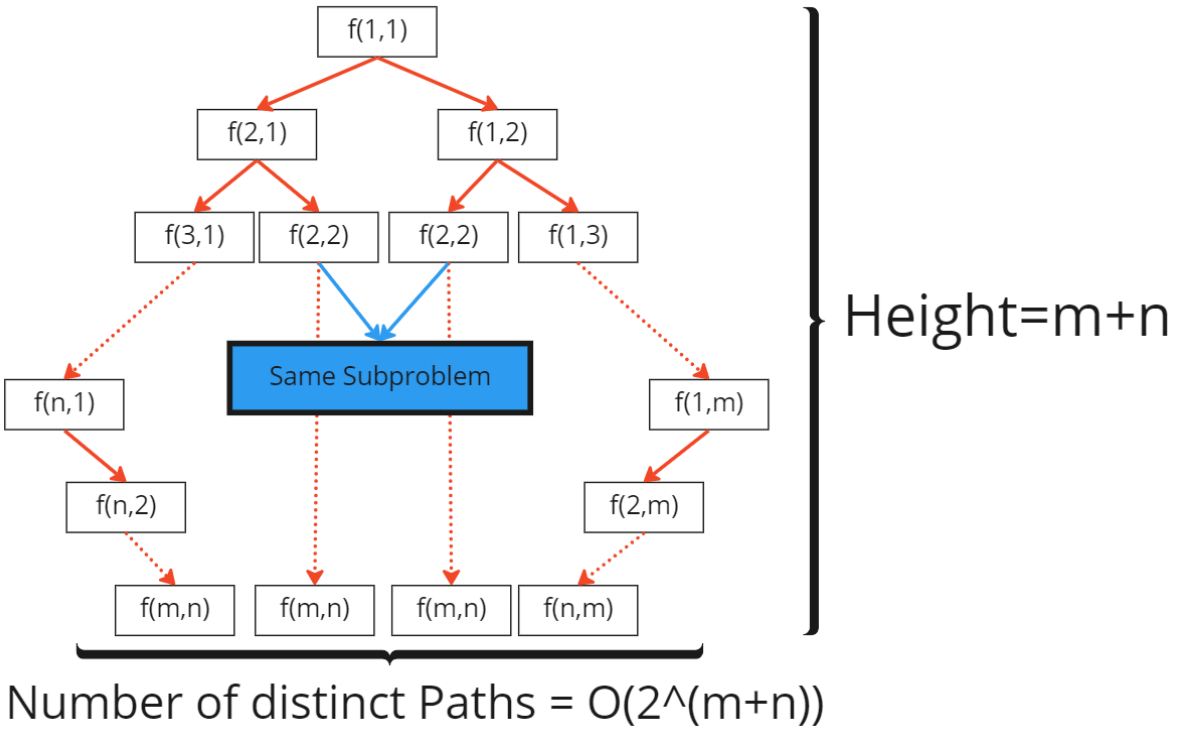
Homework 4

Ege Aktemur

December 2022

Problem Analysis

Let's start the analyzing the program from Naive Algorithm. Our Naive Algorithm will compute all possible paths and their points. Then it will give us the path(s) with most points. To analyze Naive Algorithm we can use recursion trees.

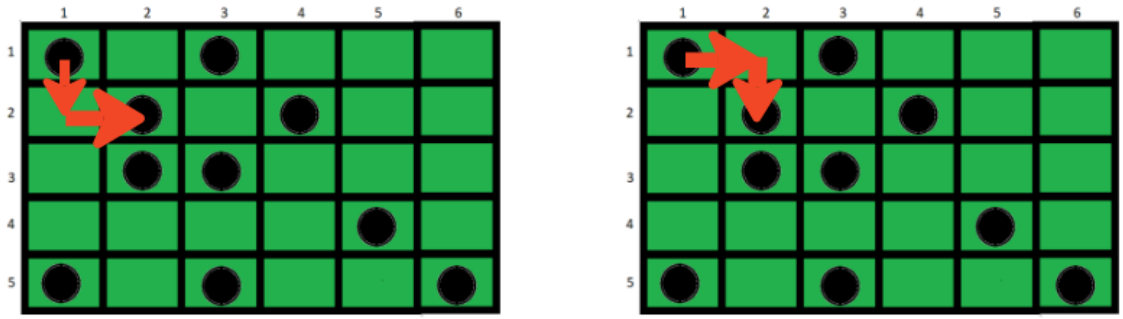


As it can be seen there are at most $O(2^{(m+n)})$ different path from the left top cell to right bottom cell since height of our recursion tree is m+n. However, as we can see there are same sub-problems. Which signs that we can use dynamic programming for this problem. Let's analyze if we can use Dynamic Programming.

