Analysis of Algorithms (COT 5405)
Fall 2020
Programming Project Report

Name Mingjun Yu
UFID 61707843

1. Problem Definition

Given a 2D array $(m \times n)$ filled with non-negative integers and a positive integer h, find the largest square or rectangle containing only integers that are larger than or equal to h. Return two coordinates (x_1, y_1) and (x_2, y_2) to indicate the location of the largest square or rectangle, where (x_1, y_1) is the upper left corner and (x_2, y_2) is the lower right corner of the optimal solution region.

2. Algorithm Design and Implementation

2.1 Algorithm1

Algorithm1 is a $\Theta(mn)$ time dynamic programming algorithm for computing a largest area square block with all cells have the height permit value at least h.

If the current cell is $matrix[0][0]$, we can find that the cells $matrix[0][1]$, $matrix[1][0]$, and $matrix[1][1]$ are each the upper left-hand corner of their own respective 3×3 subarrays. With this knowledge, current cell $matrix[0][0]$ is the only cell missing in a subarray of the next size larger. So, if $matrix[i][j] \geq h$, then it is the upper left-hand corner of the minimum size of the three subarrays to the bottom, right, and bottom-right. In this way,

Recursive formulation Assume that $dp(i, j)$ represents the side length of the maximum square whose upper left corner is the cell with index (i, j) in the original matrix. Starting from index $(0,0)$, for every $matrix[i][j] \geq h$ found in the original matrix, the value of the current element will be updated as

$$dp(i, j) = \min_{0 \leq i < m, 0 \leq j < n} \big(dp(i+1, j), dp(i, j+1), dp(i+1, j+1)\big) + 1$$

Optimal substructure The solutions to the subproblems used within an optimal solution to the problem must themselves be optimal. This can be argued by using "cut-and-paste" technique. Suppose that $dp(i, j) = dp(i+1, j) + 1$, where $dp(i, j)$ is the optimal solution but $dp(i+1, j)$ is not, which means there exits a $dp(i+1, j)' > dp(i+1, j)$. Then we can cut out the nonoptimal solution $dp(i+1, j)$ and past in the optimal one, i.e. $dp(i+1, j)'$. In this way we can get a better solution to the original problem, thus contradicting previous supposition that $dp(i, j)$ is already an optimal solution. So, this problem must have the optimal substructure.

Complexity analysis In this algorithm, we need to traverse each entry in the matrix once to compute $dp(i, j)$, and the size of matrix dp is also $m \times n$. Thus, the time complexity is

$\Theta(mn)$, and the space complexity is $O(mn)$.

But it can be noticed that for computing each $dp(i,j)$, only adjacent three data have been used: $dp(i,j+1)$, $dp(i+1,j)$ and $dp(i+1,j+1)$. Therefore, 2D dp matrix is not necessary as 1D dp array will be sufficient for this. In this way, the space complexity is able to be reduced to $O(n)$.

2.1.1 Task1

Task1 is about to give a recursive implementation of Algorithm1 using memorization and $O(mn)$ extra space.

Combining with the boundary condition, $dp(i,j)$ can be computed as follow.

$$dp(i,j) = \begin{cases} 0, if\ matrix[i][j] < h \\ \min\limits_{0 \le i < m, 0 \le j < n} \big(dp(i+1,j), dp(i,j+1), dp(i+1,j+1)\big) + 1, if\ matrix[i][j] \ge h \\ 0, if\ i = m\ or\ j = n \end{cases}$$

Recursive implementation means Algorithm1 needs to be implemented in the top-down version. In this implementation, starting from $(0,0)$, $dp(i,j)$ has been computed as above recursive formulation suggested.

Ease of implementation It is not hard to implement.

Technicalities I use overload in Java to give a recursive implementation.

Performance Recursive implementation costs more time than iterative implementation.

2.1.2 Task2

Task2 is about to give an iterative bottom-up implementation of Algorithm1 using $O(n)$ extra space.

While the idea is the same, the subproblem defined in the top-down implementation makes iteration confusing. Because recursion recurses to the end before working its way up, it traverses the array backwards. It starts solving subproblems with the bottom right corner of the array. So, it is necessary to slightly modify the definition of subproblems: instead of representing the side length of the maximum square whose upper left corner is the cell with index (i,j) in the original matrix, $dp(i,j)$ will represent the bottom right corner of the maximum square.

