Tracing Longest Sorted Path in Each Column of Matrix And in The Complete Matrix

Anmol Singh Sethi IIT2016040 | Anirudh Singh Rathore IIT2016091 | Dheeraj Chouhan IIT2016078 | Manish Ranjan IIT2016058 | Vikas Kumar IIT2016059

***Abstract*—Here, we have tried to design algorithms for two problems. First problem demands to find the longest sorted partition in each column of nxn matrix and the second problem demands to find the longest sorted child in the matrix. For the former problem an algorithm with time complexity $\theta(n^2)$ could be designed and for the latter one an algorithm of time complexity $O(n^2*4^{n^2})$ and $\Omega(n^2)$ could be designed.**

***Index Terms*—longest sorted subarray, longest sorted child of a matrix, time complexity, recursion, back-tracking**

I. Introduction and Literature Survey

The part (a) of the problem required us to trace the longest sorted path in every column of the matrix. Longest sorted path in each matrix means to find the longest sub-array which is either non-increasing or non-decreasing.
The approach to solve this part of the problem was to find the longest sorted path in first column of the array and to carry out the same procedure for every column of that matrix. To find the longest sorted path in a vertical column of the matrix, a simple linear traversal for that column and finding the maximum possible length.

The part (b) of the problem required us to trace the longest sorted path in the matrix. Starting from an element of the matrix we are allowed to move in four directions i.e. right, left, up and down. Traversing all such probable paths in all the allowed directions, we are required to print the path length of the maximum path along with the corresponding alphabets.
The concept of back-tracking has been used to solve this part of the problem. The backtracking approach incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution. So, to find the longest sorted path for the matrix, from each element of the matrix all possible paths are traversed and the one with the maximum path length is stored.[1]

II. Algorithm Design And Explanation

The algorithm has a simple approach which extends the basic algorithm for an array of size N. To understand the algorithm designed, let's first understand how to find a non-decreasing sub-array in an array of length N.
1. Consider a matrix of length N filled with random integers. To find the longest non-decreasing sub-array we take two integers, "max" and "len" both initialized to 1.
2. The array is then traversed from left to right.
3. If arr[i] >= arr[i-1] (i, being the index initialized from 1), the variable "len" is incremented. If that is not the case, the value of "max" is updated equal to the greater value between "max" and "len". "len" is then again assigned the value 1.
4. On the complete traversal of array the final value of either "max" or "len" is returned whichever is greater.
The basic algorithm can be extended to find longest non-increasing sub-array in a column of the matrix and to find the longest non-decreasing sub-array. The greater of the two shall be the longest sorted child for that column.
Since a NxN matrix can be considered equivalent to N vertical matrices that are put together. So, first, we find the longest sorted path in one column. This procedure can be carried out for every column to get the result. Hence, this solves our first part of the problem.

To solve the part (b) of the problem consider the following:-
We use the concept of backtracking to solve this problem. The approach is to traverse across the matrix and find the longest sorted chain.
1. Store the indexes of the characters in the temp array(in which each element is a structure which stores two integers).
2. For each element call the routine "longestpath" which searches the neighboring elements for the greatest. Also in this routine, we store the longest chain in ans1(using a loop till "rec")
3. In the routine "longestpath" mark the position we are in as visited, set a "flag" as 0 and store the indexes into the temp array.
4. Check the left character. If sorted, set flag to 1 and then call the same routine for the left neighbor by incrementing level as the length of the sorted chain has increased by 1. Similarly we check the right character, if sorted, set flag to 1 and call the routine "longestpath" for the right neighbor.
5. We check for the neighbor character below and the neighbor character above, update flag if condition satisfied and call the routine till there are no more characters satisfying any of the cases.
6. We check if the flag is 1 and if level is greater than "res" (if the chain is longer than previous) then we update the "res" with level (as the new length of the sorted chain)
7. Un-mark the position so that we can visit it again later and update the ans array with the indexes of the sorted chain we have obtained.

The pseudo-code for the algorithms described above is the following:-
Given : The matrix is NxN filled with random alphabets.

