IDAA432C Assignment-1 (Group: 41)

Aditya Vallabh	Kiran Velumuri	Neil Syiemlieh	Tauhid Alam
IIT2016517	ITM2016009	IIT2016125	BIM2015003

Question number: 11

Take a 10 x 10 matrix (n x n in general) filled with random integer numbers. Trace the longest sorted partition:-
a. in each column of the matrix
b. Trace the longest sorted child of the matrix.

I. INTRODUCTION

The first part of the question states that we have to find the longest sorted partition in each column of the matrix. That is we have to find the longest continuous sequence of elements in each column of the matrix which is either in ascending or descending order. An algorithm of complexity $O(n^2)$ has been developed for this problem.

As for the second part of the problem, we have to find the longest sorted child of the matrix. The child of a matrix can be obtained by starting from any element in the matrix and proceeding in either of the four directions; left, right, up or down, from that element. The longest sorted child implies that the elements in the child are in increasing or decreasing order and this path should be the longest possible path in the matrix. An algorithm implementing backtracking has been developed for this part whose complexity is non-polynomial.

Using the relevant test cases, we obtain the time complexity of the algorithms for each of best, average and worst cases. Then we plot the time versus n graphs for the above mentioned cases.

II. ALGORITHM DESIGN

The matrix for both the parts can be generated by allocating space using the $malloc()$ function and populating it with random values with the help of $srand()$ function.

A. LONGEST SORTED PARTITION IN A COLUMN

The longest sorted partition in each column can be found by iterating through each cell and keeping track of the current partition's length and the maximum length found so far. Following are the steps:

1) Initialize $maxLen$ and len to one then start traversing each column from the 1^{st} element (0-indexed)
2) Increment len if the next cell is sorted w.r.t. the current cell
3) If at any point the current length is greater than the maximum length found so far, reassign the max length and store the index of the initial cell
4) If the next cell doesn't satisfy the current trend, reset the current length and continue
   Since 'sorted' could either mean ascending or descending, both the cases should be tracked independently. Finally we end up with the length and the starting index of the longest sorted partition in each column.

Algorithm 1 Longest Sorted Partition algorithm

```
for j ← 0 to n do
    for i ← 1 to n do
        if matrix[i − 1][j] ≥ matrix[i][j] then
            len ← len + 1
        else
            len ← 1
        if len > max then
            max ← len
            start ← i − len + 1
```

B. LONGEST SORTED PATH IN THE MATRIX

Finding the longest sorted path in the matrix requires a recursive traversal throughout the
matrix. Basically we need to perform a Depth First Search on the matrix treating it to be a graph. Each cell would be a node and edges would occur between adjacent cells of non-decreasing values. By backtracking, we can find all possible paths in sorted order.

1) Assume we have a routine called $recursiveFind()$ which returns the length of the longest sorted path from a given cell.

2) It first marks the current cell as visited
3) We can implement this routine recursively by calling it for the neighbours (left, right, up, down) of the current cell which are not visited yet and are in non-decreasing order.
4) Unmark the current cell (backtrack)
5) Return the maximum of the path lengths obtained above.

Algorithm 2 Recursive Algorithm

```
procedure RECURSIVEFIND(matrix, n, x, y, len)
    visited[x][y] ← matrix[x][y]
    if len + 1 = maxLen then
        printMatrix(visited)
    l ← len + 1
    r ← len + 1
    u ← len + 1
    d ← len + 1
    if visited[x][y−1] == −1 AND matrix[x][y−1] ≥ matrix[x][y]] then
        l ← recursiveFind(matrix, n, x, y − 1, len + 1)
    if visited[x][y+1] == −1 AND matrix[x][y+1] ≥ matrix[x][y]] then
        r ← recursiveFind(matrix, n, x, y + 1, len + 1)
    if visited[x−1][y] == −1 AND matrix[x−1][y] ≥ matrix[x][y]] then
        u ← recursiveFind(matrix, n, x + 1, y, len + 1)
    if visited[x+1][y] == −1 AND matrix[x+1][y] ≥ matrix[x][y]] then
        d ← recursiveFind(matrix, n, x, y, len + 1)
    visited[x][y] ← −1
    return MAX(l, r, u, d)
```

Now we can use this routine to find all sorted paths.

1) Call $recursiveFind()$ for each cell in the matrix. This will return the length of the longest non-decreasing child of the matrix assuming path starts at the current cell.
2) Keep track of the maximum length found so far and the indices of their corresponding starting points.
3) Finally call the $recursiveFind()$ routine again for each of these starting points but this time print the visited matrix when the traversing path length reaches the maximum value i.e. when the last cell in the longest path is reached.

