Create a matrix of size 50 x 50 of numbers ranging from 0 to 9 and find the length of the largest sorted component reversed horizontally.

SHREYANSH PATIDAR(IIT2019018) , BISWAJEET DAS(IIT2019019), HRITIK SHARMA(IIT2019020)
4th Semester , Department of Information Technology
Indian Institute of Information Technology , Allahabad , India

---

Abstract— This Paper contains the algorithm to create a matrix of size 50 x 50 of numbers ranging from 0 to 9 and to find the length of the largest sorted component reversed horizontally. Two approaches have been taken and we will see the difference in complexity between both.

I. INTRODUCTION

A subsequence is a sequence contained in or forming part of another sequence.Sorting is the process of arranging the elements of a set in a fashionable order i.e either in ascending or descending order of the elements of the set.
This report further contains -
II. Algorithm Design
III. Algorithm Analysis
IV. Result
V. Conclusion

II. ALGORITHM DESIGN

Steps for designing this algorithm are -

Approach 1:

1. Assign values to a 2 D array of desired length using random function (n=50 in problem)
2. Iterate over a loop through the entire 2D array row wise.
3. In each row , use a dynamic programming approach to obtain its largest sorted sequence length.
4.Compute optimized LIS values in bottom up manner for each row.
5. For each row , store the value of its longest sorted sequence in an array row_wise_max[n].
6.print the maximum of all numbers in row_wise_max[n] array.

Algorithm 1:

```
int a[n][n] , LIS[n],row_wise_max[n];
for(i: 0→n-1 ){
  LIS[n] = {1};
    for( j: n-2→0 ){   // reverse iteration
        for(k: n-1→j)
            {     if(a[i][j] > a[i][k] &&
LIS[j] < LIS[k]+1)
                LIS[j]=LIS[k]+1;
}
}
Row_wise_max[i] = maximum(LIS[n]);
}
Ans = maximum(row_wise_max[n]);

print(Ans);
```

----- Approach 2:

1.Traverse 2D array row-wise.
2. Make a new arr[] array and assign value a[0][n-1] to arr[0] for each row i . Now using pointer to arr[] elements iterate remaining

array a[0][j] row wise , if the next element in a[0][j] is greater than the last element of arr[] then insert this element into arr[] else replace this element in place of element in a[0][j] which is just greater than or equal to that element.

3. Insertion here will be based on binary search technique(divide and conquer) and simple comparison.

4. Store the length of the longest sorted sequence of each row in the row_wise_max[] array.

4. We will follow the same steps for each row in the 2D array.

5. The maximum of all elements in the row_wise_max[] array would be the answer.

Algorithm 2:

```
for(i :0→n-1)
{
        while (r - l > 1) {
            int m = l + (r - l) / 2;
            if (v[m] >= key)
                r = m;
            else
                l = m;
          }
          return r;
}

  int a[n][n] , arr[n] , row_wise_max[n],pntr;
  for(i: 0→n-1 ){
       pntr=0,arr[0] = a[i][n-1];
      for( j: n-2→0 ){
           if (a[i][j] > arr[pntr])
                  arr[++pntr]=a[i][j] ;
            else {
          Index = search(arr,0,pntr,a[i][j]);
              arr[index] = a[i][j];
  }
  }
  Row_wise_max[i] = arr[pntr];
  }
  Ans = maximum(row_wise_max[n]);
  print(Ans);
```

----- III. ALGORITHM ANALYSIS

Approach 1:

For each row, the DP approach for LIS takes time $\propto n^2$. for each row , we need extra time $\propto$ n to calculate the maximum element in LIS.

Here the maximum value of n=100

So the time complexity will be $O(n(n + 2*n*(n+1)/2 + n)) = O(3*n^2 + n^3)$

$\mathbf{t_{best}}$**:** when n=0, = O(0) = 0ms

$\mathbf{t_{worst}}$**:** when n=100 , t_{worst} = $O(2.03 * (10^6))$

Approach 2:

Here, traversing over each row with size n will take time $\propto$ n . Finding pos for each element will take maximum log(n) time. And insertion operation will take time $\propto$ n.

So, the time complexity will be O(n*n*log(n)).

$\mathbf{t_{best}}$**:** when n=0, = O(0) = 0ms

$\mathbf{t_{worst}}$ **:** when n=100 ,=$O(2*10^4)$

SNo.	N	Algo 1	Algo 2
1	0	0	0
2	10	1300	200
3	20	9200	520.41
4	30	29700	1329.409
5	40	68800	2563.435
6	50	632500	4247.425

Table 1: values of O(n) for all algorithms

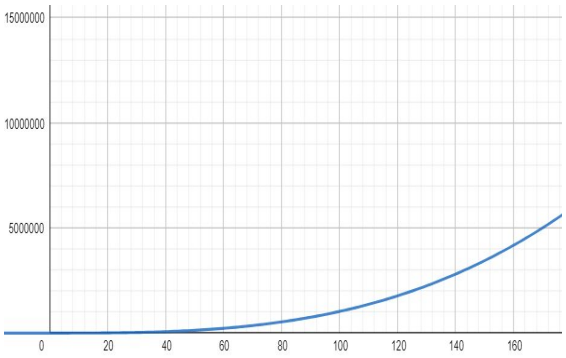

Fig 1: time complexity of algorithm 1

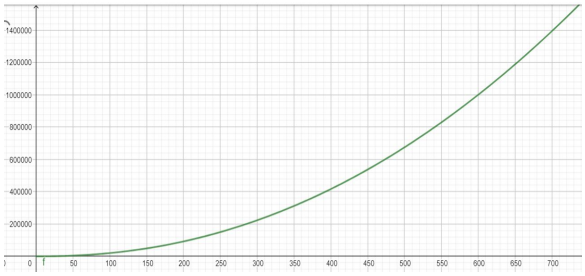

Fig 2: time complexity of algorithm 2

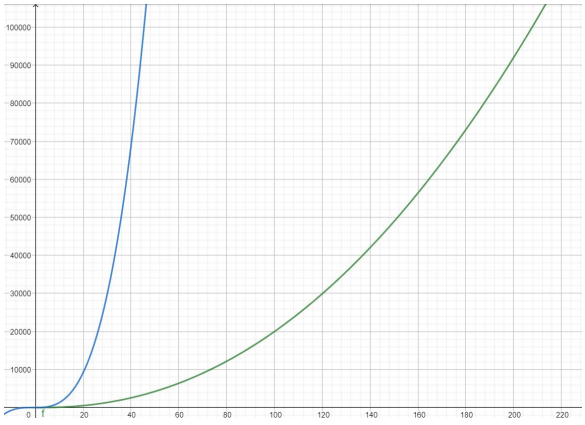

Fig 3: comparison of both algorithm

IV. CONCLUSION

Above two methods have different time complexities and meet to fulfill the problem statement. The order in which they are good can be listed as:

I. Approach 2
II. Approach 1

Based on the time complexities.

V. REFERENCES

https//en.wikipedia.org/wiki/Sequence_