Algorithm 1 Algorithm to find longest sorted length in every column of an NxN matrix.

```
procedure SortedLength(array[100][100])
    for ( j = 0 to n-1 ) do
        max_inc ←1
        len ←1
        max_inc_index ←0
        for ( i = 1 to n-1 ) do
            if arr[i][j] >= arr[i-1][j] then
                len ← len + 1
            else
                if (max_inc < len) then
                    max_inc ← len
                    max_inc_index ← i − max_inc
                len ← 1
        if (max_inc < len) then
            max_inc ← len
            max_inc_index ← n − max_inc

        max_dec ←1
        len ←1
        max_dec_index ←0
        for ( i = 1 to n-1 ) do
            if (arr[i][j] <= arr[i-1][j]) then
                len ← len + 1
            else
                if (max_dec < len) then
                    max_dec ← len
                    max_dec_index ← i − max_dec
                len ← 1
        if (max_dec < len) then
            max_dec ← len
            max_dec_index ← n − max_dec

    if (max_inc > max_dec) then
        for (i=max_inc_index to max_inc+max_inc_index)
do
            print(array[i][j])
    else
        for (i=max_dec_index to max_dec+max_dec_index)
do
            print(array[i][j])
```

Algorithm 2 Longest Sorted Child of a matrix

```
procedure longestIncreasingPath(array[100][100])
    for ( i = 0 to n-1 ) do
        for ( j = 0 to n-1 ) do
            temp[0].x ← i
            temp[0].y ← j
            longestpath(array, i, j, 0)
    for ( i = 0 to res ) do
        ans1[i] ← array[ans[i].x][ans[i].y]

procedure longestpath(array[100][100],i,j,level)
    flag ← 0
    vis[i][j] ← 1
    temp[level].x ← i
    temp[level].y ← j
    if ( j>0 && array[i][j-1] <= array[i][j] && vis[i][j-1]
!= 0 ) then
        flag ← 1
        longestpath(array, i, j − 1, level + 1)
    if (j<n-1&& array[i][j+1] <= array[i][j] && vis[i][j+1]
!= 0 ) then
        flag ← 1
        longestpath(array, i, j + 1, level + 1)
    if ( i>0 && array[i-1][j] <= array[i][j] && vis[i-1][j]
!= 0 ) then
        flag ← 1
        longestpath(array, i − 1, j, level + 1)
    if ( i<n-1 && array[i+1][j] <= array[i][j] &&
vis[i+1][j] != 0 ) then
        flag ← 1
        longestpath(array, i + 1, j, level + 1)
    if (flag != 0 && level > res ) then
        res ← level
        for ( p = 0 to level ) do
            ans[p].x ← temp[p].x
            ans[p].y ← temp[p].y
    vis[i][j] ← 0
procedure main
    longestIncreasingPath(array)
    print(res+1,ans)
```

III. Time Complexity Analysis and Discussion

For part(a) of the problem the time complexity is $\theta(n^2)$. Intuitively speaking, we can say that for the complete traversal of one vertical column of the matrix takes time of order *n*. And since there are N columns in the matrix the time-complexity will be *n* x *n* = n^2. But consider the following proof for the mathematical analysis:-
The time units required for the complete traversal for average case is as follows
time = $16n^2$ + 19n +2
n^2 <$16n^2$ + 19n +2 <$37n^2$
g(x) <f(x) <h(x)
g(x) is the tightest lower bound of the function and h(x) is the least upper bound of the function. Since the order

of polynomials is n^2, the time complexity is $\theta(n^2)$. Note, we have used $\theta(n^2)$ because the time complexity in all the cases is the same. This problem does not have any best-case scenario or any worst-case scenario, because in any case whether the length is minimum i.e. 2 or the length is maximum i.e. N, the array has to be completely traversed giving the polynomial an order of n^2.

Hence, the time complexity for the part(a) of the problem is $\theta(n^2)$

For part(b) of the problem the complexity analysis is complex in itself. Consider the three scenarios:-

1. *Worst Case* : Consider an NxN matrix filled with the same English alphabet. To find the longest sorted path for this case the "longestpath" function will be called recursively and each time it will be called for the whole matrix in every direction. Since there are n^2 elements in total and there are 4 directions for which the function can be called, and this process will be carried out for n^2 elements giving it a total complexity of 4^{n^2}.And we have run the loop for n^2 elements of the matrix so the worst complexity will be with some constant k (k*n^2*4^{n^2}). This concludes our worst-case scenario.
2. *Best Case*:- Consider an NxN matrix filled with English alphabets such that the maximum length of sorted path from any element of the array is 2. For e.g.

$$\begin{pmatrix} a & b \\ b & a \end{pmatrix}$$

. For such a case the function will reach the terminal point as soon as the function is called for the second time from every element of the matrix. Taking any random element of a matrix we can show that it can go in four directions atmost and then it cannot go any further so it will always take a constant time "k" atmost so the time complexity will be n^2 only for the traversal of the matrix.
3. *Average Case*:- Consider an NxN matrix that is filled with English alphabets that are randomly generated. The time take to compute this will lie between the best-case and the worst-case. It is difficult to compute the time for such a case i general because it will be purely case-dependent. But the limits can be found.