Algorithm 3 Driver function

```
for i ← 0 to n do
    for j ← 0 to n do
        len ← recursiveFind(matrix, n, i, j, 0)
        init[i][j] ← len
        if len > max then
            max ← len
            start ← i − len + 1
```

III. ANALYSIS

A. LONGEST SORTED PARTITION IN A COLUMN

The proposed algorithm traverses each cell only once while simultaneously searching for ascending and descending sequences. So for a single column of size n, the complexity would be $\theta(n)$. As there are n such columns again in an $n * n$ matrix, the same algorithm would be executed n more times giving us the final complexity as $\theta(n^2)$.

On analysing the time taken by every significant computation step in the algorithm and the frequency of the steps occurrences, the following expression is obtained for the time complexity:
$WorstCase$:

$$t \ \alpha \ 20n^2 - 8n + 1$$

$BestCase$:

$$t \ \alpha \ 15n^2 + 2n + 1$$

Either way, the time complexity is of the order of n^2.

B. LONGEST SORTED PATH IN THE MATRIX

Analysis of the $recursiveFind$ algorithm is quite difficult because the backtracking steps add an exponential component to the complexity.

1) Best Case: The best case for this algorithm would occur when the max path length is least as it would lead to the least number of recursive calls and reduces the backtracking steps. The least path length cannot be 1 as any two neighbours are always sorted. So 2 is the minimum path length and therefore the matrix should be filled with alternate elements. This would give a complexity of order $\Omega(n^2)$ as each call to $recursiveFind$ would return after calling a maximum of 4 times.

2) Worst Case: Each call to the routine produces 4 more calls corresponding to the 4 directions in the worst case and this continues for each cell until are the cells in the matrix are exhausted. This gives a complexity of $O(4^{n^2})$. Moreover this has to repeated for each cell again taking each one as the beginning of the path giving a final complexity of $O(n^2 * 4^{n^2})$.

In practice the recursiveFind() will mostly only call 1 or 2 more times as only the first cell would

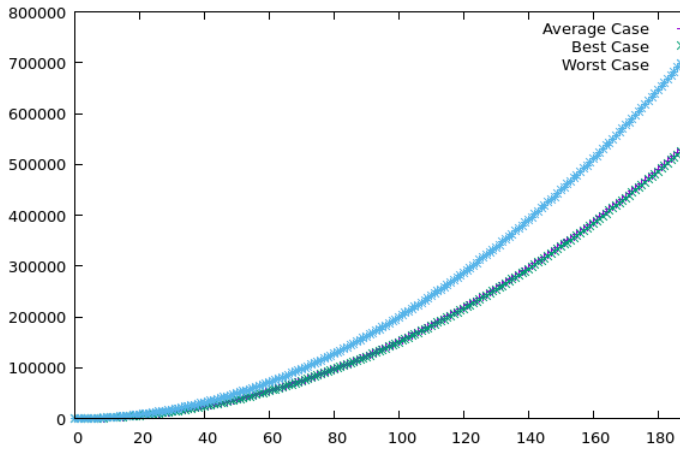

Fig. 1. Time Complexity for Part A

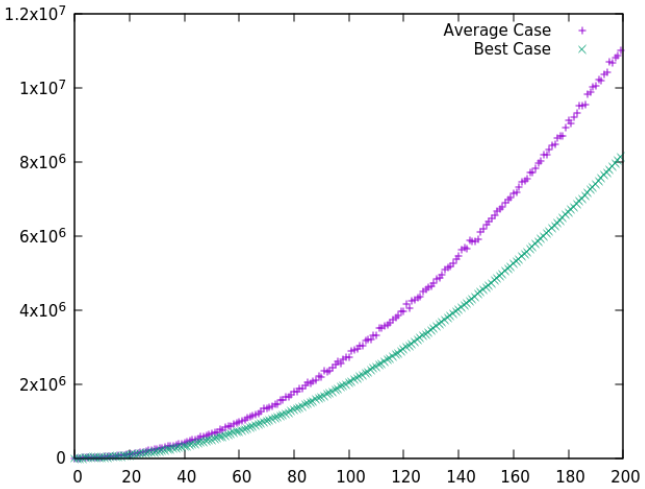

Fig. 2. Time Complexity for Part B

have 4 valid neighbours that too if it is not on the boundary of the matrix and after tracing a part of the path, many cells become visited and hence blocked. Therefore $O(n^2 * 4^{n^2})$ is an over-exaggeration of the worst complexity and the actual order would be of the form $O(n^2 * k^{n^2})$ where $1 < k < 2$.

IV. EXPERIMENTAL STUDY

Following are the experimental findings after profiling the data and plotting the relevant graphs using $gnuplot$.

A. Graphs

1) Fig. 1. gives the best case, worst case and the average case time complexity for the $findPartition$ algorithm.
2) Fig. 2. gives the best case and average case time complexity of the backtracking algorithm vs values of n till 200.
3) Fig. 3. is the worst case time complexity for part B plotted for n = 1 to 6.
4) Fig. 4. is additional findings for the part B. It gives the longest sorted path length vs n for 200 cases.

