Group-51- Take a 10 x 10 matrix (n x n in general) filled with Devanagari alphabets. Trace the longest sorted partition:- a) in each column of the matrix b) Trace the longest sorted child of the matrix.

IHM2016005[1] BIM2016004[2] IIM2016006[3] IHM2016501[4] RIT2015050[5]

I. INTRODUCTION AND LITERATURE SURVEY

The Devanagari script is used to write many Indian languages such as Hindi, Konkani, Sanskrit, Marathi and Nepali.In Devanagari, there are 14 vowels and 33 consonants (conjunct consonants included).Given is a n x n matrix filled with Devanagari alphabets we have to trace the longest sorted partition:- a) in each column of the matrix b) Trace the longest sorted child of the matrix. .Each devanagari alphabet has a unique Unicode entity .These entities range from 2309 to 2361 i.e; 52 entities for each alphabet which includes both Vowels and Consonants.We use a special data type i.e $wchar\_t$ for storing these Unicode values. The $wchar\_t$ (wide character)type is intended for storing compiler-defined wide characters, which may be Unicode characters in some compilers.In the first part of the question we have traverse through the columns of the matrix by comparing the two consecutive elements to find the longest sorted partition(column wise).Whereas in the second part of the question we have to traverse through the entire matrix i.e row wise,column wise and also diagonally to find out the longest sorted child of the matrix.

II. ALGORITHM DESIGN

This part of the report can be further divided into the following subsections.

A. Input Format

In first few lines of the code we use $srand()$ function to generate a random number and assign it to the variable n which defines the size of the matrix .We also use this $srand()$ function to generate the Unicode values of the devanagri alphabets i.e by bounding the numbers generated by $srand()$ between 2309 and 2361.We write these alphabets in the $testcases.txt$ file and later read them from the file and store them in a 2D array $a[n][n]$.So now the 2D array a[n][n] acts like a n x n matrix filled with devanagri alphabets.

B. Algorithm

Algorithm 1 Part a

```
1:  procedure LONGEST(a[n][n]) ▷ longest sorted partition
2:      lengthinc ← 1
3:      lengthdec ← 1
4:      maxlength ← −1
5:      index ← 0
6:      for j = 0 to n-1 do
7:          lengthinc ← 1
8:          lengthdec ← 1
9:          maxlength ← −1
10:         for i = 1 to n-1 do
11:             if a[i-1][j] <a[i][j] then
12:                 lengthinc + +
13:                 if maxlength <lengthinc then
14:                     maxlength ← lengthinc
15:                     index ← i − maxlength + 1
16:                 if maxlength <lengthdec then
17:                     maxlength ← lengthdec
18:                     index ← i − maxlength + 1
19:                 lengthdec ← 1
20:             else if a[i][j] <a[i-1][j] then
21:                 lengthdec + +
22:                 if maxlength <lengthinc then
23:                     maxlength ← lengthinc
24:                     index ← i − maxlength + 1
25:                 if maxlength <lengthdec then
26:                     maxlength ← lengthdec
27:                     index ← i − maxlength + 1
28:                 lengthinc ← 1
29:             else
30:                 lengthinc + +
31:                 lengthdec + +
32:                 if maxlength <lengthinc then
33:                     maxlength ← lengthinc
34:                     index ← i − maxlength + 1
35:                 if maxlength <lengthdec then
36:                     maxlength ← lengthdec
37:                     index ← i − maxlength + 1
38:             print(— for j column—)
39:             print(the length of max column is maxlength)
40:             printf(the array is:)
41:             for h =index to maxlength+index do
42:                 print(a[h][j])

1:  procedure MAIN ▷ main function
2:      longest(a)
3:      return0
```