1 Dynamic Programming

1.a Overlapping sub-problems

As we proved in the last section there are overlapping sub-problems in this problem. We can compute these sub-problems once and use them later if needed. To visualize the overlapping sub-problems I also prepared this graphic. Where it can be seen that (2,2) can be reached both by (1,2) and (2,1). Which means path from (2,2) to finish is calculated two times.



1.b Optimal Substructure

To prove that there is an optimal substructure in this problem we should prove that there is Topological ordering and solution to the problem can be created using solutions to the sub-problems.

1.b.i Topological Ordering

Since we can go over a cell only once because we can only go right or down we can say that in a path, one cell will be used only once. Which means that there will be no loops in the recursion tree which means there is a topological ordering. If the problem was including the case that robot would be able to go to all 4 directions the problem would not have topological ordering property.

1.b.2 Are the solutions to sub-problems construct a solution for the main problem?

To prove that sub-problems do construct a solution for the main problem we can use cut and paste method. So lets say that we created a path from (1,1) to (n,m) like [(1,1), ... , (i,j), ... , ((m,n)] called P1 that gives us the path with maximum number of weeds collected using the path. To prove that sub-problems do construct a solution for the main problem we should be able to prove that subpath of P1 [(1,1), ... , (i,j)] called P1a is the optimal solution between (1,1) and (i,j). Suppose that P1a is not the optimal route that gives us the maximum amount of weeds and there is P1b such that [(1,1), ... , (i,j)] gives us better result between (1,1) and (i,j), our statement that "P1a is the optimal solution between (1,1) and (i,j)" contradicts. We would use P1b instead of P1a using cut and paste.

1.c Conclusion and Recursive formulation

To sum up and create a recursive formulation of this problem we should first define our function.
Lets say our function is F(i,j) (where $1 \leq i \leq n$ and $1 \leq j \leq m$) gives us the route with the most weeds collectible from (i,j) to (n,m) in nxm table.
So we will try to compute F(1,1).
Lets say we are in cell (i,j) (where $1 \leq i \leq n$ and $1 \leq j \leq m$) and we are trying to decide if we should go to right or down. Then we will decide if path from (i+1,j) (if $i+1 \leq n$) or (i,j+1) (if $j+1 \leq m$) is more advantageous. To decide which one is a better node in the path we will recursively search.
Then we must get the maximum of (i+1,j) (if $i+1 \leq n$) and (i,j+1) (if $j+1 \leq m$).
To calculate the score of the route we must increase the score of the route by one if the cell we are on has a weed.
To define this I will use notion W(i,j) which will give 1 if (i,j) has a weed and give 0 if (i,j) has no weed.
Also we should make some restrictions so that we are always in the table. This restrictions are: F(a,b)=0 if a¿n or b¿0.
So our recursive formulation will be:

$$F(i,j) = \begin{cases} 0, & \text{if } (j = 0 \text{ and } 1 \leq i \leq n) \text{ or } (i = 0 \text{ and } 1 \leq j \leq m). \\ MAX(F(i-1,j), F(i,j-1)) + W(i,j), & \text{otherwise.} \end{cases}$$

Then since we showed that we could use dynamic programming for this problem we should how we can use. We should first decide how many distinct sub-problems are there. One way to calculate that is in the Naive Algorithm overlapping sub-problems was the same i and j's. Since i is in range (1,n) and j is in range (1,m), Therefore number of distinct sub-problems are m*n. Which can be also explained by calculating score of (i,j) to (n,m) once and save it to a external score table. This method is called Memoization and helps us to calculates routes faster if we need this specific route later.