In order to satisfy the condition that using $O(n)$ extra space, I initialize a 1D array $dp[n+1]$. The initial value of each entry in the array dp is 0. Starting from $(1,1)$, the recursive

formulation will be defined as follow. In the formulation, $prev$ refers to the old $dp[j-1]$.

$$dp(i,j) = \begin{cases} 0, if\ matrix[i][j] < h \\ \min\limits_{1\le i\le m, 1\le j\le n}(dp[j-1], dp[j], prev), if\ matrix[i][j] \ge h \end{cases}$$

Ease of implementation Once the definition of subproblems has been modified, it is not hard to implement.

Technicalities Since not all the values of dp have been used in the recursive formulation, 1D array would be sufficient.

Performance The performance of Task2 is better than Task1.

2.2 Algorithm2

Algorithm2 is a $\Theta(m^3n^3)$ time brute force algorithm for computing a largest area rectangle block with all cells have the height permit at least h.

Algorithm2 is intuitive. We can enumerate every possible rectangle in the matrix, and then verify if every cell in the current rectangle satisfies $matrix[i][j] \ge h$. This is able to be accomplished by traverse all the coordinates(x_1, y_1) and (x_2, y_2), where (x_1, y_1) represents the upper left corner of the rectangle and (x_2, y_2) represents the bottom right corner of the rectangle. The time complexity of traversing all the diagonal vertices is $\theta(m^2n^2)$, and the time complexity to verify if the current rectangle satisfies the condition is $O(mn)$. So, the time complexity of this brute force algorithm is $\theta(m^3n^3)$.

2.2.1 Task3

Task3 is about to give an implementation of Algorithm2 using $O(1)$ extra space. Since nothing needs to be stored, the space complexity is $O(1)$.

Ease of implementation Brute force algorithm is intuitive, so it is easy to implement.

Technicalities Same as Algorithm2 suggested.

Performance Brute force does not store useful information during traversal. This implementation is very poor.

2.3 Algorithm3

Algorithm3 is a $\Theta(mn)$ time dynamic programming algorithm for computing a largest area rectangle block with all cells have the height at least h.

This problem has overlapping subproblems, and the brute force algorithm will recompute subproblems that have already been computed. If we store some information during the traversal, recomputing can be avoided.

In order to design a $\Theta(mn)$ time dynamic programming algorithm, we can first come up with a $O(m^2n)$ dynamic programming algorithm Algorithm2.9. The size of a rectangle is decided by its width and height, so we can fix one of them and use dynamic programming to compute the maximal value of another. In this case, it would be easier to fix the height. Then traverse each row to calculate the maximal width of each cell if we view the current cell as the bottom right corner of a rectangle. Using $dp(i,j)$ to represent the width of the possible maximum rectangle whose bottom right corner is the cell with index (i,j) in the original matrix, then $dp(i,j)$ can be calculated as follow.

$$dp(i,j) = \begin{cases} 0, if\ j = 0 \\ dp(i-1,j) + 1, if\ matrix[i][j] = 1 \\ 0, if\ matrix[i][j] = 0 \end{cases}$$

Then the maximal area of the rectangle can be calculated as following pseudo codes.

```
Compute the maximum area rectangle with the lower right at (i,j)
maxWidth = dp (originalRow, j)
for(int r = originalRow; r ⩾ 0; r--) {
    maxWidth = min (maxWidth, dp (r, j))
    currentArea = maxWidth × (r – originalRow + 1)
    maxArea = max (maxArea, currentArea)
}
```

Time complexity of the above algorithm is $O(m)$. Other than that, we need to traverse each cell to compute $dp(i,j)$, the time complexity of this part is $O(mn)$. Thus, the time complexity of the entire algorithm is $O(m^2n)$.

If we can find out the maximum width of a rectangle with its height fixed in one iteration, we can improve this $O(m^2n)$ time algorithm, and design a $\Theta(mn)$ time algorithm. This is where Monotone Stack comes into play. The stack saves indices which gives the highest so far. For example, if histogram heights are 1, 2, and 3, the stack saves all three indices of 0, 1, 2. Once it hits a lower height, a rectangle area will be calculated at that index. For calculation, it needs the first no greater height seen so far compared to the current height. The information is saved in the stack. Since the stack saves indices of highest so far, popping out indices until the corresponding height is higher gives the targe index. The target index is the starting index of a sub-histogram. With starting and current indices, the area will be calculated by multiplying width (difference of indices) and height. The area is the result of the subproblem.

So, in Algorithm3, to fix the height of rectangles, the first step of dynamic programming sees vertically. In order to calculated the maximal width, the second step of dynamic programming

sees horizontally.

