DAA432C Assignment-01

Vikram Singh
IIT2019213
iit2019213@iiita.ac.in

Akshat Agrawal
IIT2019214
iit2019214@iiita.ac.in

Ravutla Ruthvik
IIT2019215
iit2019215@iiita.ac.in

Abstract**—** In this report, we design an algorithm to solve the problem that given an n x n matrix, where every row and column is sorted in increasing order. Given a key, how to decide whether this key is in the matrix.

Keywords**—** Divide and conquer approach, 2-D matrix, time complexity,space complexity

I. INTRODUCTION

This paper discusses an algorithm that is designed to find the given key element in a 2-D matrix, where every row and column is already sorted in increasing order, using divide and conquer approach.

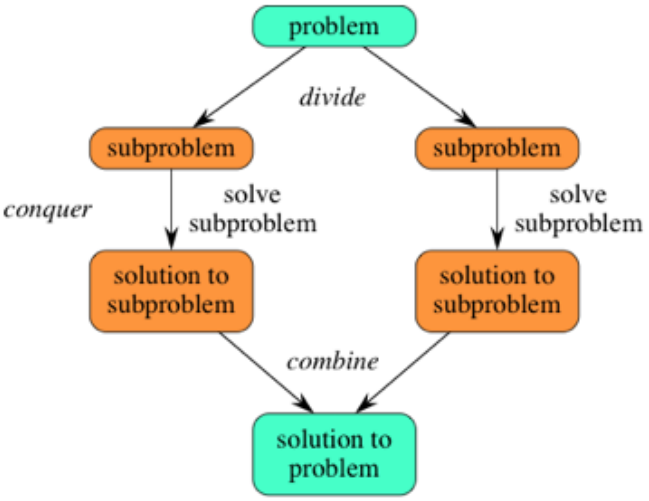
Divide and Conquer is an algorithm design paradigm in which we recursively break down a problem into two or more sub-problems of the same or related type, until these subproblems become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. The divide-and-conquer paradigm is often used to find an optimal solution of a problem in which we recursively simplify the problem by decreasing the constraints of a subproblem and selecting / dis-selecting a given subproblem based on some conditions.

The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g., the Karatsuba algorithm),finding the closest pair of points.

The approach of divide and conquer generally follows three important aspects:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

Let us have a pictorial look of the image which makes the concept of Divide and Conquer Paradigm self explanatory.



Advantages of Divide and Conquer

The Divide and Conquer technique generally has the following mentioned advantages over Brute Force Algorithms.

Time Efficiency: This approach generally reduces the running time of the algorithm because the running time of an algorithm based on Brute Force is in general high order in nature. But when we are using the approach based on Divide and Conquer, this running time generally decreases and becomes logarithmic in nature. This is because the approach of Divide and Conquer keeps on dividing which makes it logarithmic in nature.

Space Efficiency: As no need to mention the algorithm, but still, the space complexity also reduces up to a much extent due to the nature of dividing and then merging up. Also it is worthy to note that some divide and conquer algorithms which are based on the recursive algorithms generally tend to occupy more space in the internal allotted stack of the process in RAM(Random Access Memory).

II. ALGORITHM DESIGN

The given problem can be solved using divide and conquer approach which is similar to binary search. Basically, we divide the given problem into smaller sub-problems and appropriately combine their solutions to get the solution to the main problem.

Approach: The idea is to find the middle element of the 2-D matrix and depending on whether that element is smaller or greater than the key element , search the key element in the following sub-matrices.

Algorithm

1) First, we find the middle element of the matrix.

2) Then, if the value of the middle element is the same as the key value, we simply return true which means the element is found.

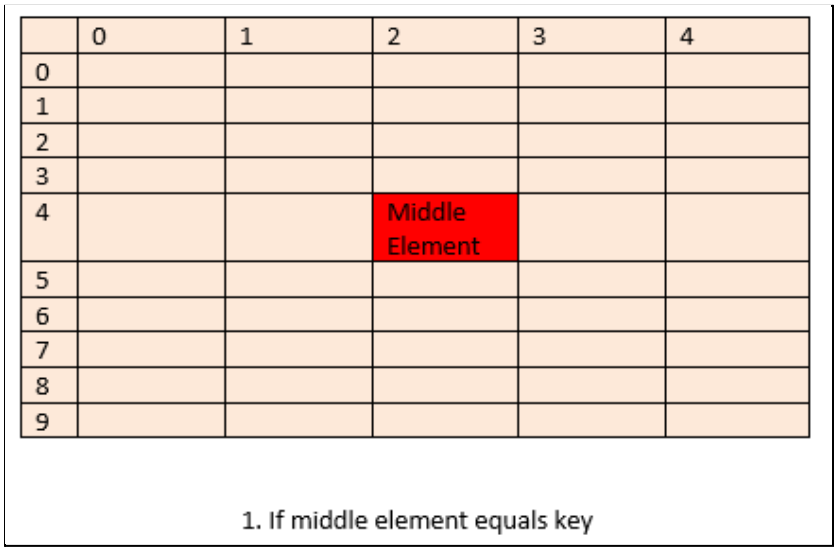
3) If the value of the middle element is lesser than key value then:
a) We search the submatrix on lower side of middle element
b) Search submatrix on right hand side.of middle element

4) If the value of the middle element is greater than key value then:
a) search vertical submatrix on left side of middle element
b) search the submatrix on the right hand side.

