ALGORITHMS PROGRAMMING PROJECT

COT5405

Analysis of Algorithms

Problem - Finding the largest possible area (shaped square or rectangle) of blocks within city (grid layout of $m \times n$ cells) limits that allows a building of height at least h.

Submitted to: DR. ALPER ÜNGÖR

Submitted by:

Name: Parth Gupta

UFID: 9199-7064

Email: parthgupta@ufl.edu

Problem Description and Implementation

Consider a city in Florida named Gridville that has a grid layout of m × 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 find the largest possible area (shaped square or rectangle) of blocks within city limits that allows a building of height at least *h*.

I have implemented the program that will return the upper left and bottom right coordinates of a square or a rectangle based on different tasks with the largest possible area. The project has been implemented in C++. The input file is supplied as a command line argument while running the program, then the output will be shown on the command prompt itself.

Firstly, my program will convert the given input for each value cell(i,j) to 0 and 1 according to the *h* value. If a particular value cell(i,j) is greater than or equal to *h* than that value will be equal to 1 or else it will be 0. I have done this since we have to return largest area such that the largest area consists of all the cells that have height of at least h. For Example:

if(cell[i,j] >= h) cell[i,j] = 1

else cell[i,j] = 0

5	4	9	1	3
10	2	**5**	**6**	**7**
8	11	**78**	**7**	**11**
9	1	4	15	3

h = 5 →

1	0	1	0	0
1	0	**1**	**1**	**1**
1	1	**1**	**1**	**1**
1	0	0	1	0

Structure of the Program and Function Description

Variables in main function:

1) Rows: Integer, represented as m.
2) Columns: Integer, represented as n.
3) Height: Integer, represented as h.
4) 2-D Vector: Integer, represented as vec.

Functions:

1) Int main(int argc, char* argv[]) // main function, where the program starts and conversion to 0/1 take place here.
2) Int sum(vector<vector<int>> vec, int left, int right, int left1, int right1) // returns the sum of a rectangle from (left,right) to (left1,right1).
3) Int task1(int m, int n, int vector<vector<int>> vec) // implementation of task 1
4) Int task2(int m, int n, int vector<vector<int>> vec) // implementation of task 2
5) Int task3(int m, int n, int vector<vector<int>> vec) // implementation of task 3 and also calls a function sum().
6) Int task4(int m, int n, int vector<vector<int>> vec) // implementation of task 4
7) Int task5(int m, int n, int vector<vector<int>> vec) // implementation of task 5

Command Line Arguments

1) g++ algorithm.cpp -o AlgoTowers // this will create an executable named AlgoTowers and this would be implemented via make file.
2) AlgoTowers 2 InputFile // this will implement/run task 2 and read the input from the input file and then print the output on the command prompt itself.

1) Algorithm 1

Design a *Θ(mn)* time **Dynamic Programming** algorithm for computing a largest area **Square** block with all cells have the height permit value at least *h*.

Task 1

Give a recursive implementation of Algorithm 1 using **Memoization** and *O(mn)* extra space.

Could Not Implement This Task, Rest All The Tasks Are Fully Functional!!

Task 2

Give an iterative **BottomUp** implementation of Algorithm 1 using *O(n)* extra space.

Implementation

a) In order to perform this task, I have taken one 1-dimensional vector of size n. The idea behind this is to construct the vector in which each entry represents the size of sub-square with all 1's.

b) We are filling the vector such that if a value at particular index in the original input is 1 then we write 1 in the solution vector otherwise we write Min(left,diagonal,top) + 1. Since we are doing this on O(n) space therefore we have to store diagonal in a temporary variable every time.

c) The process continued and maxarea is stored on basis of which the upper left and bottom right coordinates are printed.

d) Mathematical Formulation: Let Vec be the input vector. 0<=i<m, 0<=j<n

- Solution vector base case

$$Sol[j] = \begin{cases} 1, \text{ if } Vec[i][j] = 1 \\ 0, \text{ otherwise} \end{cases}$$

- Solution vector

$$Sol[j] = \begin{cases} 1 + Min(Sol[j],Sol[j-1],diagonal), \text{ if } Vec[i][j] = 1 \\ 0, \text{ otherwise} \end{cases}$$

Time Analysis

a) Let number of rows be denoted by *m* and number of columns by *n*.

b) Time taken to convert the original input to 0 and 1 is *Θ(mn)* time since we are using 2 for loops that will iterate through the input. This thing is common in every task

and has been done in main() function once and then the input is being passed as reference.

c) There are two nested for loops that are being iterating through the input and creating a solution vector accordingly. Since each row and column is being evaluated therefore the time take by algorithm is $\Theta(mn)$.

d) Therefore, the total time taken by algorithm is $\Theta(mn + mn) = \Theta(2mn) = \Theta(mn)$.

