DAA ASSIGNMENT-2
GROUP-3
Finding a particular key without sorting in a randomly generated square matrix of size n x n

Sai Charan Teja IIT2016039 | Mantek Singh IIT2016007 | Shruti Reddy IIT2016019 | Rakesh Lakra IIT2016052

***Abstract*—Here, we have tried to design an algorithm to locate a key in a randomly generated square matrix of size n x n. An $O(n^2)$, $\Omega(n^2)$ and $\theta(n^2)$ algorithm using linear search concept has been designed to solve this problem.**

***Index Terms*—Linear Search, Finding Key, time complexity, graphs**

I. Introduction and Literature Survey

Linear search or sequential search is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched. The approach is to traverse through each row of the matrix and search for the desired key. The algorithm also stores up number of instances of the key appearing in the matrix.

II. Algorithm Design

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 locate a key in an array of length N.

1. Consider an array of size N filled with random integers.
2.Let count store the number of instances on which the element in the array matched that of the key we are locating
3. If the element in the array matches the key being located, increment count by 1 and print out the coordinates of the location of the array element
4. Now we are extending the above algorithm for each row of the matrix so that all elements of the matrix can be traversed.
5. In each row only the column index gets incremented every time and while moving from one row to the next, the row index gets incremented by 1.

The pseudo-code for the above explained algorithm is as follows:

Algorithm 1 Locating a key

```
1:  procedure FIND(mat, n, key)
2:      count ← 0                                   ▷ t+1
3:      ans
4:      k ← 0                                       ▷ t+1
5:      for i = 0 to n-1 do                         ▷ t+2*n
6:          for j = 0 to n-1 do                     ▷ t+2*(n)
7:              if mat[i][j]=key then               ▷ t+3
8:                  ans[k] ← (i, j)                 ▷ t+2
9:                  k ← k + 1                       ▷ t+1
10:                 count ← count + 1               ▷ t+2
```

III. Time Complexity Analysis

The time complexity for the above mentioned algorithm $O(n^2)$, because we are traversing through the matrix only once.

Here the worst case complexity and best case complexity both are proportional to n^2 because in both cases we are traversing through the whole matrix only once.In the best and worst cases only the coefficient of n^2 varies.

Worst Case :In this algorithm the worst case is when every element in the matrix is equal to the given key, so that every time we find the index in which the element is present and every time the counter is incremented.

Concretely the expression obtained for time complexity is $6 * n^2 + 4 * n + 3$ which serves as the upper bound i.e the worst case time complexity.

Best Case :Similarly the worst case is when the given key is not present in the matrix, so we never find the index as the element is not present and the counter never gets incremented.

Concretely the expression obtained for time complexity is $4 * n^2 + 4 * n + 3$ which serves as the lower bound i.e the best case time complexity.

The expression for time complexity in the average case is bounded below by the expression for time complexity obtained in the best case and bounded above by the expression for time complexity obtained in the worst case.

$$g(x) < f(x) < h(x)$$

where 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 case is the same. This problem does not have worst case and best case time complexities of differing orders as both are of the the order of n^2.

IV. Experimental study

We have found the time complexity of the algorithm by calculating it in each step. Now to will verify it with the experimental results.

We have plotted T VS N graph where is T is the total number of computations performed for a given N. The T and N values for finding key in matrix are mentioned below in table and N vs T graph for them are plotted accordingly.

Table I
N vs T

n	t_{best}	$t_{average}$	t_{worst}
1	11	11	13
10	443	451	643
20	1683	1735	2483
30	3723	3795	5523
40	6563	6751	9763
50	10203	10477	15203
60	14643	15017	21843
70	19883	20381	29683
80	25923	26569	38723
90	32763	33471	48963
100	40403	41387	60403

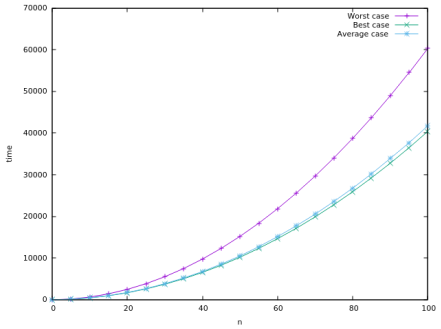

Figure 1. T(y-axis) VS N(x-axis)

We can verify that the best and worst case values in the tables satisfy the equations mentioned in the complexity analysis. So we can verify our complexity analysis for this algorithm is correct.

V. Discussions

Here we have searched for the key by searching in each row one by one ,we can also search for the key by searching in each column one by one. In either way the complexity will be same

From time complexity analysis and experimental study we can see the overall complexity of the algorithm is always proportional to n^2 i.e always $O(n^2)$.

So we can see that the curve is a semi parabola with axis of symmetry as Y-axis , but the co-efficient of the n^2 term varies in best and the worst cases.

we can observe that the curve for random case is very much closer to the best case than the worst case. This is because when we consider a random matrix the probability of repetition of same number many times is very less and to get a key which is repeated more number of times in the matrix is very less.

Here we had to find only one key for each query and also every position in which the key is present, so we performed brute force.

If we had to find multiple number of keys for only one matrix then we have different ways to solve that.

we can solve it by sorting the given matrix along with their position and we can find the key by performing binary search in the sorted sequence. Then the complexity for sorting will be proportional to $(n^2)log(n^2)$ and for binary search $m^(log(n^2))$ where m is the number of keys.

If we had to find only one position of that element in the matrix them we can perform hashing but need a big hash table that can fit the maximum given element then even for m number of keys the complexity will be proportional to n^2, else we want every position of a given element we can perform it by combining the concepts of hashing and linked lists.

We can decrease the time for given algorithm by given each row to a different thread for finding the key in that particular row so that the time complexity will decrease to $O(n)$. But for this the system should support mutithreading.

VI. Conclusion

Here, we have tried to design an efficient algorithm that locates a key in an n x n randomly generated matrix of integers. A simple approach of row-wise traversing accros each element of the matrix and checking if the element's value is equal to that of the key has been used. The details of implementation and pseudo code along with relevant calculations of time complexities and experimental details is done above to the best of our abilities. Hence, it can be concluded that the given question can be solved in $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$. Moreover, all the possible cases have been discussed for the formation of as efficient algorithm as possible.