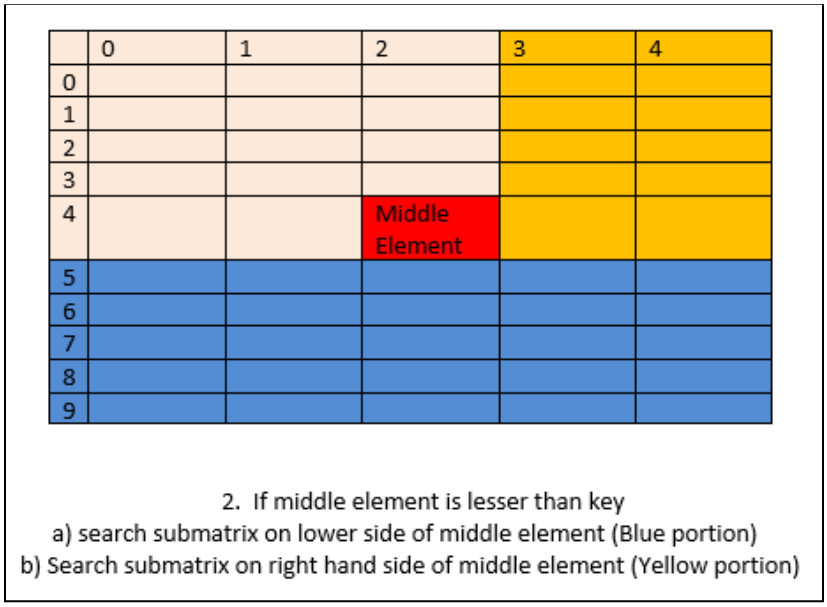
5) We continue to do this recursively, until only one element is left to be checked in the recursive call. If the value that element equals the key value ,we return true else we return false which means that the given key value is not present in the matrix.

Pictorial representation:

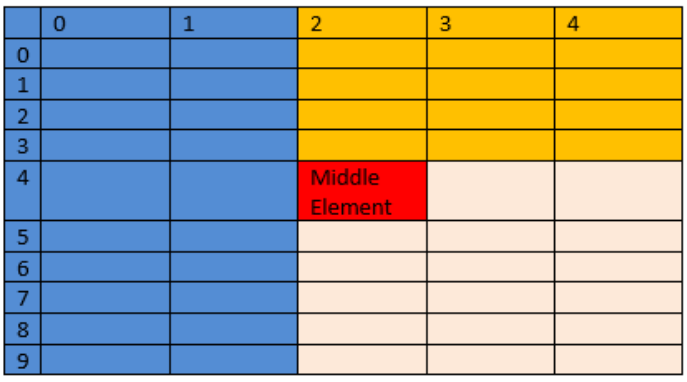
For understanding the above algorithm in a better way, below is the pictorial representation of the approach:



1. If middle element equals key



2. If middle element is lesser than key
a) search submatrix on lower side of middle element (Blue portion)
b) Search submatrix on right hand side of middle element (Yellow portion)



3. If middle element is greater than key
a) search vertical submatrix on left side of middle element (Blue portion)
b) search submatrix on right hand side. (Yellow portion)

III. PSEUDO CODE

Below is the pseudo code for the **search** function which contains the algorithm described above.

As this function is a recursive function it has the following base conditions which needs to be checked every time the function is called:

- If the given row column pair does not contain any value i.e. number of elements is 0, then return false.
- If the given row column pair contains a single value i.e. number of elements is 1, then return true if the element is equal to key else return false.
- If the biggest element in the row column pair is less than the key element or the smallest element in the row column pair is greater than the key element, then return false.

After checking the base conditions, we check if the middle element is equal to key. If not, we further check the submatrices as described above.

```
search
(mat,row_start,row_end,column_start,column_end,key)
{
if (row_start > row_end || column_start > column_end)
return false

if (row_end == row_start && column_end == column_start)
if (mat[row_end][column_end] == key)
return true
else return false

if (mat[row_start][column_start] > key ||
mat[row_end][column_end] < key)
return false
// Base conditions

mid_row = (row_end + row_start) / 2;
mid_column = (column_end + column_start) / 2;

if (key == mat[mid_row][mid_column])
return true

else if (key > mat[mid_row][mid_column])
return search(mat, row_start, mid_row, mid_column + 1, column_end, key) || search(mat, mid_row + 1, row_end, column_start, column_end, key);

else
return search(mat, row_start, mid_row - 1, mid_column, column_end, key) || search(mat, row_start, row_end, column_start, mid_column - 1, key);
}
```

IV. ALGORITHM ANALYSIS

For the above approach based on divide and conquer, the algorithm can be seen as recurring for 3 matrices of size n/2 x n/2.