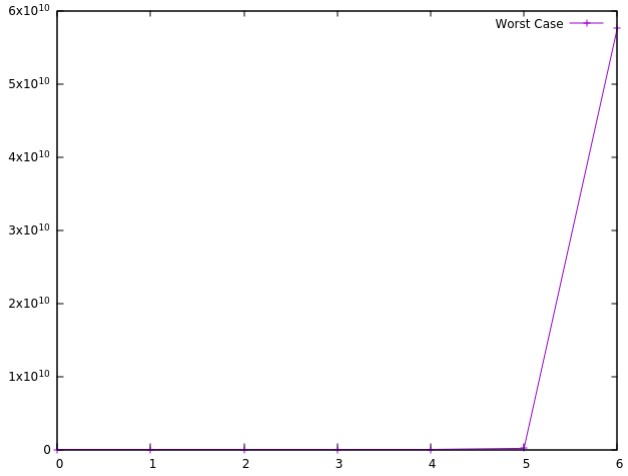

Fig. 3. Worst Case for Part B

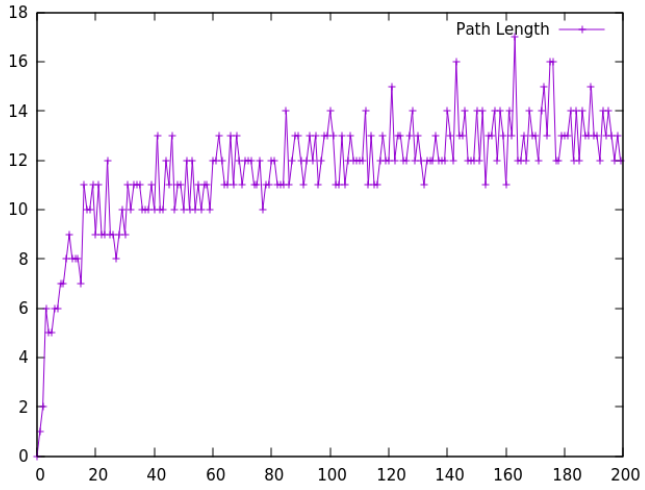

Fig. 4. Longest Path Length vs N

B. Profiling

$$TimeComplexity$$

Size (n)	50	75	100	125	150	175	200
t_{max}	49601	111901	199201	311501	448801	611101	790429
t_{avg}	38216	85601	151646	236631	340216	462591	597819
t_{min}	37601	84526	150201	234626	337801	459726	594414

TABLE I
PART 1 - LONGEST SORTED PARTITION

n	35	70	105	140	175	200
t_{avg}	309086	1311148	2893500	5229024	8279143	10753165
t_{min}	225249	913612	2065024	3679487	5756999	7449203

TABLE II
PART 2 - LONGEST CHILD BEST AND AVERAGE CASES

n	1	2	3	4	5	6
t_{max}	82	1892	34101	1882452	154667081	5.7×10^{10}

TABLE III
PART 2 - LONGEST CHILD WORST CASE

LONGEST SORTED PARTITION IN A COLUMN

It can be observed both from the graph and the table that the best, worst and average cases have almost similar time complexities and the graph resembles

$$y = Ax^2 + Bx + C$$

LONGEST SORTED PATH IN THE MATRIX

The average and the best case plots are almost identical whereas the worst case plot escalates very quickly making it hard to plot for larger values of n. This implies due to the randomness in the average case, it is somewhat close to the best case which is a parabola.

Thus we can safely say that our theoretical calculations match with the experimental study.

V. DISCUSSION

A. Generation of random numbers

In order to fill the $n*n$ matrix with random numbers, we have used the $rand()$, $srand()$ functions and the $< time.h >$ header file.

The $rand()$ function generates a pseudo random number, that is, the same random number is generated every time we compile and run the program. So, we used the $srand()$ function to set the seed for the $rand()$ function.

B. Complexity of the longest sorted path

This problem is likely to belong to the NP class of problems. However the average case is good owing to the random numbers implying the probability to get a matrix close to the best case seems to be relatively greater.

C. Path Length vs N

This is an interesting observation in the problem. As we increase n, the rate of increase in the length of the longest sorted path appears to be decreasing; meaning the larger the matrix, the lesser the chances of getting a longer path.

D. Tracing all possible paths

In the algorithm all the possible paths are being explored while backtracking. So instead of breaking after finding a possibility we modified the algorithm to trace all paths of the maximum length. This number was plotted alongside n but as such no relation was found.

VI. CONCLUSION

In the first module, we have implemented an $\theta(n^2)$ algorithm and for the second module an algorithm using backtracking has been developed whose average case very close to the best case which has a $\Omega(n^2)$ complexity. Graphs have also been plotted and agree with the theoretically calculated complexities.