Space Analysis

a) The space is of $O(n)$ complexity since I am using 1 additional 1-dimensional vectors of size n for storing the size of each sub-square.

2) Algorithm 2

Design a $\Theta(m^3n^3)$ time **Brute Force** algorithm for computing a largest area **Rectangle** block with all cells have the height permit value at least h.

Task 3

Give an implementation of Algorithm 2 using $O(1)$ extra space.

Implementation

a) In this approach I am returning the coordinates rectangle with maximum sum containing all 1's which would be equivalent to the h value constraint. **Note, I am calculating sum because when each cell value is 1 then the sum and area are always equal**. So, the rectangle with maximum area is (1,2) and (2,4) with area of 6 units in the above diagram. Also, this will work for every case since each cell value is equal either to 1 or 0.

b) The Brute Force Approach is something that try out all possible solutions and return the best one. In this task I am evaluating sum of every possible rectangle starting from top left corner. After each rectangle being evaluated from the starting point, we than move the starting corner to the right by 1. After all the rectangles in the first row are being evaluated then we shift towards the left again and start from the second row. This process continues until we reach the bottom right corner.

c) In the end I am returning the upper left and bottom right coordinates of the rectangle with maximum sum such that all the elements/cell value in the rectangle are 1.

Time Analysis

e) Let number of rows be denoted by m and number of columns by n.

f) Time taken to convert the original input to 0 and 1 is $\Theta(mn)$ time since we are using 2 for loops that will iterate through the input.

g) The algorithm make uses of 6 for loops, which takes $\Theta(m^3n^3)$ that is $\Theta(m^2n^2)$ to choose all pairs of rectangles [(left,right) to (left1,right1)] which uses 4 for loops and check if its filled with 1's and $\Theta(mn)$ for computing the sum which uses 2 for loops.

h) Therefore, the total time taken by algorithm is $\Theta(mn + m^3n^3) = \Theta(m^3n^3)$.

Space Analysis

a) The space is of $O(1)$ complexity. The amount of auxiliary space is not dependent upon the size of the input data structure.

b) At any point in the iteration, only four variables are used to store the top left and bottom right coordinates of current optimal solution. Temporary variables are used in sum() function which gets de-scoped proceeding further, all of which are used irrespective of the size of input supplied.

3) Algorithm 3

Design a $\Theta(mn)$ time **Dynamic Programming** algorithm for computing a largest area **Rectangle** block with all cells have the height permit value at least h.

Task 4

Give an iterative **BottomUp** implementation of Algorithm 3 using $O(mn)$ extra space.

Implementation

a) This task is also being implemented using dynamic programming approach. In order to perform this, I have used three 2 dimensional vectors each of size (rows + 1)*(columns) that is (m+1)*(n). These vectors are used to store height (height of rectangle that can be formed on each cell), right boundary, and left boundary of a rectangle. Then we will iterate through every element and use this tables/vectors to compute max area of the rectangle.

b) Each of the three vectors will have an extra row as a padding because we are computing the height, right boundary, and left boundary on the basis of information in the previous row. It means first row in the input will correspond to the second row in each table.

c) Finally, on the basis of max area and with the help of 3 tables/vectors I am computing upper left and bottom right coordinates of the optimal rectangle.

d) Mathematical Formulation: Let Vec be the input vector. 1<=i<m+1, 0<=j<n

- For height table

$$\text{Height}[i][j] = \begin{cases} \text{Height}[i-1][j] + 1, & \text{if Vec}[i-1][j] = 1 \\ 0, & \text{otherwise} \end{cases}$$

- For left boundary table, leftbound = 0 initially

$$\text{Left}[i][j] = \begin{cases} \max(\text{leftbound}, \text{Left}[i-1][j]), & \text{if Vec}[i-1][j] = 1 \\ 0 \text{ and leftbound} = j+1, & \text{otherwise} \end{cases}$$

- For right boundary table, rightbound = n-1 initially

$$\text{Right}[i][j] = \begin{cases} \min(\text{rightbound}, \text{Right}[i-1][j]), & \text{if Vec}[i-1][j] = 1 \\ 0 \text{ and rightbound} = j-1, & \text{otherwise} \end{cases}$$

