COT 5405 - Project Report, Mohammad Rahman, 64199132

1 INTRODUCTION

A hypothetical city in Florida named Gridville is divided into a grid layout of $m \times n$ cells (a two-dimensional matrix). Each cell is associated with a value cell (i, j) that represents the maximum allowed height of a building in that cell. Our goal is to maximize the largest possible area of blocks within city with limit of at least h. Therefore, the input to the algorithm would be as follows:

(1) Size of the matrix in number of rows (m) and columns (n) along with the height limit h.
(2) Rows of the matrix (M) including all the elements of all columns in that row, $c(i, j),\ (i, j) \in (m, n)$.

The output of the program will be the coordinates of the maximum area in the form of top-left corner and bottom-right corner that satisfies the given requirements.

2 MODELING OF THE PROBLEM

The problem asks to maximize the area of cells that satisfies the height limit of h. Therefore, any cell that satisfies the constraint $c(i, j) >= h$ are of equal priority here. The collection of such cells that maximizes the area towards x and y direction for the given shape requirement (either rectangular or square) will be out area of interest. Therefore, we can model the given grid layout matrix M by following condition and prepare a new array S.

$$for\ (i, j) \in (m, n),\ S(i, j) = \begin{cases} 1, if\ M(i, j) >= h \\ 0,\ otherwise. \end{cases} \tag{1}$$

Once the above modeling is done, then how to handle the array elements of S, will decide how much extra space the program is going to incur.

3 ALGORITHM DESIGN

In this section we describe the way we design and develop the algorithms mentioned in the project guideline.

3.1 ALG1: Dynamic Programming Algorithm for largest square block with Space Complexity O(mn)

The task is to design a $O(mn)$ time dynamic programming algorithm for computing the largest area square block with cells having the height permit value at least h.

3.1.1 Mathematical Recursion of Optimal Substructure: Now consider an example of 5×5 grid layout matrix M below at the left and its associated S matrix in the right.

$$\begin{bmatrix} 3 & 9 & 6 & 5 & 4 \\ 8 & 0 & 5 & 8 & 9 \\ 1 & 5 & 6 & 0 & 1 \\ 0 & 1 & 7 & 1 & 1 \\ 2 & 6 & 0 & 5 & 7 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

One way to calculate the maximum square area cells that satisfies the height limit h, is to enumerate through all the row, column coordinates of (i, j) and keep track of the area that is maximum and also each of whose cells satisfies the height limit h. If we look close to the above matrix (2), we can clearly notice that, once we have already calculated the maximum area for S(2,4), we no longer need to calculate the area for S(2,5) from scratch. We can rather use memoization and utilize the area that is already calculated for S(2,4) to calculate the area for S(2,5). Therefore,

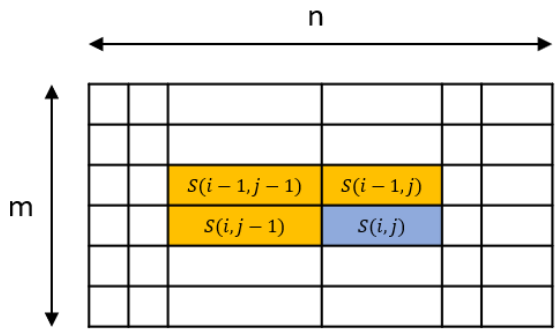

Fig. 1. Recursion for Dynamic programming to calculate maximum square block with $O(mn)$ space.

we can calculate the are for smaller sub-problems and cache those values. Later calculate area for a larger problem, we can utilize those areas of smaller sub-problems to calculate the area of the larger sub-problem. Therefore, the idea of the algorithm is to construct an intermediate size matrix $S(m, n)$ in which each entry $S(i, j)$ represents size of the square sub-matrix with all the cells with height limit at least h including $M(i, j)$. The optimal substructure of the recursive dynamic programming can be represented as below equation (2).

$$S(i,j) = \begin{cases} min(S(i,j-1), S(i-1,j), S(i-1,j-1)) + 1;\ if\ (i,j)\ >\ 0\ and\ M(i,j) >= h \\ 1;\ if\ (i,j)\ ==\ 0\ and\ M(i,j) >= h \\ 0;\ otherwise \end{cases} \tag{2}$$

3.1.2 Optimal Substructure Correctness:

According to the above equation (2), at each of the step, the algorithm enumerates through all the possible values of $(i, j) \in (m.n)$ sets the corresponding auxiliary matrix $S(i, j)$ to 0 if $M(i, j) < h$. The algorithm sets $S(i, j)$ to 1 if $M(i, j) >= h$ to assert the boundary condition. For all the other values of (i, j) *when* $1 < (i, j) <= (m, n)$ *and* $M(i, j) >= h$, the algorithm takes the minimum of its neighbouring $S(i, j)$ values that are smaller than or equal to (i, j) and adds 1 for its own. For example, for the given grid matrix M below at the left, the corresponding $S(i, j)$ matrix is provided at the right.

