MATRIX SORTING

Swaraj Bhosle (IIT2019024)
Ritesh Raj (IIT2019025)
Utkarsh Garg(IIT2019026)

4th Semester B.Tech , Department of Information Technology
IIIT, Allahabad, India

Abstract: In this paper, we have devised an algorithm to sort a given matrix such that all the rows, columns and diagonals get sorted. In this we have used 2D MATRIX, analysed the worst case and average case time complexities with respect to input data. We have also calculated the time taken by the function to execute.

I. INTRODUCTION

We are given a matrix with random entries and then we have to output a matrix which is sorted row wise
, column wise and diagonally, we can do so by taking a 1D array and coping all the entries of the matrix to 1D array, then we sort the array using merge sort.
Now we can just copy back the elements of the sorted array to matrix. This algorithm works as when we put back the sorted array and put back the elements row wise, all the elements in particular row are sorted and afterwards when we fill the next row all the elements will be greater than the previous row, this way we obtain a matrix which is sorted row wise, column wise and diagonally.
For creating a new algorithm from scratch, we look into the many crucial aspects of algorithm designing and analysis.The algorithm created is tested for different sample test cases for time complexity, space complexity and accuracy. In our case we tend to focus on establishing a rule or a relationship between the time and input data.

II. ALGORITHM DESCRIPTION

The algorithm requires to solve this problem is merge sort. This algorithm takes 2D matrix of size ‘m’ and ‘n’ as input, we assume initial dimensions as 100*100.
The method consist of mainly four stages:-

Stage 1: In stage one, we take the random value of elements using random function., store the elements of the 2D array in 1D array of size 100000.

Stage 2: In this stage merge sort algorithm is applied on 1D array for sorting the array.

Stage 3: In stage three traverse the 1D array and store all the elements of 1D array in to

2D array, the matrix so obtained will be sorted row wise, column wise and diagonally

Stage 4 : Print the resultant 2D matrix.

Advantage of using merge sort:-

1) It is quicker for larger lists because unlike insertion and bubble sort it doesn't go through the whole list several times.
2) It has a consistent running time, carries out different bits with similar times in a stage.

III. ALGORITHM AND ANALYSIS

Input: A 100x100 matrix with random entry of elements using rand() function..

Output result: The sorted matrix which is sorted row wise, column wise and diagonally is the required output.

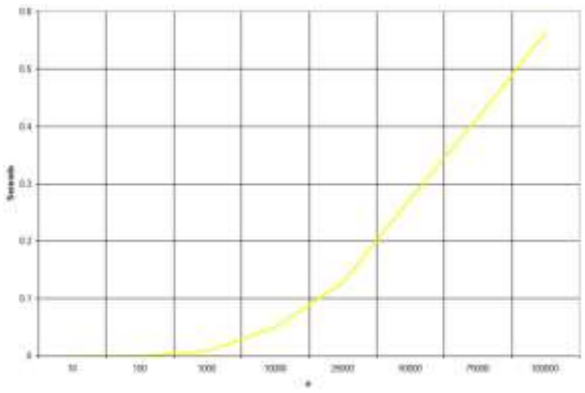
Pseudocode:

```
main() {
// create the matrix with given entries
        for i in 0 to 100
                for j in 0 to 100
                        input matrix[i][j]

// create an array of size m*n
        int arr[10000];

// fill the array with entries of the matrix.
        int index=0;
        for i in 0 to 100
                for j in 0 to 100
                        arr[index] = matrix[i][j]
                        index++

// sort the array using merge sort
        mergeSort(arr, 0, 100 - 1);

// fill up the matrix row by row using the array
        index=0;
        for (int i = 0; i < 100; i++) {
                for (int j = 0; j < 100; j++) {
                        matrix[i][j] = arr[index++];
                        printf("%d ", matrix[i][j]);
                }
                printf("\n");
        }
        for i in 0 to 100
                for j in 0 to 100
                        matrix[i][j]=arr[index]
                        index++

// print the matrix
        for i in 0 to 100
                for j in 0 to 100
                        print(matrix[i][j]+' ')
                print('\n')
}
```

```
Initial matrix is :
41 17 34 0 19
24 28 8 12 14
5 45 31 27 11
41 45 42 27 36
41 4 2 3 42

Sorted matrix is :
 0 2 3 4 5
8 11 12 14 17
19 24 27 27 28
31 34 36 41 41
41 42 42 45 45
```

IV. TIME CALCULATION

VALUES	Execution Time(ms)
5*5	0.31939
10*10	0.32023
100*100	0.67831
1000*1000	3.49021



V. TIME COMPLEXITY

If the matrix is of size n*m, N is the total number of cells.

- **N=n*m**

We only traverse the matrix and make the 1D array using the matrix which will result in complexity of O(N).

The time complexity will be affected due to the merge sort in sorting the 1D array and the total time complexity will become O(N*log(N)) in all the cases ie. best case, worst case and average.

Best case : O(N) + O(N*log(N)) + O(N) => O(N*log(N))

Average case : O(N) + O(N*log(N)) + O(N) => O(N*log(N))

Worst case : O(N) + O(N*log(N)) + O(N) => O(N*log(N))

BEST	AVERAGE	WORST CASE
N*log(N)	N*log(N)	N*log(N)

VI. SPACE COMPLEXITY

We make an extra 1D array to store and sort the entries of the matrix so the extra space of n*m is required.

Space Complexity: O(n*m)

VII. CONCLUSION

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list in logarithmic time.

This algorithm works as when we put back the sorted array and put back the elements row wise, all the elements in particular row are sorted and afterwards when we fill the next row all the elements will be greater than the previous row, this way we obtain a matrix which is sorted row wise, column wise and diagonally.

VIII. REFERENCES

1. https://www.geeksforgeeks.org/merge-sort/