(a) Recursive formulation

Recursive formulation is given below as it was calculated before.

$$F(i,j) = \begin{cases} 0, & \text{if } (j = 0 \text{ and } 1 \leq i \leq n) \text{ or } (i = 0 \text{ and } 1 \leq j \leq m). \\ MAX(F(i-1,j), F(i,j-1)) + W(i,j), & \text{otherwise.} \end{cases}$$

(b) Pseudocode

Algorithm 1 Creating Score Matrix

Input Weeds Table W with n rows and m columns.
Output Matrix that stores to data of solutions to each subproblem.

```
1: procedure CreateScoreMatrix(W, n, m)
2:     M ← (n + 1) * (m + 1) Matrix with all elements 0
3:     for i=1 to n do
4:         for j=1 to m do
5:             M ← MAX(M[i − 1][j], M[i][j − 1]) + W[i − 1][j − 1]
6:         end for
7:     end for
8:     return M
9: end procedure
```

Algorithm 2 Finding Optimal Path

Input Weeds Table W with n rows and m columns.
Output Path that maximizes the removed weeds and amount of this weeds.

```
1: procedure FindPath(W, n, m)
2:     M ← CreateScoreMatrix(W, n, m)
3:     x ← n
4:     y ← m
5:     P ← [x, y]
6:     while x > 1 or y > 1 do
7:         if (x is 1) or (M[x][y-1] bigger than M[x-1][y]) then
8:             y ← y − 1
9:         else if (y is 1) or (M[x-1][y] bigger than M[x][y-1]) then
10:            x ← x − 1
11:        else
12:            x ← x − 1
13:        end if
14:        P ← [x, y]
15:    end while
16:    return P, M[n][m]
17: end procedure
```

b.1 Explanation of Algorithms

b.1.1 Create Score Matrix

In this part I will explain my algorithm's Score Matrix Calculation part.

1. We will create an empty matrix to store matrix. As I explained in the first section we need n*m space to keep the score values. However since the indexing starts from 1 but not 0, I created (n+1)*(m+1) space for simplicity. But it can also be done with n*m space. However because it is $\theta(n * m)$ it does not matter.

2. Then for all i and j's I calculate if coming a cell from top or left is better. I do that by getting Maximum of left cell's and upper cell's. As a remainder since I initialized M with (n+1)*(m+1) all the uppermost cell's upper cell would be 0. Same way all the leftmost cell's left cell would be 0. So this would fix some additional checks.

3. Lastly algorithm would return the M.

b.1.2 Finding Path

In this part I will explain my algorithm's Path Calculation part.

1. Get M from CreateScoreMatrix, initialize x and y as the right bottom cell. Lastly add right bottom cell to path.

2. This is the most important step. We will start from the right bottom cell and try to find our path to the left top cell.

3. In order to decide we must go to left or top we should check some things. These are:

   - If we are in the leftmost column go up.
   - If we are in the topmost column go left.
   - If upper cell's score is bigger than left cell's score go up.
   - If left cell's score is bigger than upper cell's score go left.
   - If upper cell's score is equal to left cell's score go left (This can also be up).

4. Do these steps until reaching the start(1,1).

5. Lastly algorithm would return the Path and it's score.

(c) Asymptotic time and space complexity analysis

c.1 Time Complexity of Algorithm

Time Complexity of the Algorithm is $\theta(n * m)$ and the analysis is given below.

Algorithm 1 Creating Score Matrix

Input Weeds Table W with n rows and m columns.
Output Matrix that stores to data of solutions to each subproblem.

```
1: procedure CreateScoreMatrix(W, n, m)
2:     M ← (n + 1) * (m + 1) Matrix with all elements 0        } Θ(n*m)
3:     for i=1 to n do                                          } Θ(n*m)   } Θ(n*m)
4:         for j=1 to m do
5:             M ← MAX(M[i − 1][j], M[i][j − 1]) + W[i − 1][j − 1]
6:         end for
7:     end for
8:     return M
9: end procedure
```