Algorithm 2 Part b

```
1: procedure INMATRIX(x,y)                                   ▷
2:     if  x<n && 0<= x && y <n && 0 <= y then
3:         return1
4: procedure SOLVE(x,y,xx[],yy[],counter)                    ▷
5:     xx[counter] ← x
6:     yy[counter] ← y
7:     if inmatrix(x + 1, y) && !visited[x + 1][y] && arr[x
   ][y] <= arr[x+1][y]  then
8:         visited[x + 1][y] ← 1
9:         solve(x + 1, y, xx, yy, counter + 1)
10:        visited[x + 1][y] ← 0
11:    if inmatrix(x - 1, y) && !visited[x - 1][y] &&
   arr[x][y] <= arr[x-1][y]  then
12:        visited[x − 1][y] ← 1
13:        solve(x − 1, y, xx, yy, counter + 1)
14:        visited[x − 1][y] ← 0
15:    if inmatrix(x , y+1) && !visited[x][y+1] && arr[x
   ][y] <= arr[x][y+1]  then
16:        visited[x][y + 1] ← 1
17:        solve(x, y + 1, xx, yy, counter + 1)
18:        visited[x][y + 1] ← 0
19:    if inmatrix(x , y-1) && !visited[x][y-1] && arr[x
   ][y] <= arr[x][y-1]  then
20:        visited[x][y − 1] ← 1
21:        solve(x, y − 1, xx, yy, counter + 1)
22:        visited[x][y − 1] ← 0
23:    if mxx <counter then
24:        mxx ← counter
25:        for i to mxx do
26:            dest_x[i] ← xx[i]
27:            dest_y[i] ← yy[i]
28: procedure MAIN                                           ▷
29:    for i=0 to n-1 do
30:        for j=0 to n-1 do
31:            visited[i][j] ← 1
32:            solve(i, j, xx, yy, 0)
33:            visited[i][j] ← 0
34:    print(mxx+1)
35:    for i=0 to mxx do
36:        print(dest_x[i],dest_y[i])
37:    return0
```

C. Output Format

The part **a** of the question checks the given 2D array column wise to find the longest sorted partition and outputs the length of the longest sorted partition and the unicode values of the devanagari alphabets present in it.

The part **b** of the question traverses through the entire matrix to find the longest sorted child by using back tracking and outputs the length and the the row and column index of it.

III. ANALYSIS

A. Analysis of Part a

The 'longest' function in part a of the given question calculates the longest sorted partition and also traces its path.And in this function while entering into the for loop we increment the number of instructions by 3 ,1 instruction for condition and 2 for updation and we increment the number of instructions by 1 during the assigning.We are also assuming that each instruction takes unit time.

And for fetching an instruction from an 2D array we increment the number of instructions by 1.So while comparing two elements in a 2D array we are increment the number of instructions by 2.And each arithmetic operation is considered as one instruction.

1) Best Case Analysis - (n=1): When we consider a 1x1 matrix ,it is already sorted .So,the column of this matrix ia already sorted.So in total it takes 6 instructions.

$$time_{best(n=1)} \propto 6 \text{ instructions.}$$

$$O(1) \text{ - A Constant Time Computation}$$

2) Best Case Analysis - (1 <n): When the matrix is of size greater than 1 we will have to traverse through the entire matrix column wise in order to obtain the desired result.

$$time_{best(n>1)} \propto 24n^2 + k \text{ instructions.(k-constant)}$$
$$=> time_{best(n>1)} \propto n^2$$

$$=> O(n^2) \text{ - A Quadratic Time Computation}$$

3) Worst Case Analysis : As discussed above in the worst case senario also we will have to traverse through the entire matrix column wise in order to obtain the desired result.So,the worst case time complexity is equal to best case time complexity.

$$time_{worst} \propto 24n^2 + k \text{ instructions.(k-constant)}$$
$$=> time_{worst} \propto n^2$$

$$=> O(n^2) \text{ - A Quadratic Time Computation}$$

$$=> t_{best} = t_{worst}$$

4) Average Case Analysis : We know that

$$time_{best} <= t_{average} <= t_{worst}$$

But from above

$t_{best} = t_{worst}$

So

$=> t_{best} = t_{avg} = t_{worst}$

B. Analysis of Part b

The solve() function traces the longest sorted child of matrix by the concept of backtracking. Backtracking helps in solving an overall issue by finding a solution to the first sub-problem and then recursively attempting to resolve other sub-problems based on the solution of the first issue. If the current issue cannot be resolved, the step is backtracked and the next possible solution is applied to previous steps, and then proceeds further. In fact, one of the key things in backtracking is recursion. It is also considered as a method of exhaustive search using divide and conquer. A backtracking algorithm ends when there are no more solutions to the first sub-problem

The inmatrix() function checks whether the co-ordinates of the 2D matrix are with in the boundary of matrix.if it is with in the boundary it returns true are else it returns false.