$$\begin{bmatrix} 3 & 9 & 6 & 5 & 4 \\ 8 & 0 & 5 & 8 & 9 \\ 1 & 5 & 6 & 0 & 1 \\ 0 & 1 & 7 & 1 & 1 \\ 2 & 6 & 0 & 5 & 7 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 2 & 2 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Once the auxiliary matrix is formed, the algorithm's job will be to find the maximum entry is $S(i, j)$. Using the value and the coordinates of the maximum entry, the algorithm can return the top-left and bottom-right coordinate of the maximum square area.

3.1.3 Analysis of the Algorithm: The algorithm requires following complexity in terms to time and space.

- **Time Complexity:** The algorithm needs to enumerate through all the possible values of (i, j) to keep the optimal substructure and return maximum square area coordinate. Therefore, its time complexity is $O(m \times n)$.
- **Space Complexity:** The algorithm needs to create an equal size auxiliary matrix of the given grid layout of size $m \times n$. Therefore, the space complexity is $O(m \times n)$.

3.2 ALG1: Dynamic Programming Algorithm for largest rectangle block with Space Complexity $O(n)$

The overall approach for this algorithm and the one described in subsection 3.1 is pretty much similar except the fact that how we handle the auxiliary array to keep track of the largest square block in this algorithm.

3.2.1 Mathematical Recursion of Optimal Substructure: For an $O(n)$ space algorithm, rather

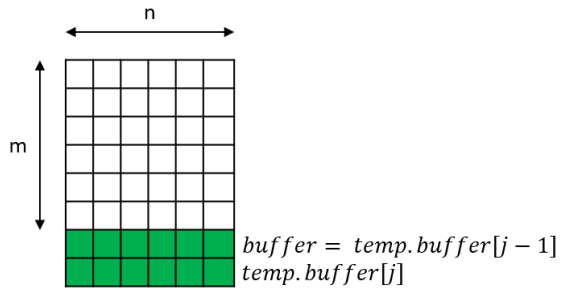

Fig. 2. Recursion for Dynamic programming to calculate maximum square block with $O(n)$ space by using two intermediate buffer of size n.

than having an auxiliary matrix of equal size, we create two auxiliary array $buffer,\ and\ temp.buffer$ of size n to keep track of the local maximum and global maximum of the corresponding column. We keep track of the maximum area so far of the most recent passed row at the $buffer(j)$ and current row at the $temp.buffer[j]$. If the current cell value $c(i, j) >= h$, then we set the corresponding $temp.buffer(j)$ to 1, if that cells is in the top or left boundary, i.e., $i\ or\ j = 0$. Otherwise, we set the min as the minimum of current value of $buffer(j)$, $buffer(j-1)$, and $temp.buffer(j-1)$. The associated recursive formula is shown below for $M(i, j) >= h$.

$$temp.buffer(j) = \begin{cases} 1; \\ min(buffer[j], buffer[j-1], temp.buffer[j-1]) + 1; \end{cases} \tag{3}$$

For $M(i, j) < h$, $temp.buffer = 0$.

3.2.2 Optimal Substructure Correctness:

In the above recursion, the auxiliary array $temp.buffer$ keeps track of the maximum is each column. At the same time, there is a one element array $currentMax$ that compares the current maximum square size with the maximum square of the corresponding column. After each row is compiled, the whole $temp.buffer[j]$ is copied to $buffer[j]$ to keep track of the immediate previous row. This way the algorithm confirms the correctness. This way we reduce the space complexity to $O(m + m) \rightarrow O(2m) \rightarrow O(m)$.

3.2.3 Analysis of the Algorithms: The algorithm requires following complexity in terms to time and space.

- **Time Complexity:** The algorithm needs to enumerate through all the possible values of (i, j) to keep the optimal substructure and return maximum square area coordinate. Therefore, its time complexity is $O(m \times n)$.
- **Space Complexity:** The algorithm needs to create 2 auxiliary 1-D array of n. Therefore, the space complexity is $O(n)$.

3.3 ALG2: Brute Force Algorithm for largest rectangle block

The task is to design a Brute force algorithm for computing the largest area rectangle block with cells having the height permit value at least h. To implement this algorithm, we need to consider two coordinate $(i1, j1)$ and $(i2, j2)$ that represents the top-left and bottom-right corner of the sub-matrix respectively. We need to enumerate both $(i1, j1)$ and $(i2, j2)$ through all the possible values of them $(1 \longrightarrow m, \ and \ 1 \longrightarrow n)$ and calculate the area of the block bounded by $(i1, j1)$ and $(i2, j2)$.

