000	2148	2214		23385	2238

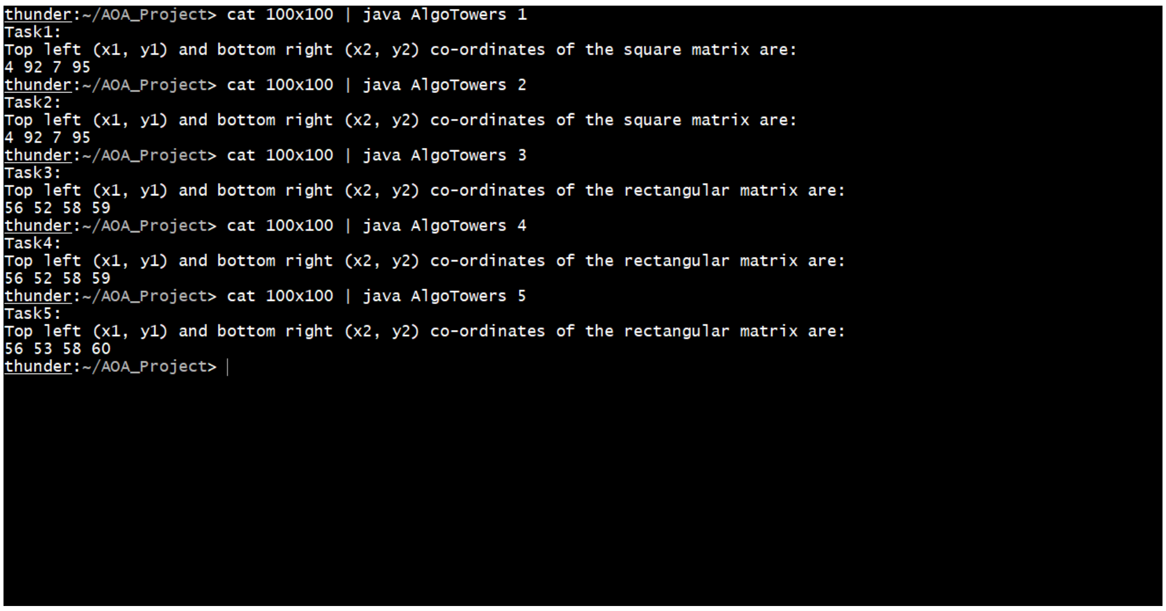
Output Screenshots

10x10 grid matrix:



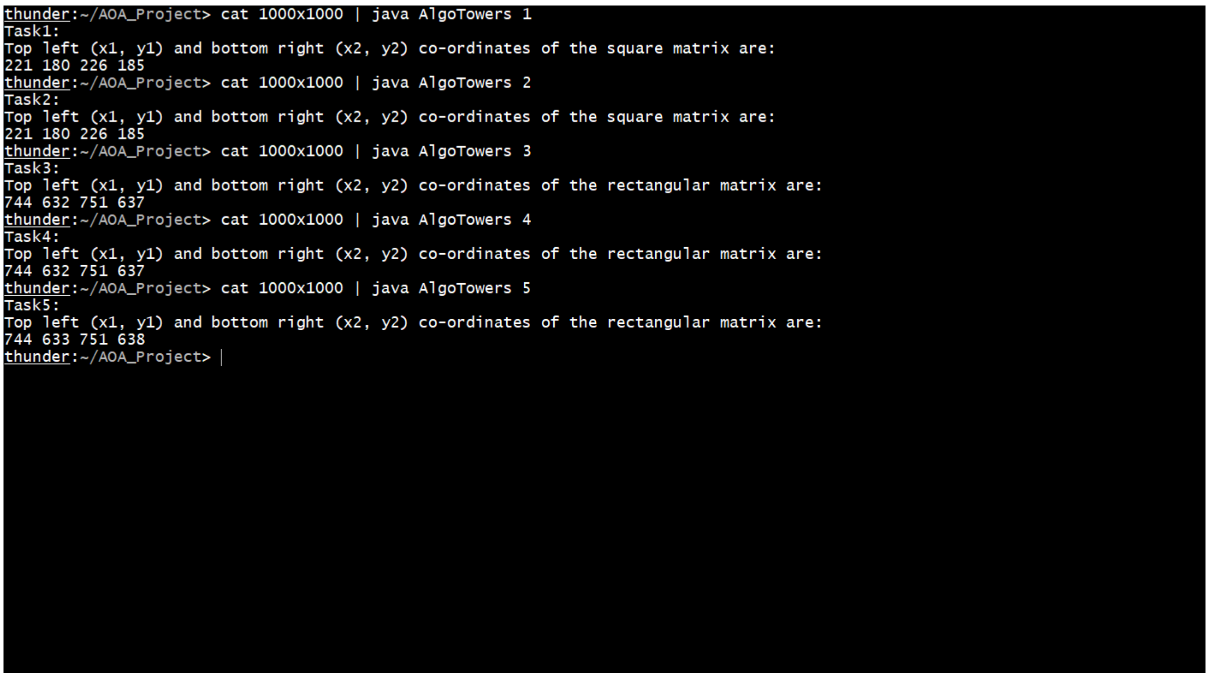
```
thunder:~/AOA_Project> cat 10x10 | java AlgoTowers 1
Task1:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the square matrix are:
1 3 2 4
thunder:~/AOA_Project> cat 10x10 | java AlgoTowers 2
Task2:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the square matrix are:
1 3 2 4
thunder:~/AOA_Project> cat 10x10 | java AlgoTowers 3
Task3:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the rectangular matrix are:
1 3 2 8
thunder:~/AOA_Project> cat 10x10 | java AlgoTowers 4
Task4:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the rectangular matrix are:
1 3 2 8
thunder:~/AOA_Project> cat 10x10 | java AlgoTowers 5
Task5:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the rectangular matrix are:
1 3 2 8
thunder:~/AOA_Project>
```

100x100 grid matrix:



```
thunder:~/AOA_Project> cat 100x100 | java AlgoTowers 1
Task1:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the square matrix are:
4 92 7 95
thunder:~/AOA_Project> cat 100x100 | java AlgoTowers 2
Task2:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the square matrix are:
4 92 7 95
thunder:~/AOA_Project> cat 100x100 | java AlgoTowers 3
Task3:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the rectangular matrix are:
56 52 58 59
thunder:~/AOA_Project> cat 100x100 | java AlgoTowers 4
Task4:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the rectangular matrix are:
56 52 58 59
thunder:~/AOA_Project> cat 100x100 | java AlgoTowers 5
Task5:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the rectangular matrix are:
56 53 58 60
thunder:~/AOA_Project> |
```

1000x1000 grid matrix:



```
thunder:~/AOA_Project> cat 1000x1000 | java AlgoTowers 1
Task1:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the square matrix are:
221 180 226 185
thunder:~/AOA_Project> cat 1000x1000 | java AlgoTowers 2
Task2:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the square matrix are:
221 180 226 185
thunder:~/AOA_Project> cat 1000x1000 | java AlgoTowers 3
Task3:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the rectangular matrix are:
744 632 751 637
thunder:~/AOA_Project> cat 1000x1000 | java AlgoTowers 4
Task4:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the rectangular matrix are:
744 632 751 637
thunder:~/AOA_Project> cat 1000x1000 | java AlgoTowers 5
Task5:
Top left (x1, y1) and bottom right (x2, y2) co-ordinates of the rectangular matrix are:
744 633 751 638
thunder:~/AOA_Project> |
```