1) Best Case Analysis: Best case comes when there are only two elements in the matrix and they alternate after each position. In this way when we start from any element then maximum sorted partition we get is of length 2 i.e. the element itself and its immediate neighbour. So loop runs only once and breaks.

$time \propto 18n^2 + k$ instructions.(k-constant)

$=> time \propto n^2$

$=> O(n^2)$ - A Quadratic Time Computation

2) Worst Case Analysis: Worst case occurs when all the elements in the matrix are equal.In this we start from an element then maximum sorted partition we get is square of size of matrix. Let us consider that we are starting from a particular cell we now need to proceed in all the 4 possible directions and in each of this direction the processes continues until all the cells in the matrix are covered.This results in $O(4^{n^2})$.And there are n^2 squares the total complexity is $O(n^2 * 4^{n^2})$.

$time \propto n^2 * 4^{n^2}$ instructions

$=> time \propto n^2 * 4^{n^2}$

$=> O(n^2 * 4^{n^2})$

3) Average Case Analysis : We know that

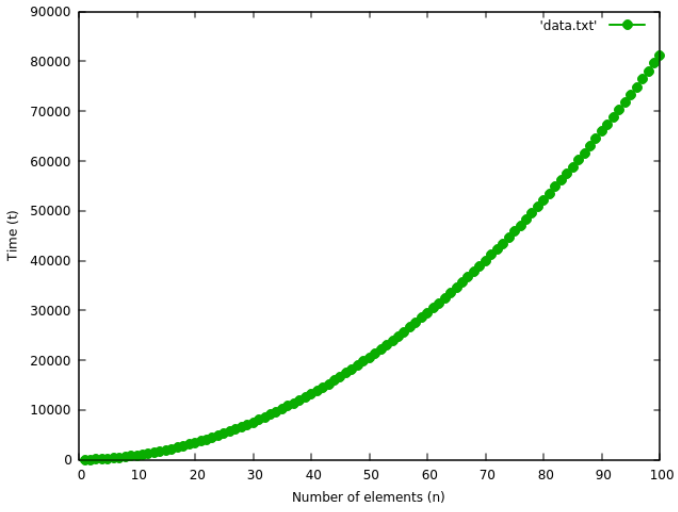
$$time_{best} <= time_{average} <= time_{worst}$$

Average case is variable as it is based on matrix with random inputs so it differs based on the input matrix taken in each and every case.So in order to get the Average Case Analysis we fill the matrices of different sizes with random elements and analyse it.And $time_{avg}$ always lies between $time_{best}$ and $time_{worst}$ .Further details regarding Average Case Analysis are mentioned below in the Part B of the Experimental Study and is explained with the help of graphs.

IV. EXPERIMENTAL STUDY

A. PART A

1) Graphs-TimeComplexity: We have seen that $t_{best} = t_{avg} = t_{worst}$ So,the graphs for best case,average,worst case time complexity remain the same.



2) Profiling:

$NumberOfComputations$

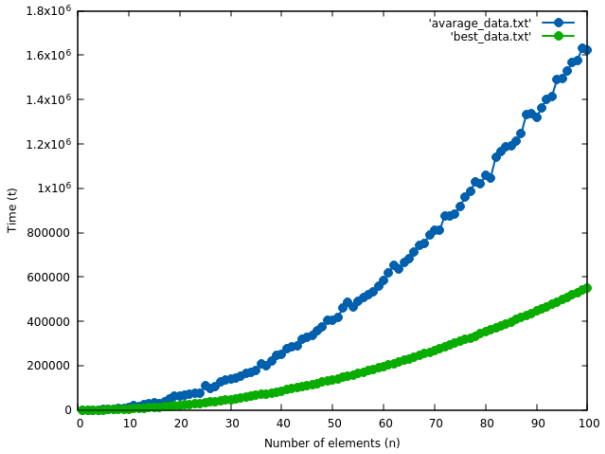
n	t_{avg}	t_{best}	t_{worst}
1	6	6	6
5	217	217	217
10	864	864	864
20	3382	3382	3382
50	20521	20521	20521
70	39972	39972	39972
80	52120	52120	52120
100	81237	81237	81237

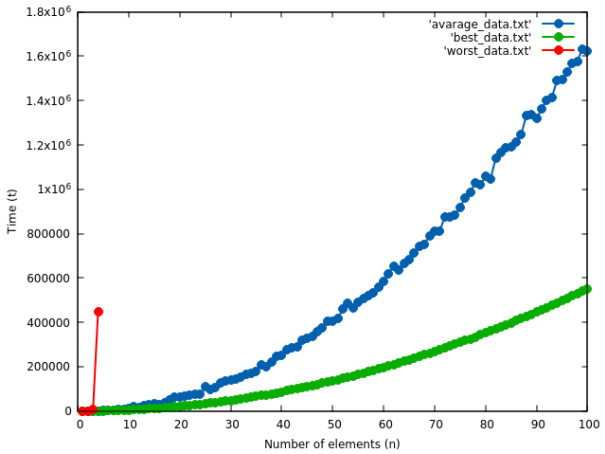
- We can seen that $t_{best} = t_{avg} = t_{worst}$

- We can also see that time increases as the value of n increases and $time_{worst} \propto n^2$,$time_{avg} \propto n^2$,$time_{best} \propto n^2$.

B. PART B

1) Graphs-TimeComplexity: We have seen that $t_{best} < t_{avg} < t_{worst}$ So,the graphs for best case,average,worst case time complexity are as follows.





2) Profiling:

$$NumberOfComputations$$

n	t_{avg}	t_{best}
$= 1$	0	0
5	5016	592
10	13714	4816
20	61908	18720
50	405012	130560
70	790212	268816
80	1018448	352016
100	1621148	552016

n	t_{worst}
$= 1$	85
2	1912
3	35796
4	1894878
5	154832948

- We can seen that $t_{best} < t_{avg} < t_{worst}$
- We can also see that time increases as the value of n increases and $time_{worst} \propto n^2 * 4^{n^2}$,$time_{avg} \propto n^2$,$time_{best} \propto n^2$.

V. DISCUSSION

A. Understanding the Use Of File Handling and generation of random numbers

In this context the file handling is used to generate the random test cases through which analysis and experimentation of algorithm becomes easier for handling different test cases. We have used $srand()$ and $rand()$ to generate the random values to fill the matrix with the help of $< time.h >$ header file. In part B of question, it was not possible to extend value of n beyond 5. As it resulted in very high complexity . Plotting was done till n=4, as for n=5 it diminished average and best case plots.

B. Complexity of the longest sorted path

The worst case is very bad for this algorithm and the problem is likely to belong to the NP(Non Deterministic) class of problems. However the average case is good owing to the random devanagari alphabets implying the probability to get a matrix close to the best case seems to be relatively greater

C. Observation on Path length Vs Size of Matrix

As the n(size of matrix) is increasing the rate of increase in the length of the longest sorted part is appears to be decreasing.It suggest that the chances of getting longest sorted path is lesser.

VI. CONCLUSION

The main module of the algorithm design is to find the longest sorted partition in any given matrix.

We have made use of a special type of datatype (wide character) wchar_t to store the devanagri alphabets.In this solution we have bounded the size of the matrix between 1 and 100 as these values are sufficient enough for us to understand and analyze the result and draw conclusions.

REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[2] Introduction to Algorithms English By Thomas H. Cormen , By Charles E. Leiserson , Ronald L. Rivest ,Clifford Stein(3rd edition)