CS 301 A 4

Oytun Kuday Duran (28357)

December 2022

1 Report

Note: Firstly, notice that the second part (finding path) of this problem has a similar solution to the 1-0 knapsack problem. Firstly, we need to find the maximum weed amount that can be collected by having a similar memoization matrix approach that was in LCS problem in order not the calculate the maximum amount of weed that can be colleted in previous (left and upper) cells which are overlapping subproblems again and again. Secondly, we need to use our memoization matrix and work backwards from the cell at the very end to find a path that we can use to collect the most amount of weeds.

1.1 (a) Recursive formulation

1.1.1 Explanation

Notice that we have common subproblems. For example, when wanting to get the maximum while passing a cell, we want to consider all paths that could be taken till reaching that cell. To not repeat the calculation of these possible paths (Overlapping subproblems) again and again. For example, to get the value of maximum amount of weeds that can be collected till c(i,j), we need to know both maximum amount can be collected from c(i-1,j) and c(i,j-1). Similarly, to get the value of c(i-1,j+1), w we need to know c(i-2,j+1),c(i-1,j). It can be observed that it is unnecessary to calculate path to c(i-1,j) here over and over again.

Starting from the first cell (1,1), since we can go the cell from either left or upside, we want to consider which one allows us to collect max amount of weeds. Notice that at left side, upper side, and first cell of the matrix, we won't be able to check respectively left, upper and both previous cells. For that reason, we need to initialize them first. We only check the existing cells (none for first cell, one upper cell for the rest of the first column, and cell at the left for the rest of the first row) and add weight of farm matrix to include whether there is a weed or not. Afterwards, for the rest of the matrix, we check both upper and left cells, and take the one which is higher to maximize the amount of weeds that needs to be collected. As a result, we will only calculate the max amount of weeds that can be collected while going to a cell only once. At the end, the last matrix stores the maximum amount of weeds that can be collected by going only right or down from the start.

1.1.2 Recursive formula

Assume that our dynamic programming matrix has n rows and m columns, then, our iterators; i goes from 1...n and j goes from 1...m. Given a matrix namely w[i,j] as in question which has only 1 when there is weed on specific block at i,j coordinates in matrix and 0 else, following recursion can be formed:

$$\text{i} \rightarrow \{1, \ldots, n\}, j \rightarrow \{1, \ldots, m\} \text{ such that}$$

$$m[i,j] = \begin{cases} w[i,j], & \text{if } i = 1 \text{ and } j = 1 \\ m[i-1,j] + w[i,j], & \text{if j } = 1 \text{ and } 1 < i \leq n \\ m[i,j-1] + w[i,j], & \text{if i } = 1 \text{ and } 1 < j \leq m \\ max(m[i-1,j], m[i,j-1]) + w[i,j], & \text{elsewhere} \end{cases}$$

In the second part, we will use our matrix that we found max amount of weeds that can be collected, go backwards to find an optimal path. We will use the previous matrix namely m, and iterate backwards to generate a list of coordinate pairs (i,j) for path. It was noted by TA's that recurrences for post processes are not needed to be included in the report.

1.2 (b) Pseudocode of your Algorithm

Input: w[i,j] that is given matrix which has 1 for every block that has weed and 0 elsewhere. $1 \leq i \leq n, and\ 1 \leq j \leq m$

Output: path[(1,1),...,(n,m)] (A list which has ascending coordinate pairs (i,j) for robot to pass by.)

```
#Create an empty list named path
if w is empty do
    print warning message
    return path
#Create a new matrix m with n rows and m columns
for i=1 to n do
    for j=1 to m do
        m[i,j] = 0     # Initialization of memoization matrix
m[1,1] = w[1,1]     # First cell (Base case)
for i=2 to n do
    m[i,1]=m[i-1,1] + w[i,1]     # Going through the leftmost side of matrix
for j=2 to m do
    m[1,j]=m[1,j-1] + w[1,j]     # Going through the upmost side of matrix
for i=2 to n do
    for j=2 to m do
        m[i,j] = max( m[i,j-1] , m[i-1,j] )+ w[i,j]     # Going through the rest of the matrix
# Second part starts here:
i=n, j=m
while i ≥ 1 and j ≥ 1 do
    path.insert(0,(i,j))     # Adding the new coordinate pair to the start of list
    if i equals to 1 do
        j = j-1
    else if j equals to 1 do
        i = i-1
    else do
        if m[i-1,j] > m[i,j-1] do
            i= i-1
        else do
            j= j-1
return path
```

Note: This implementation also works on multiple weeds on one block as well as the farms that have multiple equally optimal paths. In the case of multiple optimal paths, because of second part's last else statement, since they are equal, it prefers the vertical path.

