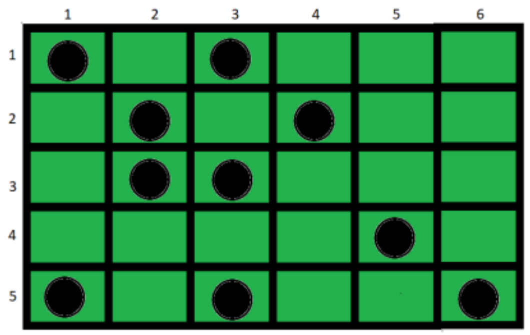
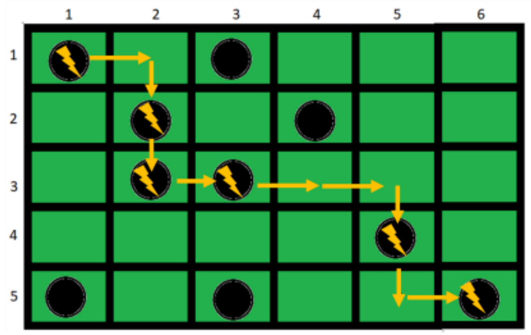
Agricultural robotics problem Consider a farm of a rectangular shape, that is divided into n×m square blocks. Some of these blocks contain weeds (i.e., unwanted plants that grow in-between crops). For instance, the left figure below illustrates a farm of size 5 × 6, where the blocks that contain weeds are denoted by black circles.





Consider an agricultural mobile robot that is initially located at block (1, 1).
The robot can navigate one block at a time, either to the right or to down. When the robot is at a block, it removes all the weeds in it. Given the information about which blocks contain weeds, the goal is for the robot to remove weeds from as many blocks as possible, and reach the charging area located at block (n, m). For instance, for the farm shown above, the robot can remove weeds from 6 blocks as illustrated in the right figure. Describe a dynamic programming algorithm to find a path that the robot can follow to remove weeds from the maximum number of blocks.

(a) Recursive formulation of this agricultural robotics problem.

Let's say the matrix stores 1 or 0's according to whether the cell contains weed or not.
0: "does not contain weed"
1: "contains weed"

Then the above matrix should look like this:

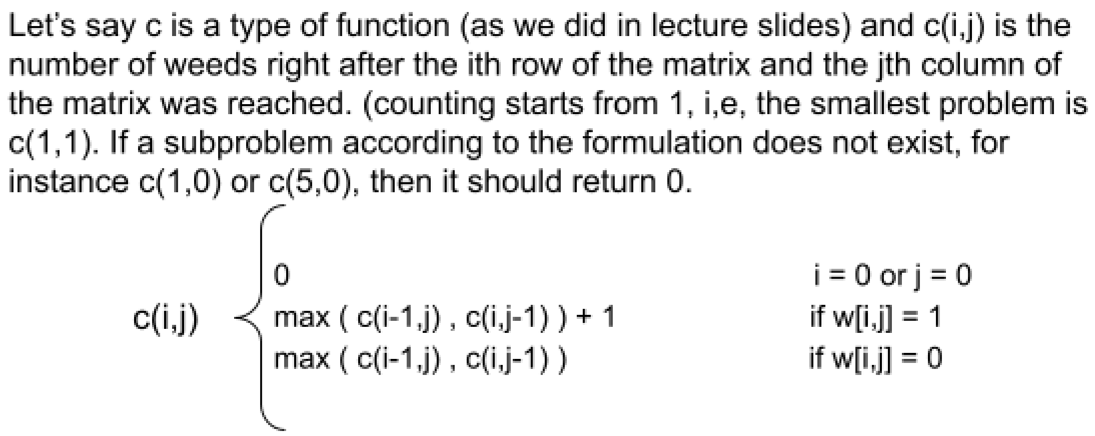
1	0	1	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	0	0	0	1	0
1	0	1	0	0	1

I will say that the given matrix (the green matrix) is the matrix W which stores the information of containing weed or not.

The main problem is to “find a path that the robot can follow to remove weeds from the maximum number of blocks. “
So, the function to find that path has the name “PWMW”: Path With Maximum Weeds”

To do that, first I should find the number of weeds so far right after that block is reached with the path containing the max number of weeds: c(i,j)
Secondly, I should extend the algorithm to find the PWMW itself.