Recursive formulation Assume that $dp(i,j)$ represents the height of the maximum rectangle whose bottom is at the cell with index (i,j) in the original matrix. Then the recursive formulation is shown as follow.

$$dp(i,j) = \begin{cases} 1, if\ i = 0\ and\ matrix[0][j] \geq h \\ dp(i-1,j) + 1, if\ i \geq 1\ and\ matrix[i][j] \geq h \\ 0, if\ matrix[i][j] < h \end{cases}$$

Optimal substructure Suppose that $dp(i,j) = dp(i-1,j) + 1$, where $dp(i,j)$ is an optimal solution but $dp(i-1,j)$ is not, which means there exists a $dp(i-1,j)' > dp(i-1,j)$. Then we can cut out the nonoptimal solution $dp(i-1,j)$ and past in the optimal one, i.e. $dp(i-1,j)'$. In this way we can get a better solution to the original problem, thus contradicting previous supposition that $dp(i,j)$ is already an optimal solution. So, this problem must have the optimal substructure.

Complexity analysis In this algorithm, we need to traverse each entry in the matrix once to compute $dp(i,j)$, and the size of matrix dp is also $m \times n$. Thus, the time complexity is $\Theta(mn)$, and the space complexity is $O(mn)$.
But it can be noticed that for computing each $dp(i,j)$, only $dp(i-1,j)$ is needed. Therefore, 2D dp matrix is not necessary as 1D dp array will be sufficient for this. In this way, the space complexity is able to be reduced to $O(n)$.

2.3.1 Task4

Task4 is about to give an iterative bottom up implementation of Algorithm3 using $O(mn)$ extra space. The recursive formulation is illustrated as above.

Ease of implementation Finding a maximum rectangle is harder than finding a maximum square.

Technicalities Same as Algorithm3 suggested. I use stack to implement Monotone Stack.

Performance Although the time complexity of Task4 and Task5 is both $\Theta(mn)$, Task5 performs better than Task4 in practice.

2.3.2 Task5

Task5 is about to give an iterative bottom up implementation of Algorithm3 using $O(n)$ extra space. The recursive formulation is a little different. In the following formulation, the array dp on the right side of the formulation stores values of previous row. Initially the dp array contains all 0's.

$$dp(j) = \begin{cases} 0, if\ matrix[i][j] < h \\ dp(j) + 1, if\ matrix[i][j] \geq h \end{cases}$$

Ease of implementation It is not hard to improve Task4 and give an implementation using $O(n)$ extra space.

Technicalities Since not all the values of dp have been used in the recursive formulation, 1D array would be sufficient.

Performance Although the time complexities of Task4 and Task5 are both $\Theta(mn)$, Task5 performs better than Task4 in practice.

2.3.3 Another Implementation of Task5

There is another way to implement Task5, and I will refer to it as **Task6**. The basic idea is same as Algorithm3: firstly, fix height; then calculate the maximum width corresponding to current height. The only difference is how to calculate the maximum width. In each height-fixed row i, using $left(j)$ to represent the left most column of current rectangle, and the cell $matrix[i][j]$ lies in the bottom column of this rectangle. Similarly, $right(j)$ represents the right most column of current rectangle. Initially, array $left$ is filled with 0 and array $right$ is filled with n. We need to auxiliary parameters: $curLeft$ and $curRight$. Initially, $curLeft = 0$, $curRight = n$. Starting from $(i, 0)$, $left(j)$ is updated as follow. $maxWidth = right(j) - left(j)$.

$$left(j) = \begin{cases} 0, curLeft = j + 1, if\ matrix[i][j] < h \\ \max(left(j), curLeft), if\ matrix[i][j] \geq h \end{cases}$$

Starting from (i, n), $right(j)$ is updated as follow.

$$right(j) = \begin{cases} 0, curRight = j, if\ matrix[i][j] < h \\ \min(right(j), curRight), if\ matrix[i][j] \geq h \end{cases}$$

The correctness of this optimal substructure can also be proved by “cut-and-paste” technique.

