Finding the Length of Largest sorted component in a 50 x 50 matrix reverse diagonally

Harsh Meena - IIT20190005,
Asha Jyothi Donga - IIT2019006,
Sampada Kathar - IIT2019007

Date: 13-02-2021

Abstract

In this paper we have devised an algorithm which finds the largest sorted component reverse diagonally in a 2D matrix. We have determined both the time, and the space complexity of our algorithm by illustrating through some graphs.

1 Problem

Create a matrix of size 50*50 of numbers ranging from 0 to 9. Find the length of the largest sorted component reverse diagonally.

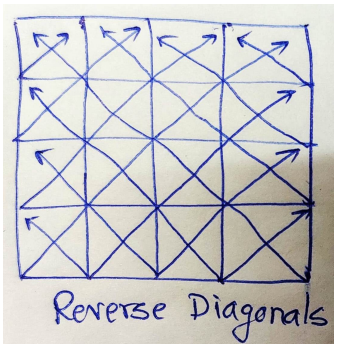
2 Keywords

Dictionary, Matrix, String, Array.

3 Introduction

A 2D array is a collection of multiple arrays, thus essentially forming a matrix with multiple rows and columns. A 2D array is usually indexed in a [row-1][column-1] format. This means the element in the 3rd row and 5th column would have the index [2][4]. Reverse Diagonal of an element would imply a diagonal going from the bottom up to the top.

Our problem statement requires us to find the largest subarray of elements that are diagonally next to each other, going from the bottom to the top, such that the resultant subarray is sorted.



4 Algorithm Design

Random numbered matrices are created by the function rand( ). For a given N x N size matrix, the number of Reverse Total Diagonals is 4n-2.

Stages:
Stage1: For each diagonal traverse the second element from the diagonal. if current element is greater than previous element, then this element helps in building up the previous increasing subarray encountered so far, else check if 'max' length is less than the length of the current increasing subarray.

Stage2: If true, then update 'max'.comparing the length of the last increasing subarray with 'max'.

5 Algorithm and Analysis

Initially maxlen = 1
Traverse the diagonals of direction from bottom-left to top-right

```
for(i=0;i<n;i++)
    j=n-2,k=i+1; len=1;
    while(k<=n-1 && j>=0)
        if(v[j+1][k-1]<v[j][k])
            len++
        else
            maxlen=max(maxlen,len)

        len=1
        k++ j--
    maxlen = max(maxlen,len)

        j=i-1,k=1
        len=1
    while(k<=n-1 && j>=0)
        if(v[j+1][k-1]<v[j][k])
            len++
        else
            maxlen=max(maxlen,len)

        len=1
        k++ j--
    maxlen = max(maxlen,len)
```

Traverse the diagonals of direction from bottom-right to top-left

```
for( i=0; i<n;i++)
    j=n-2,k=i-1   len=1
    while(k>=0 && j>=0)
        if(v[j+1][k+1]<v[j][k])
            len++
        else
            maxlen=max(maxlen,len)

        len=1
        k-- j--
    maxlen=max(maxlen,len)

    j=i-1,k=n-2 len=1
    while(k>=0 && j>=0)
        if(v[j+1][k+1]<v[j][k])
            len++
        else
            maxlen=max(maxlen,len)

        len=1
        k-- j--
    maxlen=max(maxlen,len);
```

Maxlen is the length of Longest Increasing Reverse Diagonal Length.

5.1 Time Complexity Analysis

As we can see from the above algorithm, we need to once iterate through the diagonals starting cell in the matrix which takes time of order N where N is the size of the cubical matrix.

Then from these starting cells we will traverse over the diagonal and find out the max length of increasing subarray, which takes time of order N.

So the total Time Complexity of this

Algorithm becomes of order of N^2.

The Best case is when, N=1 and he0nce TC for best case is $\Omega(1)$.

The Worst case will depend upon value of N. Hence, Worst case TC will be $O(N^2)$.

5.2 Space Complexity:

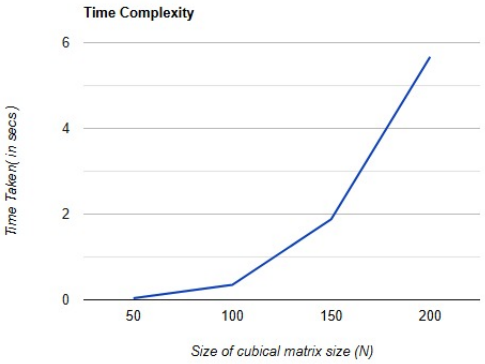
In this algorithm we have not occupied any extra space, hence Space Complexity (Extra Space used) is O(1).

6 Experimental Analysis

In the following table some cases are plotted,

N	Time Taken
2	0.0003
4	0.0005
8	0.0009
16	0.0011
20	0.0010
24	0.0019
30	0.02
40	0.034
50	0.04

Here N represents the size of the matrix, and the time taken is in seconds



The above graph shows the variance of Time with respect to size of matrix taken(N), and follows our theoretically calculated TC of $O(N^2)$.

7 Conclusion

From this paper we can conclude that the algorithm devised by us is an efficient way of solving our given problem. It takes no extra space which makes it efficient memory wise, and with a worst case complexity of only $O(N^2)$, it is efficient enough to process a 50x50 matrix, which makes it sufficient for our problem statement.

8 References

[1]https://www.geeksforgeeks.org/longest-increasing-subarray/
[2]https://www.geeksforgeeks.org/maximum-length-of-strictly-increasing-sub-array-after-removing-at-n
[3]https://stackoverflow.com/questions/14492047/longest-subarray-that-has-elements-in-increasing-order