The recursive formulation is:

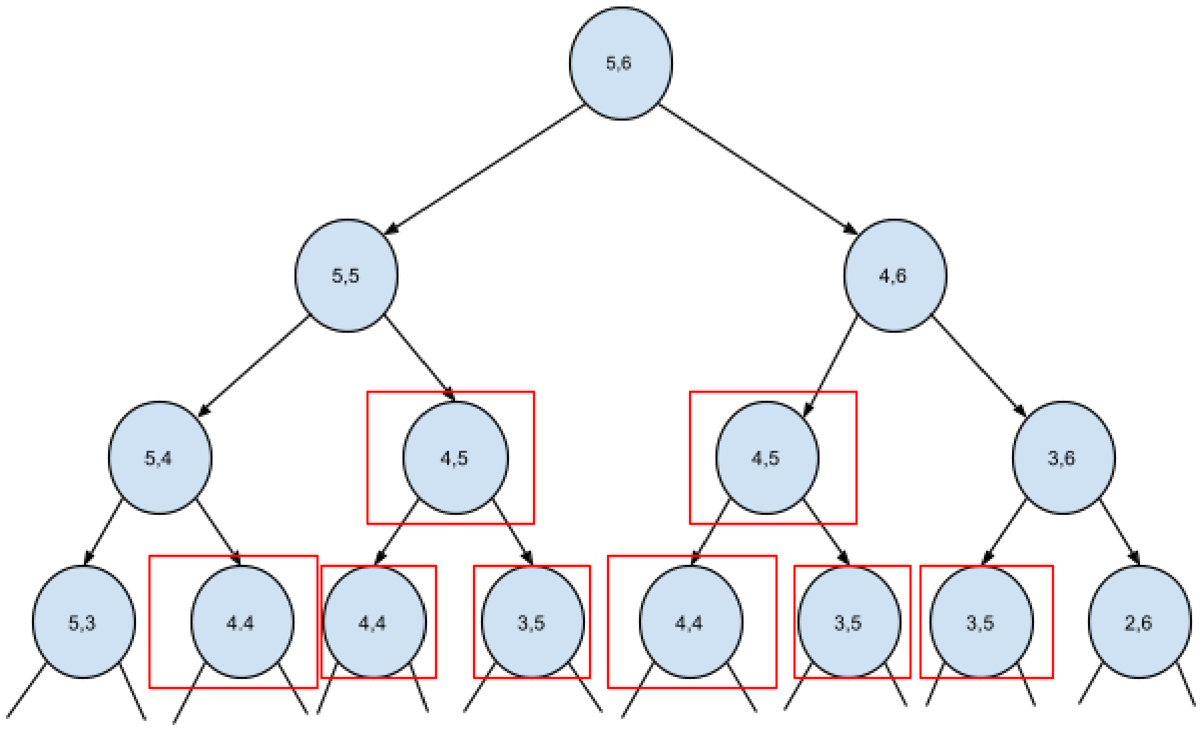


Let’s say c is a type of function (as we did in lecture slides) and c(i,j) is the number of weeds right after the ith row of the matrix and the jth column of the matrix was reached. (counting starts from 1, i,e, the smallest problem is c(1,1). If a subproblem according to the formulation does not exist, for instance c(1,0) or c(5,0), then it should return 0.

$$c(i,j) \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ \max(c(i-1,j), c(i,j-1)) + 1 & \text{if } w[i,j] = 1 \\ \max(c(i-1,j), c(i,j-1)) & \text{if } w[i,j] = 0 \end{cases}$$

(b) Pseudocode of your algorithm designed using dynamic programming based on the recursive formulation.
As one can see in the recursive formulation, the problem has the optimal substructure property. Since the solution to c(i,j) comes from solving c(i,j-1) or c(i-1,j).
Also, the problem has the overlapping subproblems property. I will show it by drawing a recursion tree.