Time Analysis

a) Time taken to convert the original input to 0 and 1 is $\Theta(mn)$ time since we are using 2 for loops that will iterate through the input.

b) Time to build height, right boundary, and left boundary table and iterating through the input is $\Theta(m*(n+n)) = \Theta(2mn) = \Theta(mn)$. This is because I have used 3 for loops. First for loop is the outer one that will iterate through each row then next 2 for loops are used to create height, right and left table and iterate through each column. These 2 for loops are nested in the outer for loop but not nested to each other. In the end 2 more separate for loops are used to calculate max area and coordinates of the rectangle.

c) Therefore, the total time taken by algorithm is $\Theta(mn + mn + mn) = \Theta(3mn) = \Theta(mn)$.

Space Analysis

a) The space is of $O(3(m+1)n) = O(mn)$ complexity since I am using 3 additional 2-dimensional vectors for creating the height, right and left tables each of size (m+1)*n.

Task 5 [Bonus]

Give an iterative **BottomUp** implementation of Algorithm 3 using *O(n)* extra space.

Implementation

a) This task is also being implemented using dynamic programming approach. This is similar to the task 4 explained above, the only difference is that we have to implement this task in *O(n)* space. This can be achieved since while computing the particular row, we need only the previous row so instead of keeping the record of complete 2-dimensional vector we can use a 1-dimensional vector to store the previous row.

b) In order to perform this, I have used three 1 dimensional vectors each of size (columns) that is (n). These vectors are used to store height (height of rectangle that can be formed on each cell), right boundary, and left boundary of a rectangle. Then we will iterate through every element and use the vector to compute max area of the rectangle.

c) Finally, on the basis of max area and with the help of 3 vectors I am computing upper left and bottom right coordinates of the optimal rectangle.

d) Mathematical Formulation: Let Vec be the input vector. 0<=i<m, 0<=j<n

- For height table

$$\text{Height}[j] = \begin{cases} \text{Height}[j] + 1, \text{ if Vec}[i][j] = 1 \\ 0, \text{ otherwise} \end{cases}$$

- For left boundary table, leftbound = 0 initially

$$\text{Left}[j] = \begin{cases} \text{Max(leftbound,Left}[j]), \text{ if Vec}[i][j] = 1 \\ 0 \text{ and leftbound} = j+1, \text{ otherwise} \end{cases}$$

- For right boundary table, rightbound = n initially

$$\text{Right}[j] = \begin{cases} \text{Min(rightbound,Right}[j]), \text{ if Vec}[i][j] = 1 \\ 0 \text{ and rightbound} = j, \text{ otherwise} \end{cases}$$

Time Analysis

a) Time taken to convert the original input to 0 and 1 is $\Theta(mn)$ time since we are using 2 for loops that will iterate through the input.

b) Time to build height, right boundary, and left boundary table and iterating through the input is $\Theta(m*(n+n)) = \Theta(2mn) = \Theta(mn)$. This is because I have used 3 for

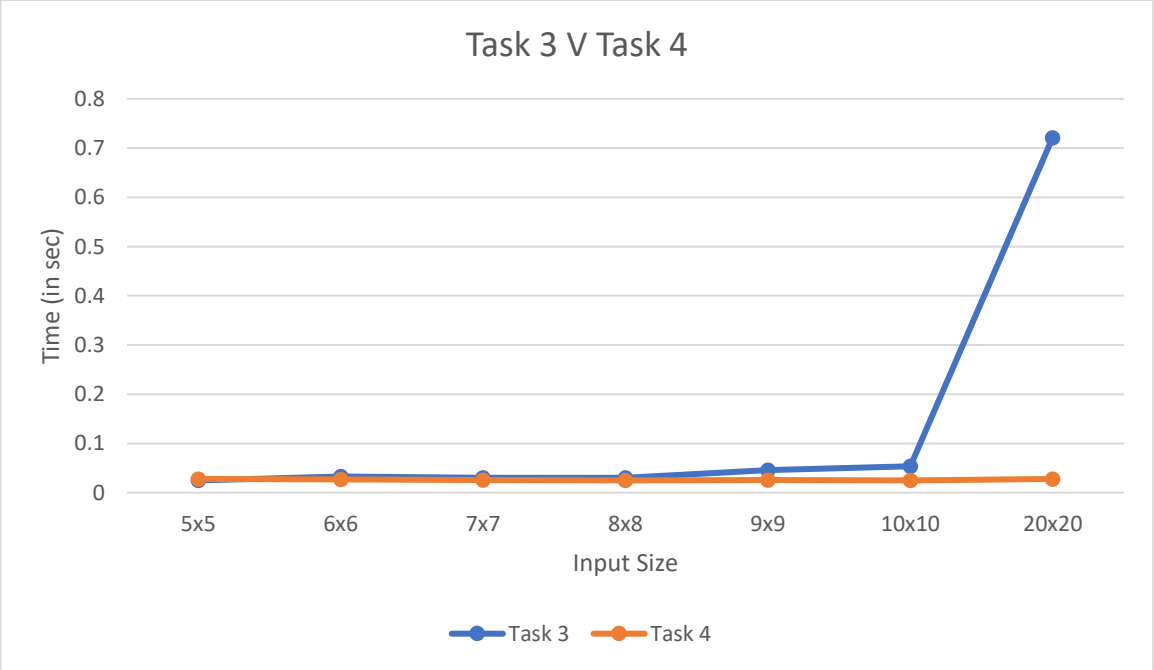
loops. First for loop is the outer one that will iterate through each row then next 2 for loops are used to create height, right and left table and iterate through each column. These 2 for loops are nested in the outer for loop but not nested to each other.