Algorithm 2 Finding Optimal Path

Input Weeds Table W with n rows and m columns.
Output Path that maximizes the removed weeds and amount of this weeds.

```
1: procedure FindPath(W, n, m)
2:     M ← CreateScoreMatrix(W, n, m)                           } Θ(n*m)   } Θ(n*m)
3:     x ← n
4:     y ← m
5:     P ← [x, y]
6:     while x not 1 or y not 1 do
7:         if (x is 1) or (M[x][y-1] bigger than M[x-1][y]) then
8:             y ← y − 1
9:         else if (y is 1) or (M[x-1][y] bigger than M[x][y-1]) then   } Θ(n+m)
10:            x ← x − 1
11:        else
12:            x ← x − 1
13:        end if
14:        P ← [x, y]
15:    end while
16:    return P, M[n][m]
17: end procedure
```

c.2 Space Complexity of Algorithm

Space Complexity of the Algorithm is $\theta(n * m)$ since only space we allocated for this problem other than weeds table are the Matrix generated at CreateScoreMatrix which has size of $\theta(n * m)$ and the path array which has size of $\theta(n + m)$. Therefore overall space complexity is $\theta(n + m) + \theta(n * m) = \theta(n * m)$.

(d) Experimental evaluations

In this section I will test my algorithm using black box and white box testing.

d.1 Black Box Testing

In the Black Box testing I tested my algorithm for:

- Table given in the document
- Table full of 1s
- Table full of 0s
- Empty table
- Lastly, I wanted to see if it could handle large tables so I checked for 10x20

My algorithm did not have any troubles with these cases.

d.2 White Box Testing

Results in the White Box testing (means of 10000):
I ran these tests on Google Colab.

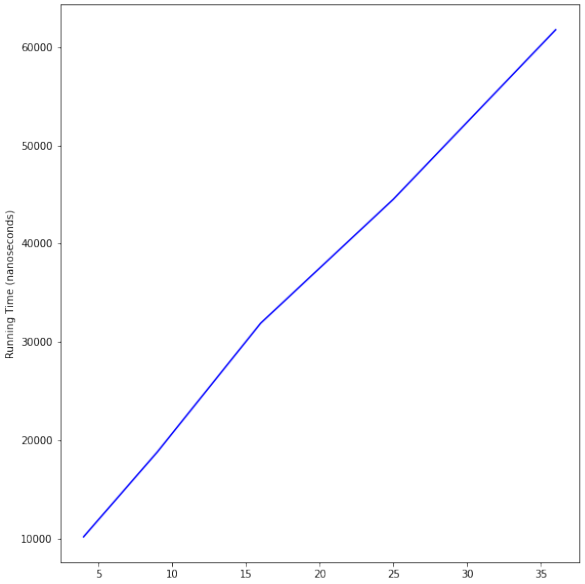
- Time recorded for 4 amount of cells (2x2 matrix): 10029 nanoseconds
- Time recorded for 9 amount of cells (3x3 matrix): 18579 nanoseconds
- Time recorded for 16 amount of cells (4x4 matrix): 31605 nanoseconds
- Time recorded for 25 amount of cells (5x5 matrix): 44030 nanoseconds
- Time recorded for 36 amount of cells (6x6 matrix): 61529 nanoseconds
- Time recorded for 8 amount of cells (4x2 matrix): 19501 nanoseconds
- Time recorded for 16 amount of cells (4x4 matrix): 30616 nanoseconds
- Time recorded for 24 amount of cells (4x6 matrix): 43458 nanoseconds
- Time recorded for 32 amount of cells (4x8 matrix): 55662 nanoseconds
- Time recorded for 40 amount of cells (4x10 matrix): 65348 nanoseconds
- Time recorded for 80 amount of cells (4x20 matrix): 129218 nanoseconds

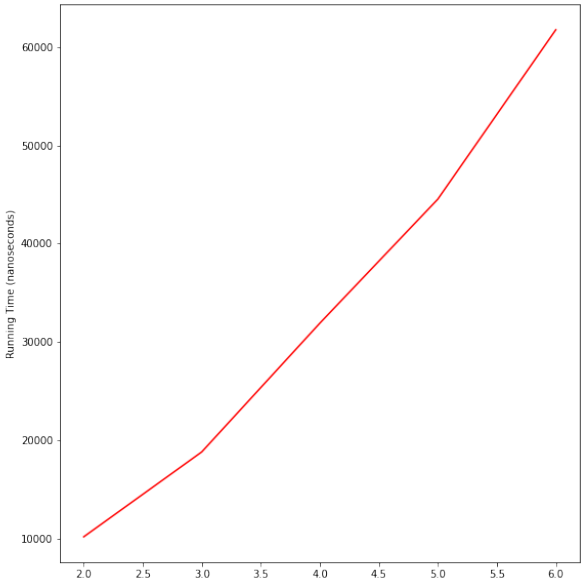
Because I wanted to see my algorithm's complexity when n = m and $n \neq m$, I prepared 2 different cases.

d.2.a n=m White Box

In this part I created cases for n=m=[2,3,4,5,6].