1.3 (c) Asymptotic time and space complexity analysis

Assume that Row amount = m, Column amount = n

Asymptotic Time Complexity Analysis:
In the first part, creation of list and memoization matrix takes constant amount of time (O(1)). Initializing the memoization matrix values takes row*col assignment operations each take a constant time,

so $\Theta(m * n)$. Afterwards, traversing through the first row and column of memoization matrix takes n and m iterations, respectively, and we will check the weight matrix to know if there are weeds again in constant time as well as the previous cell, since addition, checking a value in matrix with known coordinates take constant time, this step is $\Theta(m + n)$. For the rest of the memoization matrix, we will iterate all the remaining blocks, each will take a constant time again for checking the value of weight matrix and values of left and upper cells in memoization matrix, and for comparison/addition operation, since this part will be repeated (m-1)*(n-1) times, this step is $\Theta(m * n)$.
In the second part that we use to find the optimal path, assignment operations take a constant time O(1). Then we will iterate the process till we reach to the first block of the memoization matrix, since each time we will decrease i or j by 1 and i=n j=m at start, this process will take n+m-1 iterations till it exists the while loop that each iteration will take a constant amount of time for assignment, comparison operations and insertion to list, so this process will take k*(n+m-1) which is $\Theta(n + m)$ time.
So the overall time complexity would be:
$\Theta(m * n + m + n + (m - 1) * (n - 1) + m * n + m + n + c) = \Theta(m * n)$ where m is the row and n is the column amount.

Asymptotic Space Complexity Analysis:
The size of the first memoization matrix is c(m*n) + k where c and k are constants respectively for needed memory for each cell in matrix and for the header in memory space, which has a space complexity of $\Theta(m * n)$. For the second part, the path list consists of coordinates of cells that needs to be visited. Since robot can only go right and down, it can -and has to- move at most n+m-1 times. Since we don't include the base cell as the robot is initially at there, it can make at most n+m-2 moves that we need to store for the optimal path (n+m-1 if we include the base cell in path as I did), each needs a constant amount of memory as well as the header of list in memory space, which has a space complexity of $\Theta(n + m)$.
Notice that here there is a constant amount of memory is also needed for the other factors such as the iterators (i,j), So the overall space complexity would be c1(n+m-1) + c2(m*n) + k = $\Theta(m * n)$.

1.4 (d) Experimental evaluations of algorithm and results

From the table below, it can be seen the results for experimental evaluations. The algorithm was tested with N x M matrices where n=m for values that can be seen below for easier observation with increasing row/column size. By using random library of python, matrices were filled with weeds in random places. Random Generated test cases of different sized matrices are tested and; average and standard deviation of the results were evaluated. Moreover, to observe more, a matrix from random generated test case was expanded from 3 till 15 with preserving the ratio and structure of weeds, were ran 10 times and average/std. dev. were evaluated as can be seen in table below.

Testing with	m = n = 3		m = n = 6		m = n = 9		m = n = 12		m = n = 15	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Same Case	1.71e-4	8.2e-5	2.24e-4	9.23e-5	3.64e-4	1.14e-4	9.93e-4	3.23e-4	1.3e-3	5.79e-4
Random Gen.	1.78e-4	6.9e-5	2.14e-4	8.73e-5	3.71e-4	1.41e-4	9.73e-4	2.89e-4	1.27e-3	5.76e-4

From the graphs in figure 1 below, it can be observed that, the growth of mean running time is polynomial/quadratic and it matches with our expectance of $\Theta(m*n) = \Theta(n^2)$ (when n=m). Moreover, It can be observed that there is also a similar growth in the standard deviation with increasing n=m values. I also observed that increase in std. dev. is almost negligible compared to the mean running time as can be seen in figure 2 below.
In the benchmark suites of second part, there are lots of other graphs and features were used to understand this algorithm better including comparison with size for different n/m ratios. From what we see in this initial experimentation, our expectations are satisfied. Also, figure 2 (Screenshotted from the second part of assignment) shows that the n/m ratio doesn't affect the result and only the total size does. In the testing, as can be seen in second part, the means and standard deviations of matrices with

different ratios were taken (And they were all almost identical). It can be easily observed that as in our expectation, although observation is harder, growth is polynomial versus size. I also observed that although std. dev. doesn't very much compared to n or m, it vary almost as much as the mean against size. In graph, it can still be observed that mean's growth dominates the standard deviation's growth. Although my initial experimentations may not be the most accurate,they support to my expectations and claims. In second part, more elaborative observations,benchmarks and experimentations will be evident. Moreover, I believe that in 1 row or 1 column matrices with other dimension being very large (100x or more), the time it takes for m+n-1 iterations almost reach m*n. However, m*n loops still dominate and growth is also $\Theta(m * n)$.