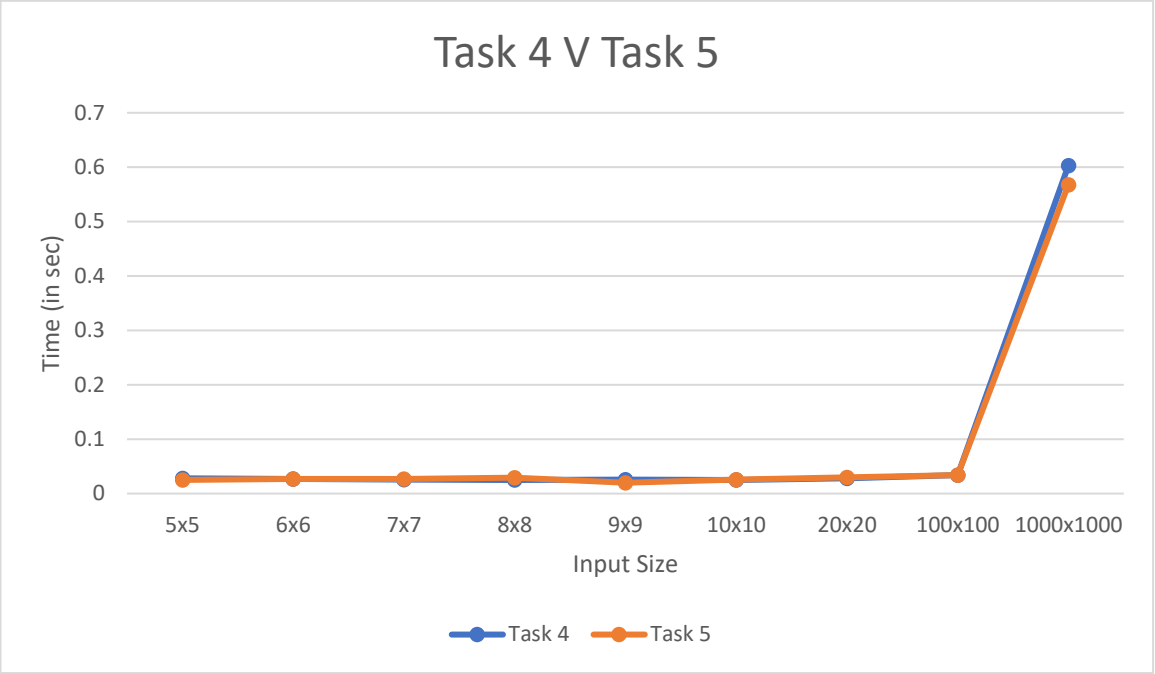
c) Therefore, the total time taken by algorithm is $\Theta(mn + mn + mn) = \Theta(3mn) = \Theta(mn)$.

Space Analysis

b) The space is of $O(3n) = O(n)$ complexity since I am using 3 additional 1-dimensional vectors for storing the height, right and left rows each of size n.

Experimental Comparative Study




Task 4 V Task 5
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Time (in sec)
5x5
6x6
7x7
8x8
9x9
10x10
20x20
100x100
1000x1000
Input Size
Task 4
Task 5