# cs301 Assignment-4

Hasan Berkay Kürkçü - 27851

December 2022

## Problem – Agricultural robotics problem

(a)Let's assume weed information is given as 2d array w with n row and m column. In w, w[i][j] == 1 implies there is a weed in the cell that is at i'th row and j'th column and w[i][j] == 0 implies there is not weed in the cell or that cell is clear.

In the question we are given that robot can move to either to the right or to down. Then in the recursive formulation for each cell visited we will consider left and upper cell of it to find the maximum number of weeds up until the current cell. Then, let f be an array with n row and m column as given w array and let f[i][j] be the maximum number of weeds cleaned up until i'th row and j'th column. Here there are 2 cases we need to think where we are at the starting edge (it should be directly equal to w[1][1] and where we are at the edge of farm where they will check only above cell or left cell.

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

(b) Normally i would go with top to bottom approach since it is generally easier to come up with. However, here we explicitly know the table we need to fill and that is why i chose bottom-up approach based on the recursive formulation given above.

Pseudocode:

**Algorithm 1** Fill the f array (f is the same f in recursive formulation)

**Input: 2d w array with n rows and m columns storing weed information**

**As output: filled f array (we can assume given as reference parameter)**

```
1: for i = 1 to n do
2:     for j = 1 to m do
3:         assign f[i][j] to 0
4:     end for
5: end for
6: assign f[1][1] to w[1,1]
7: for i = 2 to n do
8:     assign f[i][1] to f[i-1][1] + w[i][1]
9: end for
10: for j = 2 to n do
11:     assign f[1][j] to f[1][j-1] + w[1][j]
12: end for
13: for i = 2 to n do
14:     for j = 2 to m do
15:         assign f[i][j] to max(f[i-1][j],f[i][j-1]) + w[i][j]
16:     end for
17: end for
18: End of the algorithm
```

So what algorithm 1 pseudocode intends to do is initializing f array with n rows and m columns and all cells are set to 0 at the beginning. Then what we do is applying recursive formulation in part(a) above straight-forward.

**Algorithm 2** Find a path and number of maximum weed cleaned)

**Input: 2d f array with n rows and m columns storing max weeds cleaned so far up until the current cell**

**Output: String p holding path with maximum weed cleaned and m number of weeds cleaned**

```
1: Initialize p as empty string
2: assign r to n
3: assign c to m
4: while r > 1 and c > 1 do
5:     add (r,c) coordinate to p
6:     if f[r-1][c] ≥ f[r][c-1]) then
7:         decrease r by 1
8:     else
9:         decrease c by 1
10:    end if
11: end while
12: while c ≥ 1 do
13:    add (r,c) coordinate to p
14:    decrease c by 1
15: end while
16: while r ≥ 1 do
17:    add (r,c) coordinate to p
18:    decrease r by 1
19: end while
20:
21: assign m to f[n][m]
22: return reverse of p and m
23:
24: End of the algorithm
```

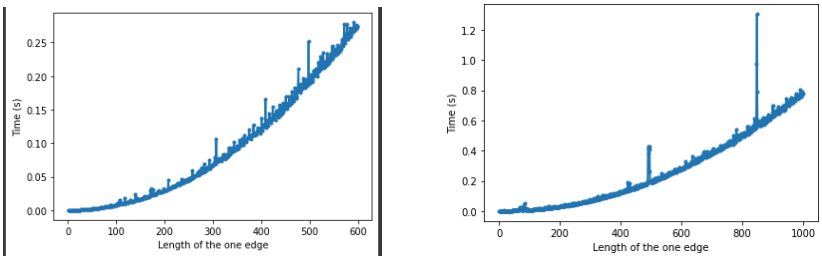
Now what algorithm 2 pseudocode intends to do is starting from the cell at n,m (the most right-below cell) and going for bigger value between left and above of it. (In case equality it does not matter where you go you just find a different path with the same maximum value. For the maximum number of weed cleaned on that path all we have to do return f[n,m]. Also note that after first while block i put 2 more while blocks in case we are on some coordinate that is on the left edge or above edge the farm.

(c) For the time complexity, we do two main operations in general: composing f array and then finding "a" path with maximum value of weeds cleaned. Here we say "a" because there may be many paths with same maximum value, but we are asked to find just one. Then, with respect to time complexity, for composing f array what we do is actually filling 2d array with n rows and m columns and while filling it, we first fill edges which is O(n + m), after that for cells that are placed inner parts we just check left and upper cells of them, take maximum of them which means we spend constant time while traversing the array that gives O((n-1) * (m-1)). Hence, time complexity for filling the f array is O(n+m) + O((n-1) *(m-1)) which is O(n*m). For finding path, note that at every step we decrease row or column value by 1 which means no matter what we go one step

further through first cell in the farm which is O(n+m). Hence, Asymptotic time complexity is O(n+m) + O(n*m) = O(n*m)

For the space complexity, w array is already given. We allocate 2d f array with n rows and m columns which is O(n*m). Also, we have string p for keeping the path but note that as explained in the time complexity we get closer to beginning cell at every step then length of the p has to be n + m − 1 which is O(n+m). Hence, we have O(n*m) space complexity.

(d)




(a) Until 600x600 (b) Until 1000x1000

Figure 1: 2 Figures side by side

So, in the left figure i ran program for every input size from 1 to 600 (i assumed square and randomly generated matrices to see the results better) and in the right figure from 1 to 1000. In lcs homework it was not easy to see asymptomatic analysis on the graph; however, here we see clearly O(n*m) $->$ O($n^2$) time complexity observing the graphs. Some outliers exists especially in the second figure but we are running the whole algorithm for 1000 times and occasionally it is normal we get a bit higher times for that input. Also i can say that whole iteration for 1 to 1000 took 4 minutes which again supports that our algorithm works efficiently and if we assume top to bottom program alternative i think again it would work efficiently but we would get much higher times because of its nature calling recursions at every step.

Code, benchmark instances and their results are in the zip.