□ : Overlapping subproblems

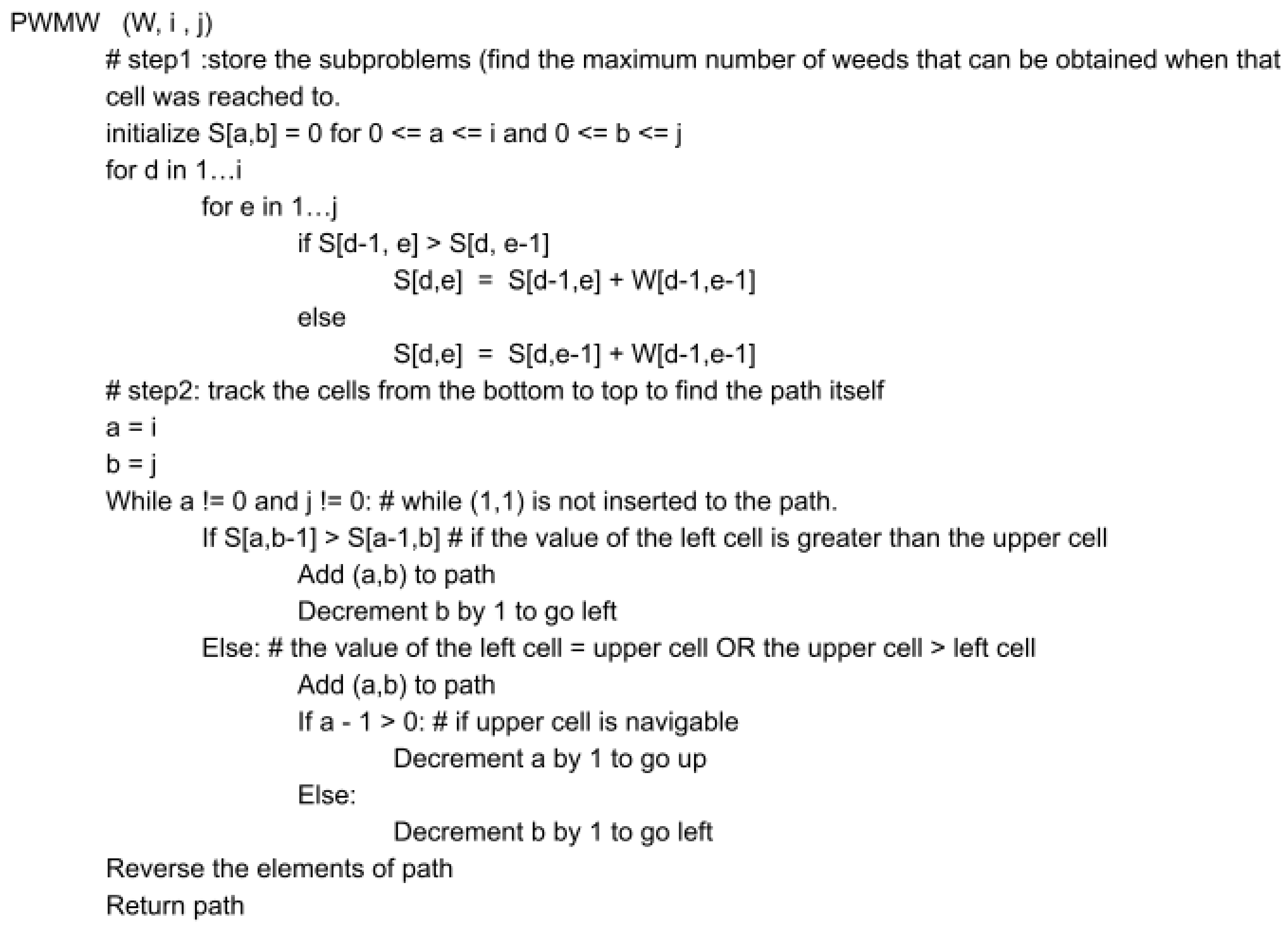


Here, I will use the bottom up approach as my dp algorithm. That's why, I will store the solutions to subproblems into a matrix S.
This matrix will have a size of (W.numberOfRows + 1) x (W.numberOfColoumns + 1).
This is for solving problems that don't exist like c(1,0), c(5.0).
For example, the green matrix is a 5x6 matrix. A 6x7 sized matrix is required to store the solutions to its subproblems.

Matrix S which is 6 x7 to solve the green matrix above, as an example:

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

The PWMW algorithm takes W , i , j as the inputs, the matrix to be solved, its number of rows and number of columns, respectively.