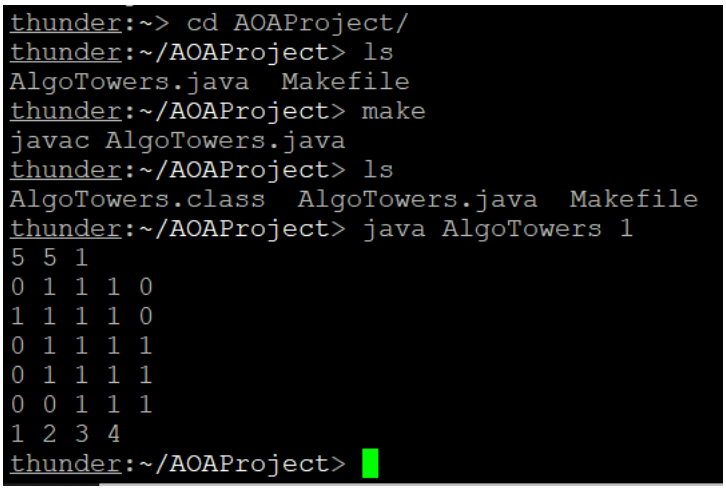
Ease of implementation It is not hard to implement once figuring out the recursive formulation of $left(j)$ and $right(j)$.

Technicalities Only store the data that have been used in recursive formulation, 1D array would be sufficient.

Performance Although the time complexities of Task4, Task5 and Task6 are all the same, Task6 performs better than Task4 and Task5 in practice.

3. Instructions for Execution

There are three files in the compressed file folder: the PDF version of this report, Makefile and AlgoTowers.java. The code has been tasted on the virtual Linux machine provided by CISE department. Below is the result of a simple test.



The program should be run as follows:

1. Use following command at terminal to build the code. This command will generate a binary file AlgoTowers.class.

    $ make

2. Run AlgoTowers using following command. The number of task integer corresponding to Task1 ~ Task6.

    $ java AlgoTowers <number_of_task>

3. Input integers m, n, h separated by one space character.

4. For the next m lines, line $i+1$ consist of n integers $p[i,1]$, $p[i,2]$, ⋯, $p[i,n]$ in this particular order separated by one space character.

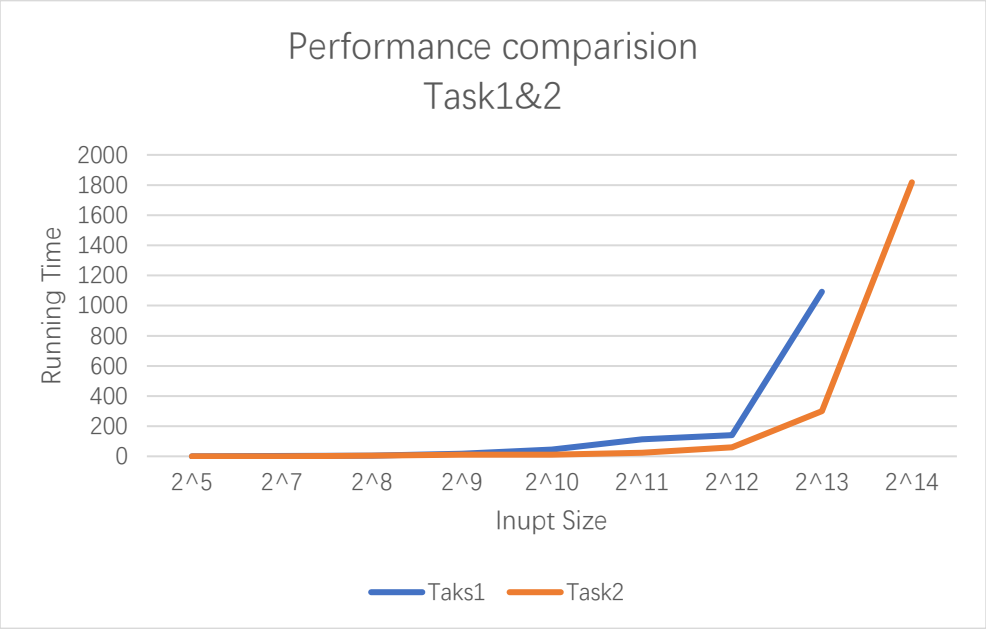
After input, the program will print four integers x_1, y_1, x_2, y_2 separated by space characters, where (x_1, y_1) is the upper left corner and (x_2, y_2) is the lower right corner of the optimal solution region.