Analysis of the Algorithms:

The algorithm requires following complexity in terms to time and space.

- **Time Complexity:** The algorithm needs to enumerate through all the possible values of $(i1, j1)$ and $(i2, j2)$ from $(1 \longrightarrow m, \ and 1 \longrightarrow n)$. This portion requires 4 nested for loops. Then to calculate the area of the block bounded by $(i1, j1)$ and $(i2, j2)$, we need to runs 2 more nested for loops inside the 4 for loops. Worst case running time for this inner 2 for loops could be $O(mn)$. Therefore, the overall time complexity is $O(m^3.n^3)$.
- **Space Complexity:** The algorithm does not need to create any extra variables just one entry to keep track of the maximum area size which is also independent of the size of the input. Therefore, space complexity is $O(1)$.

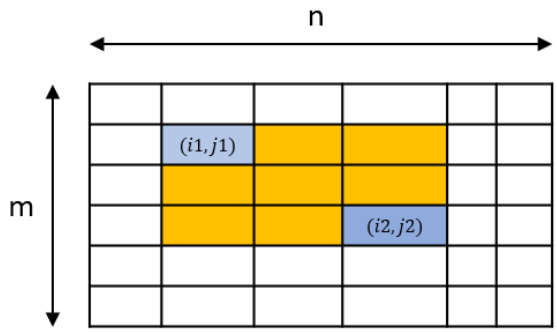

Fig. 3. Brute Force Algorithm for $O(1)$ space to calculate largest rectangle block of height limit h.

3.4 ALG3: Dynamic Programming Algorithm for largest rectangle block with $O(mn)$ extra space

This algorithm specification is pretty much similar to the one described in subsection 3.1 except the fact that, rather than calculating the maximum square block, here we need to calculate the largest rectangle block.

3.4.1 Mathematical Recursion of Optimal Substructure:

For this algorithm to calculate the maximum rectangle block, we keep track of maximum length of contiguous blocks that has height limit at least h in either direction and keep track of the corresponding area. To accomplish this, we keep two handler named $x = m, \ and \ y = 1$, where x tracks the length of the contiguous cells of height $>= h$ along a particular row and y tracks the maximum contiguous cells of height $>= h$ along a particular column.

$$max.x(i, j) = \begin{cases} 1 \ if \ M(i, j) >= h \ \& \ j == 0 \\ max.x(i, j-1) + 1 \ if \ M(i, j) >= h \ \& \ j > 0 \end{cases} \tag{4}$$

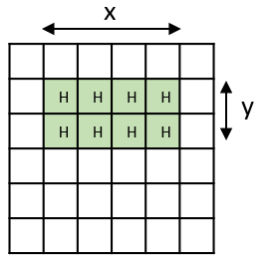

Fig. 4. Dynamic Programming Algorithm for largest rectangle block with $O(mn)$ extra space.

3.4.2 Optimal Substructure Correctness:

Everytime a row is processed in this algorithm, we increment y, if the current $M(i, j) >= h$. At the same time, we keep tracking the length of the contiguous cells of at least height limit h for each of the (i, j) cell in the above recursion equation of $max.x$. Therefore, at any particular iteration of the loop, if the area of $x \times y > currentarea$, then we store the coordinates for that (i, j). At the end of the program, we are left of maximum $x \times y$ and the coordinate bottom-right (i, j) where it occurs. We use the x, y values to extract the top-left coordinate from bottom-right one.

3.4.3 Analysis of the Algorithms:

The algorithm requires following complexity in terms to time and space.

- **Time Complexity:** The algorithm needs to enumerate through all the possible values of (i, j) to keep the optimality of the recursive substructure. Therefore, the time complexity of this algorithm is $O(mn)$.
- **Space Complexity:** The algorithm requires an auxiliary 2-D matrix $max.x$ to keep length of the maximum contiguous cells of height limit h in the x direction. Therefore the space complexity is $O(mn)$.

3.5 Bonus Algorithm: Dynamic Programming Algorithm for largest rectangle block with $O(n)$ extra space

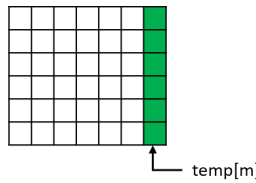

Fig. 5. Dynamic Programming Algorithm for largest rectangle block with $O(n)$ extra space.

This algorithm is also a dynamic programming algorithm that needs to calculate the maximum rectangle block of cell height limit h. The different with ALG3 is that, this one needs to finish the job with a tighter space constraint of $O(n)$. To beat the space constraint, I used Kadane's algorithm [1]. The idea is to fix the left and right corner of the columns one by one and find the maximum sum contiguous rows for every left and right column pair. this way we find the top and bottom row number for every fixed left and right column pair. In order to find the top and bottom row numbers, we calculate sum of elements in every row from left to right and store these sums in a $temp$ array. So $temp(i)$ indicates sum of elements from left to right in row i. By applying Kadane's algorithm on $temp$, and get the maximum sum subarray of the $temp$, this maximum sum would be the maximum possible sum with left and right as boundary columns. To get the overall maximum sum, we compare this sum with the maximum sum so far.