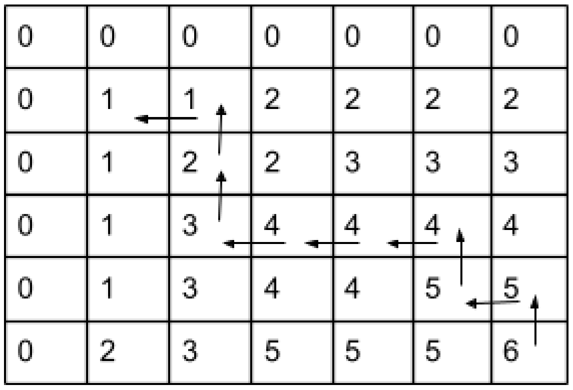


```
PWMW   (W, i , j)
        # step1 :store the subproblems (find the maximum number of weeds that can be obtained when that
        cell was reached to.
        initialize S[a,b] = 0 for 0 <= a <= i and 0 <= b <= j
        for d in 1…i
                for e in 1…j
                        if S[d-1, e] > S[d, e-1]
                                S[d,e]  =  S[d-1,e] + W[d-1,e-1]
                        else
                                S[d,e]  =  S[d,e-1] + W[d-1,e-1]
        # step2: track the cells from the bottom to top to find the path itself
        a = i
        b = j
        While a != 0 and j != 0: # while (1,1) is not inserted to the path.
                If S[a,b-1] > S[a-1,b] # if the value of the left cell is greater than the upper cell
                        Add (a,b) to path
                        Decrement b by 1 to go left
                Else: # the value of the left cell = upper cell OR the upper cell > left cell
                        Add (a,b) to path
                        If a - 1 > 0: # if upper cell is navigable
                                Decrement a by 1 to go up
                        Else:
                                Decrement b by 1 to go left
        Reverse the elements of path
        Return path
```

Note: My algorithm goes up if the values in the left and the upper cells are equal and the upper cell is navigable.

As an example, for the above algorithm, I will run this algorithm for the given green matrix.
i is 5 and j is 6 in this case.

0	0	0	0	0	0	0
0	1	1	2	2	2	2
0	1	2	2	3	3	3
0	1	3	4	4	4	4
0	1	3	4	4	5	5
0	2	3	5	5	5	6



0	0	0	0	0	0	0
0	1	1	2	2	2	2
0	1	2	2	3	3	3
0	1	3	4	4	4	4
0	1	3	4	4	5	5
0	2	3	5	5	5	6

(c) Asymptotic time and space complexity analysis of your algorithm.

I will count the number of solving subproblems to find the time complexity.
There are 2 for loops. The outer loop iterates as the number of rows of the given matrix.
The inner loop itself iterates as the number of columns of the given matrix.
So, iteration occurs as the number of rows of the given matrix x the number of columns of the given matrix.
If the given matrix W is the size of n x m, then n x m iterations run. And at each iteration, constant work is done: O(1)

The above part takes O(nm) x O(1) = O(nm).
Tracking the path takes O(n + m). Because tracking can be done always by going left or up. It starts from the right bottom corner and goes to the left upper corner. It always takes O(m+n). (m+n-1 exactly).
That's why, the whole algorithm takes O(nm) + O(n+m) = O (nm + (n+m)) = O(nm)
As the space, I use a n x m matrix. So, the space complexity is O(nm)

(d) Experimental evaluations of your algorithm: plot the results in a graph, and discuss the results (e.g., are they expected or surprising? why?)

To do experimental evaluations,
I followed the following steps:

Decide on how many different input sizes will be given to the PMWM function.
Divide this number to equal intervals.
For each input size (nm), enter the number of rows (n) and the number of columns (m).
Create a matrix size of nm. Initialize the matrix with 0's.
Settle weeds to this matrix. Determine the locations of the weeds randomly until the total number of weeds does not exceed (nm/2). If the randomly selected location was selected before, choose another location.
Record the time.
Call PMWM for the weeded matrix.
Stop recording the time. Calculate the time between.
Add each measured time to a list.
Plot a line graph with respect to the input size and the measured time list.