Plots of the results will be given below.





As it can be seen when we compared recorded times with number of cells algorithm gives results linear ($\theta(n * m)$). However when we compare by only n it becomes polynomial ($\theta(n^2)$).

This results prove that my algorithm runs in $\theta(n * m)$.

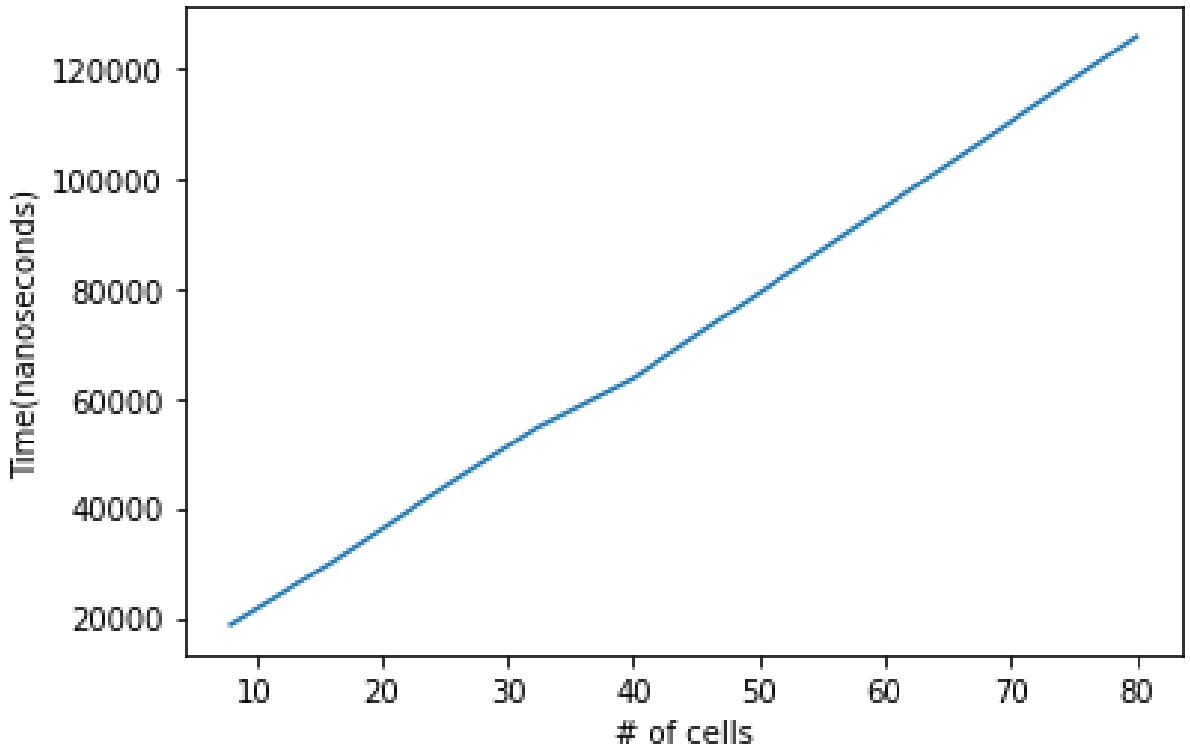
d.2.b n $\neq$ m White Box

In this part I created cases for $n \neq m$.

I specifically increased m while keeping n same.

Generated table sizes are: (4x2,4x4,4x6,4x8,4x10,4x20).

Plots of the results will be given below.



As it can be seen when we compared recorded times with number of cells algorithm gives results linear ($\theta(n * m)$).

This results prove that my algorithm runs in $\theta(n * m)$.