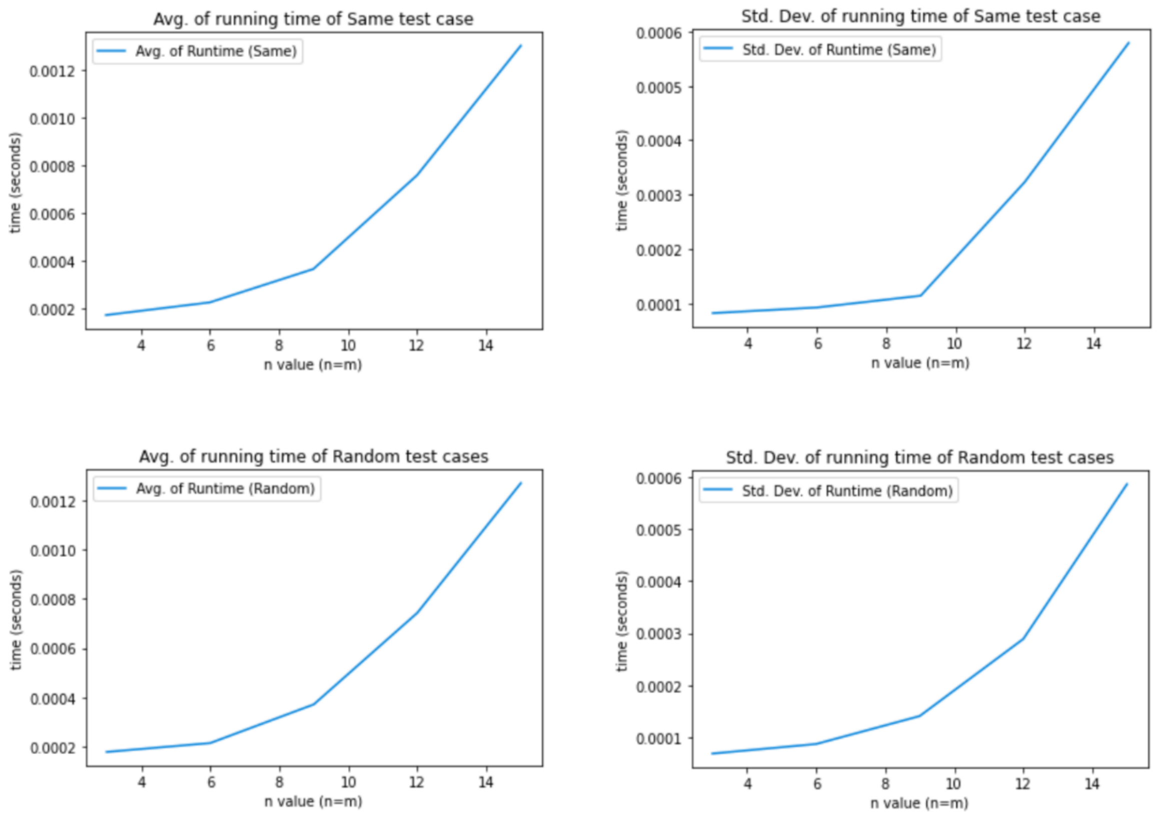

Figure 1: Graphs for Avg. and Std. Dev. of Running Times

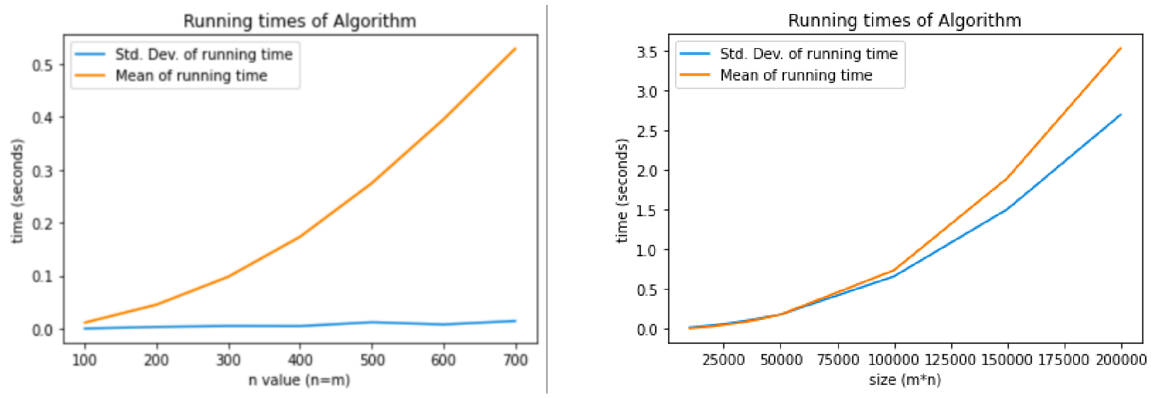

Figure 2: Plot for Avg. and Std. Dev. of Running Times on large n=m and size (m*n)