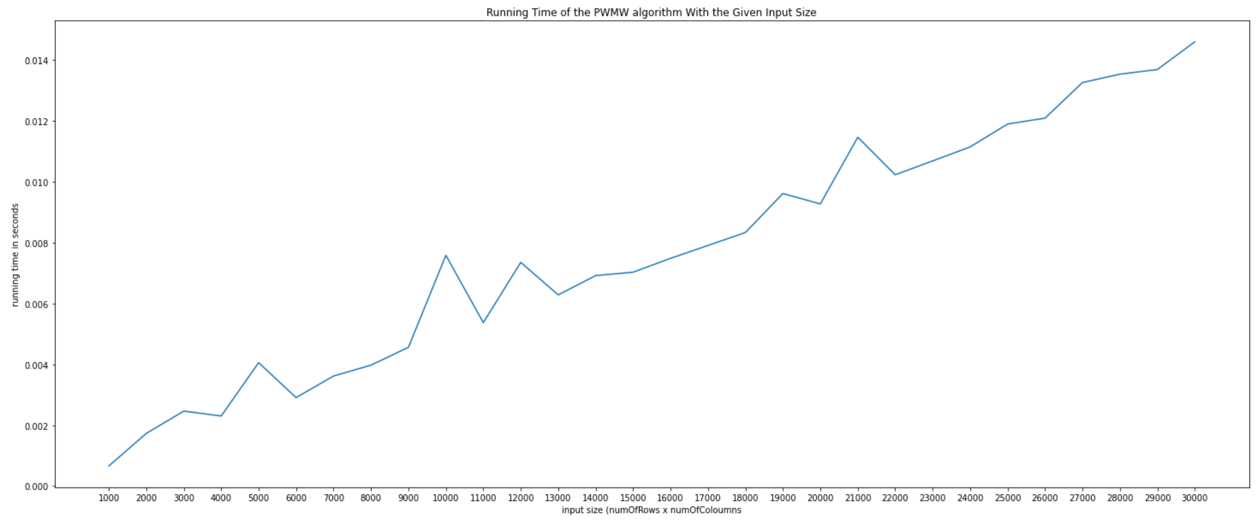
The above plots were created by taking inputs (n and m values) 30 times.
Input size was increased by 1000 at each time.

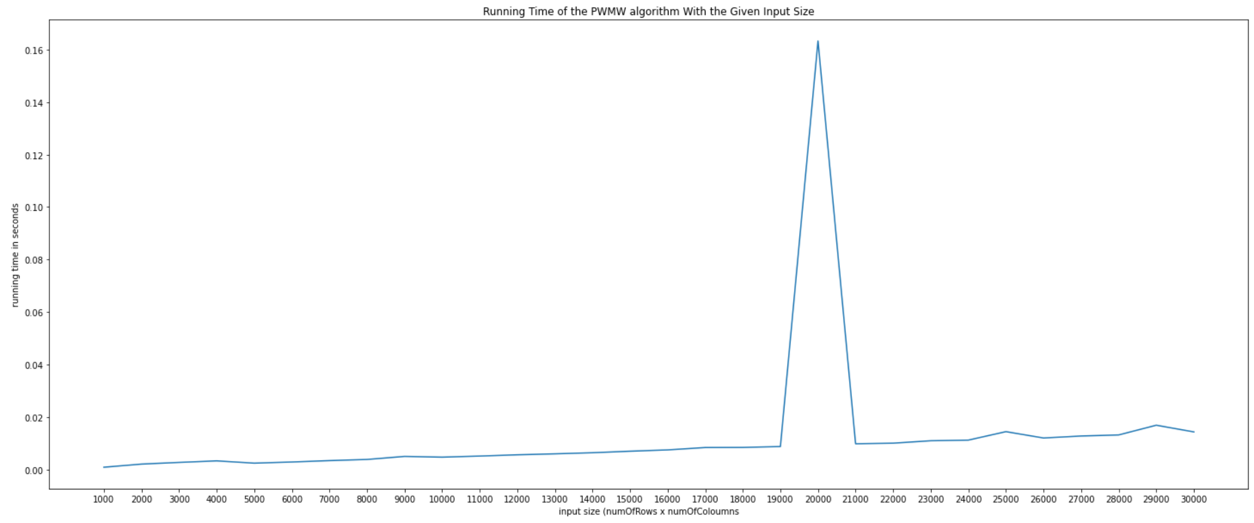
The plots show that the algorithm takes linear time, although at some points there are some exceptions.

