Find the maximum length snake sequence

DAA ASSIGNMENT-4 , GROUP 8

Swaraj Bhosle
IIT2019024

Ritesh Raj
IIT2019025

Utkarsh Garg
IIT2019026

***Abstract*—In this paper, we are devising an algorithm to find the maximum length snake sequence in a given matrix. This paper also contains the algorithm's time and space complexity analysis.**

***Index Terms*—matrix, array, dynamic programming, time complexity, space complexity.**

I. INTRODUCTION

A matrix of integers of size n*m is given, our algorithm should find the longest snake sequence from the matrix. If there exist many, it should print any one. There are some constraints for making snake sequence,

• snake sequence is made up of adjacent numbers in the grid.

• for each number, adjacent number on the right or below should be +1 or -1 of its value.

Matrix: Two dimensional arrays are called matrices. Dynamic Programming: It is an algorithmic technique for solving an optimization problem by breaking it down into simpler sub-problems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its sub-problems.

II. ALGORITHM DESIGN

We have 2 objectives for given problem:

• to find the length of the longest path which satisfies the given constraints.

• to print the longest path. Nomenclature

$$len[x][y] : maximumlengthofsequence$$

$$endingatcell(x, y)$$

Now it is given that we can only move in right or down direction when we are at a given cell in matrix. So for each cell, we can have the following recurrence relation

$$len[x][y] = max(len[x1][y], len[x][y1]) + 1$$

Assuming that the cells at $(x-1, y)$ and $(x, y-1)$ are valid with respect to given cell. Take, $len[0][0] = 0$, because starting cell will end on itself.

So, in order to find the length ending at cell (x, y) we need cells $(x-1, y)$ and$(x, y-1)$. We are going to keep track of all of this to make sure that finding length at any cell costs $O(1)$ time.

This formula has given us a way of essentially calculating the value of longest path ending at any cell (x, y). Only thing left is that we check which cell has the longest path and print that answer.

Now the remaining task is to print path. How to do it? Using an array which essentially keeps track of all the last value of the cell that have came inside the array. We can backtrack whole answer and store all these indexes in a array. Once we reach the starting point of the array we can print the answer in reverse order.

Algorithm 1 shows the complete approach for calculating distance. Algorithm 2 shows the complete approach for printing path.

III. ALGORITHM AND ANALYSIS

So, for each cell we require $O(1)$ time for comparison, and we have to repeat this for each and every cell and then find the longest possible length. Also, to keep track of last indexes and distance we will need an array.

A. Time Complexity

We have found out time complexity to be: $O(n * m)$, as total number of comparisons will be in the order of $O(n * m)$.

B. Space Complexity

Since, we will need 2 two-dimensional arrays of size n*m i.e. $O(n * m)$, and other variables require constant space. So, overall space complexity is $O(n * m)$.

IV. EXPERIMENTAL ANALYSIS

For conducting the experiment we have made a $testgenerator.cpp$ file. First we are going to generate 2 random numbers n and m; then we will generate n*m random values.

The range of n ans m is as below:-

$$1 \leq n \leq 1000$$

$$1 \leq m \leq 1000$$

To make sure that random values generated for each case are not same we are have used to seed function to change the initial seeding of random values. The initial seeding will depend on the time of execution of program. This test case generator file will create a $input.txt$, which is further used in our main program.

Algorithm 1: DP approach for calculating length of sequence

Data: Matrix of integers
Result: Length of longest snake sequence

```
/* matrix to keep track of
   length/distance                          */
dp[n][m] ← 0
/* keep track of last indexes of
   current cell to print                    */
p[n][m] ← −1
ans ← 0
for i ← 0 to n do
    for j ← 0 to m do
        dp[i][j] ← 0
        if valid(i-1, j) then
            dp[i][j] ← max(dp[i][j], dp[i − 1][j] + 1)
        if valid(i, j-1) then
            dp[i][j] ← max(dp[i][j], dp[i][j − 1] + 1)
        p[i][j] = greater((i-1, j), (i, j-1))
        ans = max(ans, dp[i][j])
```

Algorithm 2: Print longest snake sequence

```
/* stores index of paths                    */
arr[]
/* co-ordinates where indexes are
   ending                                   */
x, y
while x ≠ −1 and y ≠ −1 do
    // Store the current indices
    arr.push({x, y})
    // Go to last cell
    {x, y} = p[x][y]
reverse(arr)
print(arr)
```

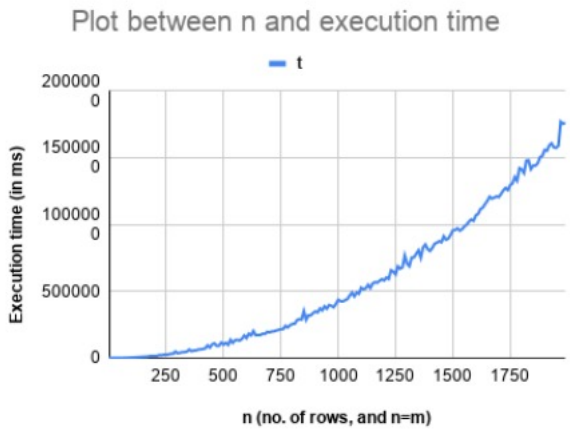

Fig. 1. Plot between n and execution time(t)

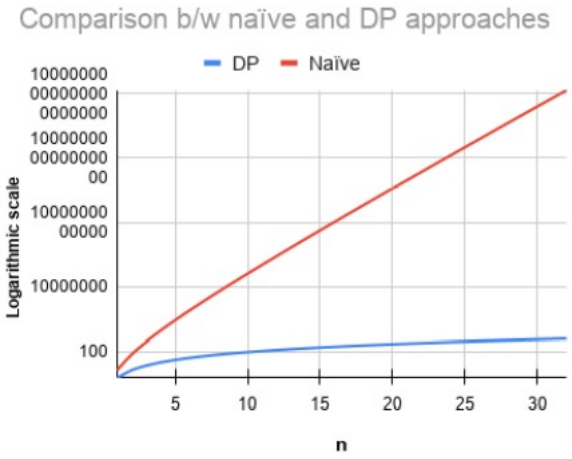

Fig. 2. Comparison between time complexity of DP and Naive approach

V. TIME CALCULATION

BEST	AVERAGE	WORST CASE
$\mathcal{O}(M * N)$	$\mathcal{O}(M * N)$	$\mathcal{O}(M * N)$

VI. CONCLUSION

In this document we discussed about the question which required concepts from dynamic programming to solve the problem. First if all we came up with a recursive formulae to get the desired answer. Then using appropriate tabulation, we have iteratively solved the problem to find the maximum length of snake sequence from input matrix.
Given problem could have been solved without using dynamic programming but, it would require more space and time complexity. So, we finally came up with an efficient approach that uses dynamic programming and finds our answer in $O(n * m)$ time and space complexity.
Fig 2 shows the comparison between the time complexities of naive and dynamic programming based approach to solve above problem.

VII. REFERENCES

1) https://www.geeksforgeeks.org/find-maximum-length-snake-sequence/
2) https://en.wikipedia.org/wiki/Dynamics
3) https://www.javatpoint.com/dynamic-programming-introduction