To verify the theoretical claim, we plotted the graph between T VS N, to find out that the worst-case if proportional to k^{n^2} and the best-case is proportional to n^2 and all other cases lie in this range.
The time complexity of the part(b) of the problem is case dependent and cannot be generalized.

IV. Experimental Study

In this section of the repost the actual data of the time calculation has been shown. The time plots have been constructed using the unit time equations at every step of the computation. Further the data was written in a file and GNU plots were constructed to verify our claims in the previous sections.

TABLE I
TIME COMPLEXITY TABLE- PART(A)

n	$t_{average}$
1	14
2	69
3	160
4	293
5	456
6	667
7	896
8	1176
9	1489
10	1841

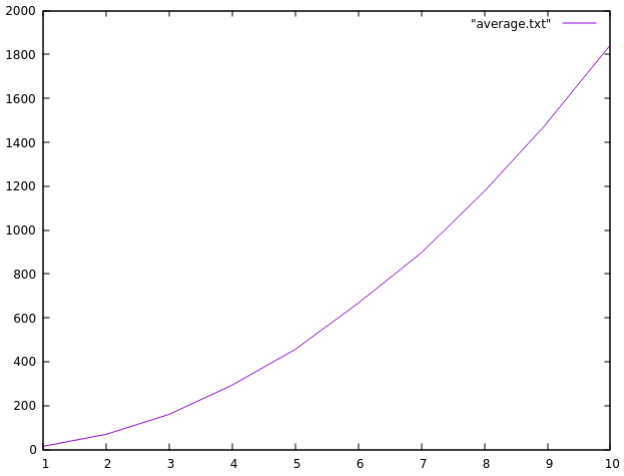

Fig. 1. Part(a) T(y-axis) VS N(x-axis)

TABLE II
TIME COMPLEXITY TABLE- PART(B)

n	t_{best}	$t_{average}$	t_{worst}
1	80	80	80
2	485	485	1429
3	1153	2323	29747
4	2121	2975	1283669
5	3389	5738	137719735
6	4957	11876	39295791125

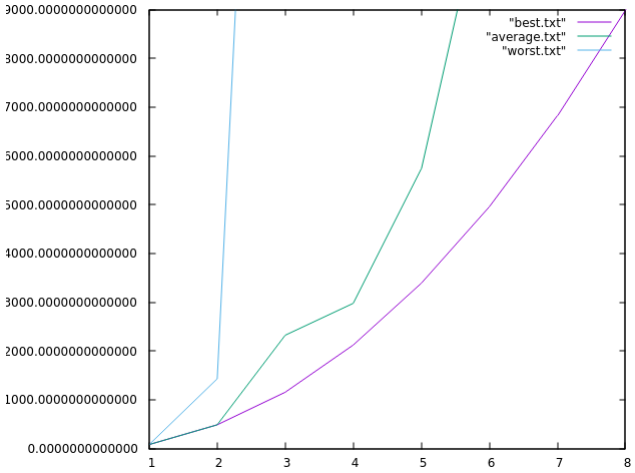

Fig. 2. Part(b) T(y-axis) VS N(x-axis)

V. Conclusion

Here, we have tried to design an efficient algorithm that finds out the maximum sorted path in every column of an NxN matrix and another algorithm to find out the maximum sorted path in the matrix, given constraint of moving in four directions. A simple approach of traversing the vertical columns to find maximum length was adopted for the first part. And the concept of back-tracking was adopted to solve the problem in the latter part. The details of the algorithms have been also discussed and experimental analysis of the designed algorithms was carried out.Moreover, all the possible cases have been discussed for the formation of as efficient algorithm as possible.

References

[1] https://www.geeksforgeeks.org/longest-increasing-subarray/

[2] https://en.wikipedia.org/wiki/Backtracking/

[3] http://jeffe.cs.illinois.edu/teaching/algorithms/notes/03-backtracking.pdf