4 PROGRAMMING TASK

There are 4 programming tasks assigned in this project along with a bonus task.

- Programming task 1 requires to implement ALG1 using memoization and $O(mn)$ extra space.
- Programming task 2 requires to implement ALG1 using bottom-up dynamic programming and $O(n)$ extra space.
- Programming task 3 requires to implement ALG2 using brute force and $O(1)$ extra space.
- Programming task 4 requires to implement ALG3 using bottom-up dynamic programming and $O(mn)$ extra space.
- Programming task 5 (bonus) requires to implement ALG3 using bottom-up dynamic programming and $O(n)$ extra space.

I have used C-programming to implement each of these 5 tasks. All the programming tasks as written in one single file (AlgoTowers.c) under one main function. The tasks are organized and compiled according to the way the project submission guideline mentioned.

5 EXPERIMENTAL STUDY

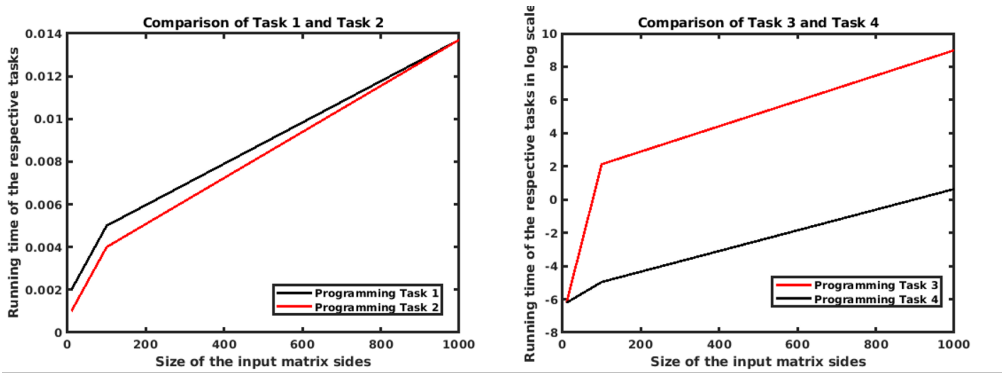

Fig. 6. Comparison between running time of different tasks. All the plots along the y-axis are in seconds. (a) Comparison between Task 1 and Task 2; (b) Comparison between Task 3 and Task 4 in log scale.

Table 1. Comparison table of different tasks in seconds.

Programming Tasks	Size of the input grid side		
	10	100	1000
Task 1	0.002	0.005	0.137
Task 2	0.001	0.004	0.137
Task 3	0.002	8.408	8000 (est.)
Task 4	0.002	0.007	1.884
Task 5 (Bonus)	4.48	4.467	5.213

Table 1 and Fig. 1 shows a comparison of these tasks in terms of their running time. In case of task 3 (brute-force), the running time for 1000×1000 input matrix didn't finish after 2 hours, therefore an estimated value of 8000 is used. Due to the Task 3 taking exceptionally longer time than the other tasks (which is expected due to 6 nested for loops of the brute force algorithm), Fig 1(b), is plotted in log scale make comparison with the Task 4.

Interesting, observations can be made for the plots shown in Fig. 1. In Fig. 1(a), the algorithm with tighter space constraint runs faster for smaller sized matrix. Once the size of the matrix grows

bigger, the running time for task 1 and task 2 merges to similar points. Alongside, the difference in running time for these tasks in Fig. 1(a) is insignificant.

In Fig. 1(b), the brute-force algorithm (Task 3) runs multiple order of complexity slower than the $O(mn)$ complexity algorithm of Task 4. In this case Task 4 is the clear winner with only 2 nested for loops though consuming a bit more extra space.

Bonus Credit: In Table 1 and Table 2, comparisons are made for that of the bonus credit (Task 5) portion too. Due to running nested Kadane's algorithm [1], Task 5 takes more time than expected though not out of rangle.

Table 2. Comparison table of different task in terms their technicalities, implementation, performance, etc.

Programming Tasks	**Implementation**	**Performance**	**Extra Space**
Task 1	Moderate	Good	$O(mn)$
Task 2	Moderate	Slightly better	$O(n)$
Task 3	Easiest	Worst	$O(1)$
Task 4	Hard	Great	$O(mn)$
Task 5 (Bonus)	Hardest	Good	$O(m)$

REFERENCES

[1] R. S. Bird, "Algebraic identities for program calculation," *The Computer Journal*, vol. 32, no. 2, pp. 122–126, 1989.