Time Complexity:

Following is recurrence for time complexity

$$T(n) = 3T(n/2) + O(1)$$

The time complexity for such an recurrence relation can we found out using the

Masters Algorithm:

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.

Then, we have-
a = 3
b = 2
k = 0
p = 0

Now, a = 3 and $b^k = 2^0 = 1$.
Clearly, $a > b^k$.
So, we follow case-01 which is If $a > b^k$, then $T(n) = O(n^{\log_b(a)})$

So, we have-
$T(n) = O((n^{\log_b(a)})$
$T(n) = O\ (n^{\log_2(3)})$
$T(n) = O\ (n^{1.58})$

The solution of recurrence is **$O(n^{1.58})$** using Master Method.
But the actual implementation calls for one submatrix of size n x n/2 or n/2 x n, and another submatrix of size n/2 x n/2.

Time Complexity: **$O(n^{1.58})$**

Space Complexity:

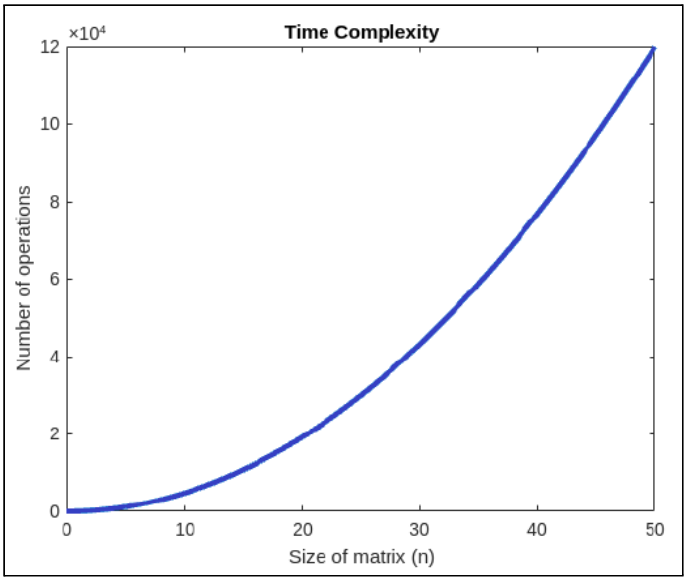
Since we input a 2-D matrix of size NXN, it will require **$O(n^2)$ space.**

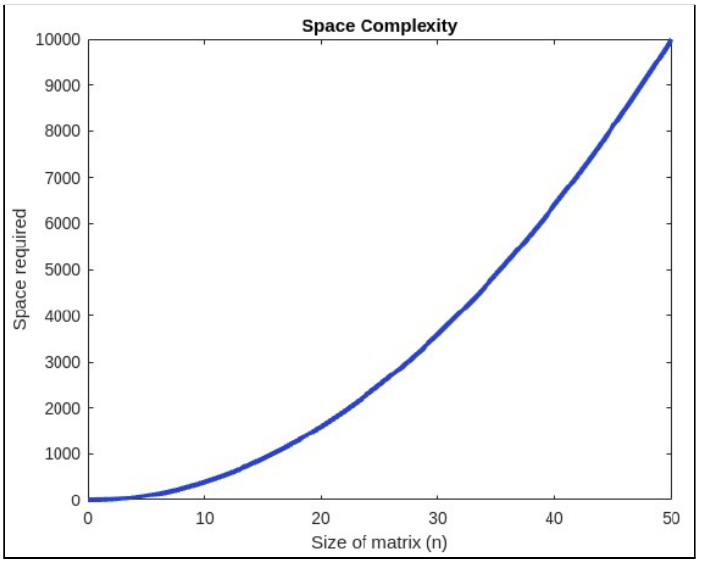
Also as the number of variables used in every function call are fixed, if we try to calculate the total number of function calls made (using the method used for calculating time complexity), we get **$O(n^{1.58})$** space complexity.
Thus the overall space complexity of the algorithm is = $O(n^{1.58}) + O(n^2) \approx$ **$O(n^2)$.**

Space Complexity: **$O(n^2)$**

V. PROFILING

Apriori analysis is performed prior to running it on a specific system. This analysis is a stage where a function is defined using some theoretical model. Hence, we determine the time and space complexity of an algorithm by just looking at the algorithm rather than running it on a particular system with a different memory,processor, and compiler.So,as we discussed under the heading complexity analysis we arrived at the conclusion that the average time complexity is $O(n^{1.58})$.
Also the space complexity that we arrived at was $O(n^2)$.