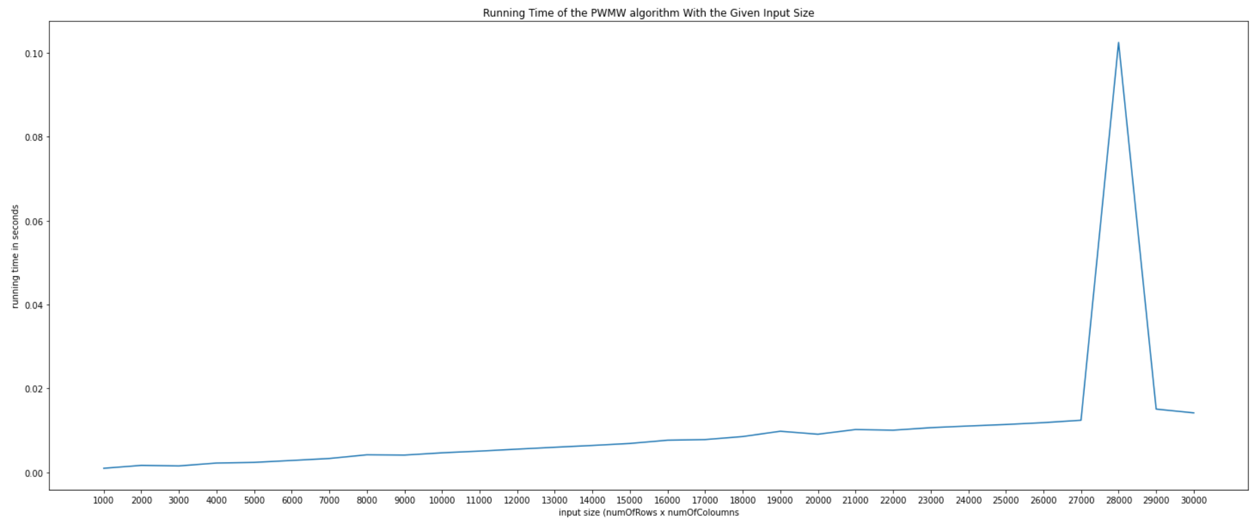
However, the graph doesn't seem to be linear when the input sizes are relatively small and the number of times to execute the algorithm is relatively small.

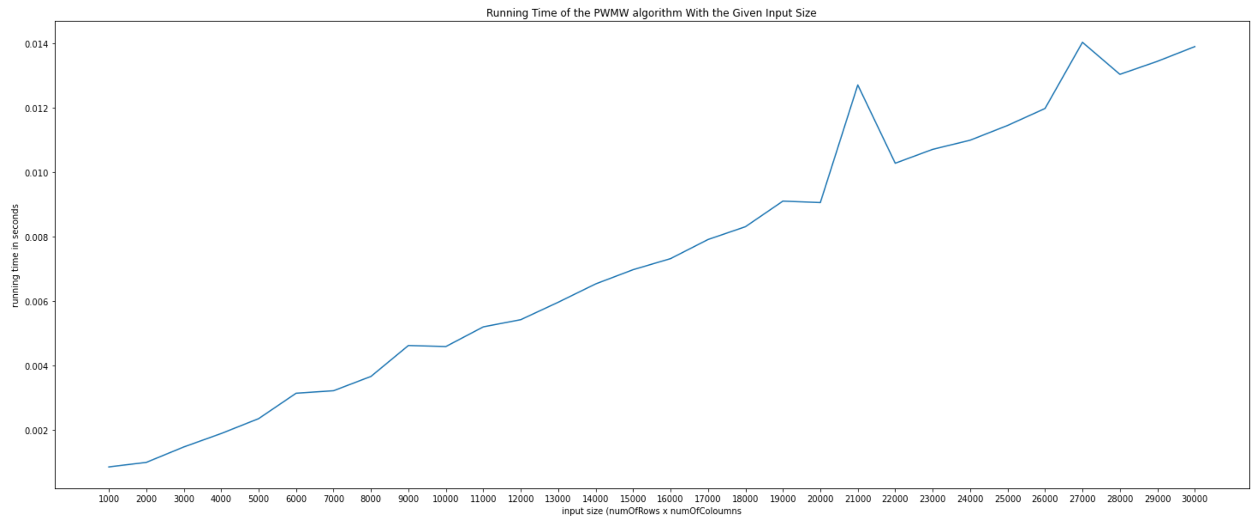
My expectation was O(mn). The experimental results are as I expected because my input size is nm and the expected time complexity is O(mn), which requires a linear line graph. The exceptional cases (like the case in the second graph when the input size is 20000) may stem from the running conditions on my computer during the execution of this case.