4. Performance Comparison

In order to measure the performance of different tasks, I created randomly generated input files of various sizes. Due to the limitation of Java heap space, the maximum size of input file I can generate is $2^{14} \times 2^{14}$. I think if Java heap size could be larger, then more data are able to be processed by these implementations. Other than Task3, other implementations are able to output correct results when the size of input file is larger than $2^{10} \times 2^{10}$. In the following

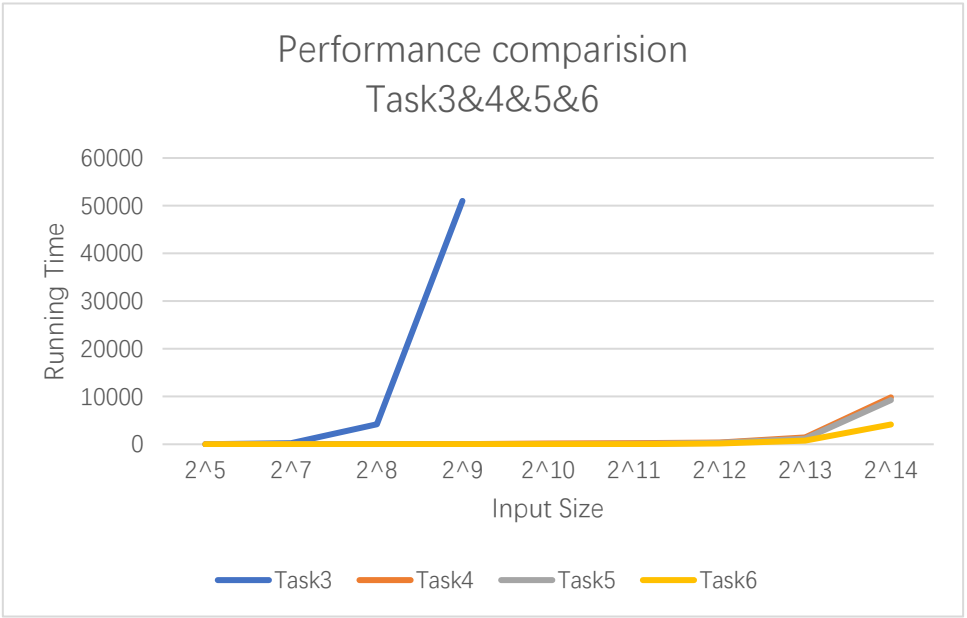
chart, the input size represents the length of matrix's rows (For simplicity, the length of matrix's rows equals to the length of matrix's columns).

Performance comparison between Task1 and Task2:



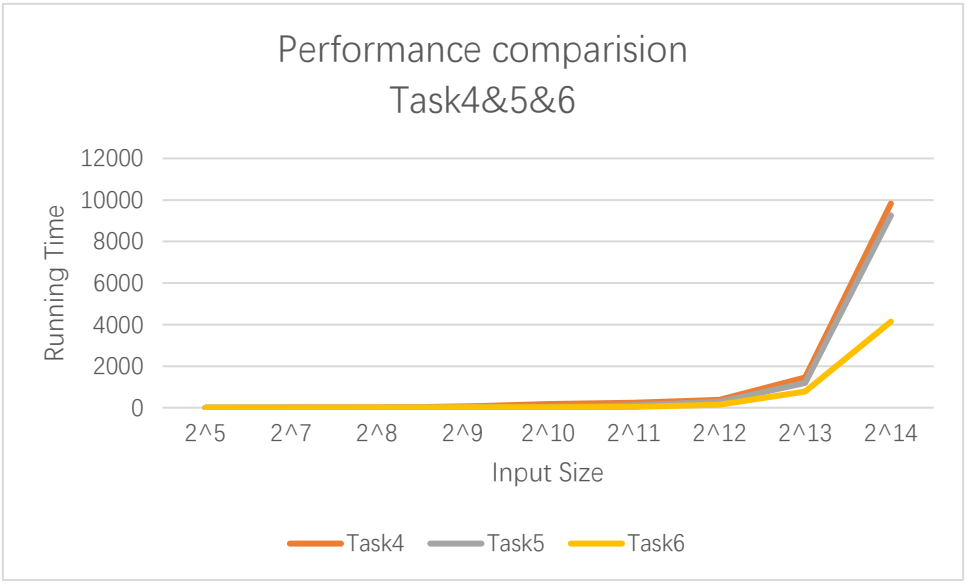
From the chart, we can see that Task2 performs better than Task1. The reason is that calling recursive function will cost more time. Also, due to the limitation of Java stack size, Task1 is not able to handle data size larger than 2^{14}, while Task2 is able to output results.

Performance comparison between Task3, Task4, Task5 and Task6:



From above chart we can see that the implementation of brute force algorithm (Task3) failed to compute results when input size is as large as 2^{10}. In order to tell the difference between Task4, Task5 and Task6, I drew another chart.

Performance comparison of Task4, Task5 and Task6:



From this chart we can tell that although their time complexities are all the same, Task6 performs better than Task5 and Task5 performs better than Task4. The extra space constraint does have impact on performance.