Posteriori analysis of an algorithm means we perform analysis of an algorithm only after running it on a system. It directly depends on the system and changes from system to system. So for the A Posteriori analysis of the algorithm, we have run our code on the compiler and get values of the time by specifying the value of n and as a result we got different values of the time.

S. No	n	Time taken (in microseconds)
1	5	3
2	10	3
3	50	4
4	100	6
5	500	12
6	1000	13

Unfortunately, in the compilers on our system, declaration of a 2-D array of size more than $n=10^3$ was not possible. To plot a meaningful graph, large values of n are required. Hence, plotting of the graph was not possible..

VI. APPLICATIONS

Divide and Conquer is a wide variety of algorithmic paradigm which believes in the strategy of dividing the task and then applying an internal mechanism or algorithm in order to get the required answer. This programming paradigm has several predefined algorithms which work on the basis of Divide and Conquer approach.
Some of the applications of the Divide and Conquer Approach are:

1. Binary Search: It is a powerful programming algorithm based on the divide and conquer paradigm which works very efficiently and gives answers in generally time cost, which is in general logarithmic in nature.

2. Segment Trees: It is another data structure which uses three standard operations:insert,query and update. All these three operations are based on divide and conquer because for example for building a segment tree, we first need to create in a recursive bottom up divide and conquer based technique.

3. Strassen's Algorithm: It is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops. Strassen's algorithm multiplies two matrices in efficient time.

4. Quick Sort: It is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.

5. Merge Sort: is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

VII. CONCLUSION

We have used a divide and conquer approach with the motive to reduce the time and space complexity of the algorithm. But we find that although this algorithm is better than the

standard brute force algorithm which has a time complexity of $O(n^2)$, upon further analysis we find that this problem could also have been solved in O(n) time complexity. We shall not elaborate that method in this research paper, as this research paper focuses on solving the problem through divide and conquer method.
Hence, we conclude that in this case although the divide and conquer method is better than the brute force approach, it is not the best method for this particular problem.

VIII. ACKNOWLEDGEMENT

We are very much grateful to our Course instructor Dr Rahul Kala and our mentor, Tejaswi Bisen, who have provided the great opportunity to do this wonderful work on the subject of Data Structure and Algorithm Analysis specifically on the programming paradigm of Divide and Conquer.

REFERENCES

[1] Introduction to Algorithms / Thomas H. Cormen . . . [et al.]. - 3rd edition.
[2] The Design and Analysis of Algorithms (Pearson) by A V Aho, J E Hopcroft, and J D Ullman
[3] Algorithm Design (Pearson) by J Kleinberg, and E Tard
[4]https://www.geeksforgeeks.org/auxiliary-space-recursive-functions/#:~:text=For%20example%20if%20we%20need,space%20required%20by%20a%20program
[5]https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/

APPENDIX

The full code for the above problem is given below:

```
#include <bits/stdc++.h>
using namespace std;
#define ROW 4
#define COL 4

bool search(int mat[ROW][COL], int row_start, int row_end, int
column_start, int column_end, int key)
{
    if (row_start > row_end || column_start > column_end)
        return false;
    if (row_end == row_start && column_end == column_start)
    {
        if (mat[row_end][column_end] == key)
        {
            cout << "Element found at index {" << row_end << "," <<
column_end << "}";
            return true;
        }

        else
            return false;
    }

    if (mat[row_start][column_start] > key || mat[row_end][column_end] <
key)
        return false;

    int mid_row = (row_end + row_start) / 2;
    int mid_column = (column_end + column_start) / 2;

    if (key == mat[mid_row][mid_column])
    {
        cout << "Element found at index {" << mid_row << "," << mid_column
<< "}";
        return true;
```

```
    }
    else if (key > mat[mid_row][mid_column])
    {
        return search(mat, row_start, mid_row, mid_column + 1, column_end,
key) || search(mat, mid_row + 1, row_end, column_start, column_end, key);
    }
    return search(mat, row_start, mid_row - 1, mid_column, column_end,
key) || search(mat, row_start, row_end, column_start, mid_column - 1,
key);
}

int main()
{
    srand(time(0));
    int mat[ROW][COL];
    int x = 0;
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            x += (rand() % 10) + 1;
            mat[i][j] = x;
        }
    }
    cout << "Below is the randomly generated 4X4 matrix:-" << endl;
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
            cout << mat[i][j] << " ";
        cout << endl;
    }
    int key;
    cout << "Enter the element to be found:" << endl;
    cin >> key;

    if (search(mat, 0, ROW - 1, 0, COL - 1, key) == 0)
        cout << "Not found";
}
```