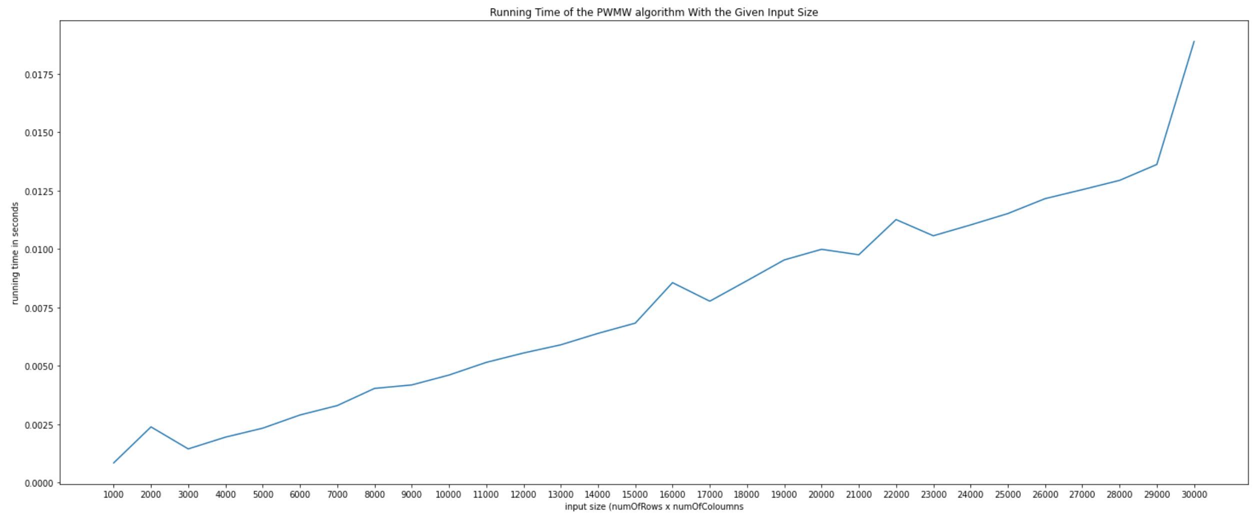
Here are some execution results:

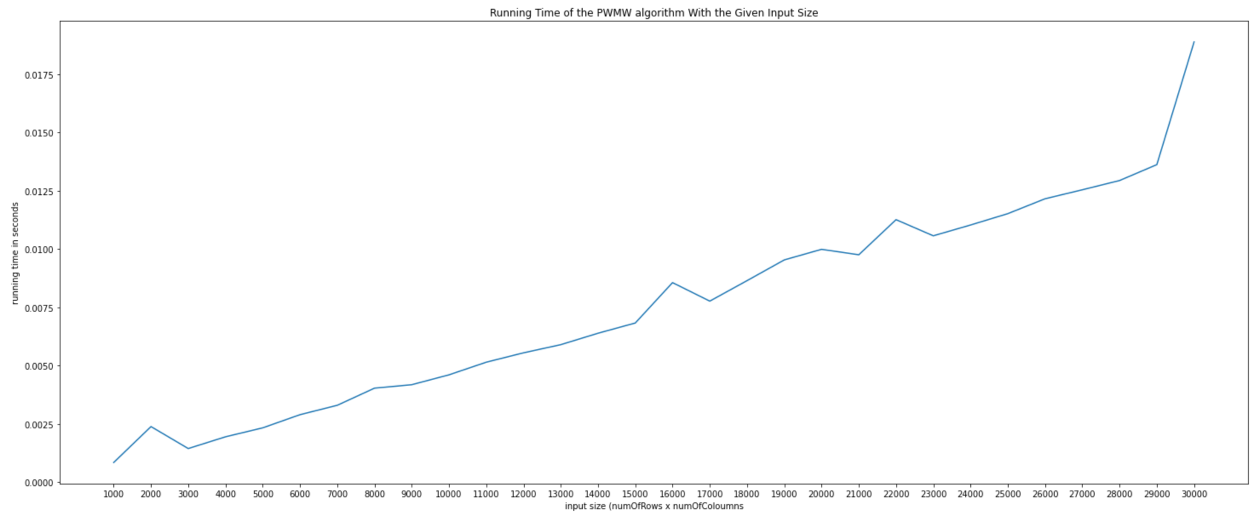


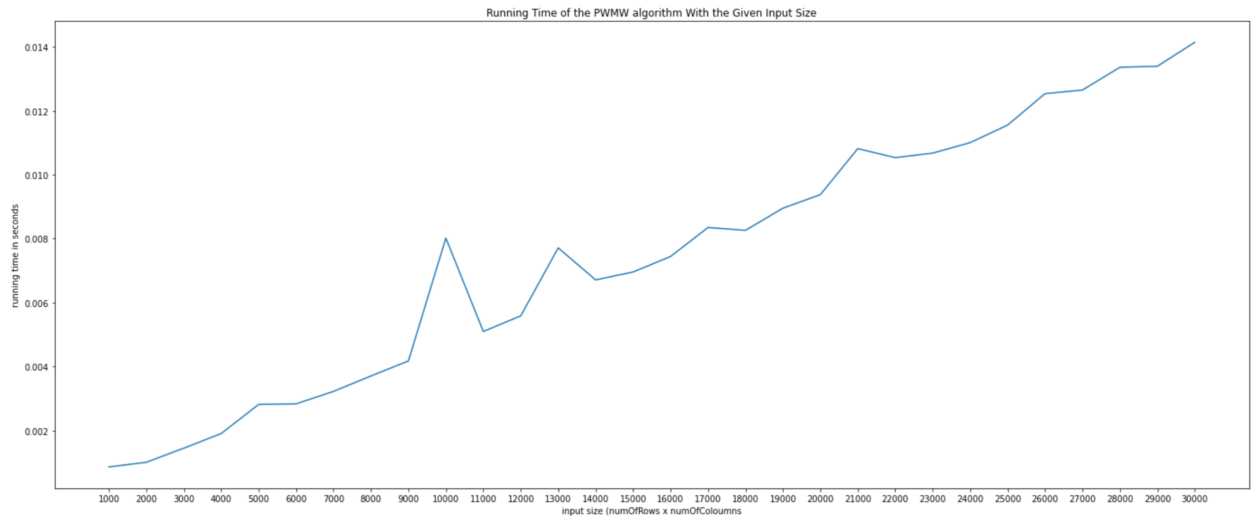



Running Time of the PWMW algorithm With the Given Input Size
running time in seconds
input size (numOfRows x numOfColoumns
0.00
0.02
0.04
0.06
0.08
0.10
1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000 13000 14000 15000 16000 17000 18000 19000 20000 21000 22000 23000 24000 25000 26000 27000 28000 29000 30000


Running Time of the PWMW algorithm With the Given Input Size
running time in seconds
input size (numOfRows x numOfColoumns
0.002
0.004
0.006
0.008
0.010
0.012
0.014
1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000 13000 14000 15000 16000 17000 18000 19000 20000 21000 22000 23000 24000 25000 26000 27000 28000 29000 30000


Running Time of the PWMW algorithm With the Given Input Size
running time in seconds
input size (numOfRows x numOfColoumns
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000 13000 14000 15000 16000 17000 18000 19000 20000 21000 22000 23000 24000 25000 26000 27000 28000 29000 30000


Running Time of the PWMW algorithm With the Given Input Size
running time in seconds
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000 13000 14000 15000 16000 17000 18000 19000 20000 21000 22000 23000 24000 25000 26000 27000 28000 29000 30000
input size (numOfRows x numOfColoumns


Running Time of the PWMW algorithm With the Given Input Size
running time in seconds
0.002
0.004
0.006
0.008
0.010
0.012
0.014
1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000 13000 14000 15000 16000 17000 18000 19000 20000 21000 22000 23000 24000 25000 26000 27000 28000 29000 30000
input size (numOfRows x numOfColoumns