A04G17Q17: Searching an element from matrix using merge sort

IIT2018049-Harsh Kochar, IIT2018050-Avneesh Gautam, IIT2018051-Avishek

*IV Semester B.Tech, Information Technology,*

*Indian Institute of Information Technology, Allahabad, India*

***Abstract: In this paper, we have devised an algorithm to search an element from an m*×*n sorted matrix which is sorted using merge sort.***

***We have discussed the time complexity of the algorithm by both Apriori and Aposteriori analysis.***

***Key terms – Matrix, search, sort***

I. INTRODUCTION

A matrix is a rectangular array of [numbers](https://en.wikipedia.org/wiki/Number) or other mathematical objects for which operations such as [addition](https://en.wikipedia.org/wiki/Matrix_(mathematics)#Basic_operations) and multiplication are defined. The size of a matrix is defined by the number of rows and columns that it contains. A matrix with *m* rows and *n* columns is called an *m*×*n* matrix.

Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides the input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves.

# Searching is the method to find an element in a sorted matrix. Given a sorted matrix(m\*n) and an element. We need to find the position of that element in the matrix if it is present. Matrix is sorted in a way such that all elements in a row are sorted in increasing order and for row ‘i’, where 1 <= i <= m-1, first element of row 'i' is greater than or equal to the last element of row 'i-1'.

The idea is to create a new array of size *m*×*n* and store the elements of the matrix in a newly created arrayand then sort the created array using merge sort and finally store the elements of the sorted array in the matrix. After sorting the matrix, we will search the required element using binary search(by assuming the matrix to be 1-D array) from the sorted matrix.

II. ALGORITHM DESCRIPTION

Steps to design this algorithm are-

**Step 1**- *m*×*n* integers are randomly generated using rand( ) function and stored in a matrix.

**Step 2**- Elements of the matrix are sorted in an array of size *m*×*n*.

**Step 3**- The created array is sorted using merge sort.

**Step 4**- Store the elements of the sorted array in the matrix again.

**Step 5**- Search the required element from the sorted matrix using binary search assuming the matrix to be 1-D array.

III. ILLUSTRATION

Here is the illustration of the designed algorithm. Suppose we have a matrix of size *3*×*3* as given in the table and we need to store the elements in the array of size 9.

|  |  |  |
| --- | --- | --- |
| 3 | 28 | 5 |
| 7 | 9 | 17 |
| 1 | 15 | 13 |

The created array and sorting using merge sort:

Updated matrix:

|  |  |  |
| --- | --- | --- |
| 1 | 3 | 5 |
| 7 | 9 | 13 |
| 15 | 17 | 28 |

Let the element to be searched is 5.

Now, we will assume the matrix to be a 1-D array and apply binary search in order to find the required element.

Let l=0 and h=8[(3\*3)-1].

Iteration1:

Middle element((l+h)/2)=4.

i(mid/n)=1

j(mid%n)=1

Now, 5 < 7(arr[1][1]) then h=3(mid-1)

Iteration2:

Middle element((l+h)/2)=1.

i(mid/n)=0

j(mid%n)=2

Now, 5 = 5(arr[0][2]) then found at (0, 2).

IV. ALGORITHM AND ANALYSIS

**Algorithm:**Matrix search using merge sort

**Input:**Matrix elements

**Output:**Position of the element to be searched and no, if the element is not present.

**DEFINE** MAX 100

**Algorithm1:** merge(int arr[], int l, int m, int r)

1. *n1 m - l + 1*
2. *n2 r - m*
3. *create temp arrays L[n1], R[n2]*
4. **for** *i 0 to n1*
5. *L[i] arr[l + i]*
6. *i++*
7. **for** *j 0 to n2*
8. *R[j] arr[m + 1+ j]*
9. *j++*
10. *i 0*
11. *j 0*
12. *k l*
13. **while** *i < n1 and j < n2* **do**
14. **if** *L[i] <= R[j]* **then**
15. *arr[k] L[i]*
16. *i++*
17. **else**
18. *arr[k] R[j]*
19. *j++*
20. *k++*
21. **while** *i < n1* **do**
22. *arr[k] L[i]*
23. *i++*
24. *k++*
25. **while** *j < n2* **do**
26. *arr[k] R[j]*
27. *j++*
28. *k++*

**Algorithm2:** mergeSort(int arr[], int l, int r)

1. **if** *l < r* **then**
2. *m l+(r-l)/2*
3. *mergeSort(arr, l, m)*
4. *mergeSort(arr, m+1, r)*
5. *merge(arr, l, m, r)*

**Algorithm3:** sortedMatrixSearch(int matrix[m][n], int l, int h, int x)

1. **while** *l<=h* **do**
2. *mid=(l+h)/2*
3. *i=mid/n*
4. *j=mid%n*
5. **if** *x == matrix[i][j]* **then**
6. *Print “i and j”*
7. **if** *x < matrix[i][j]* **then**
8. *h=mid-1*
9. **else**
10. *l=mid+1*
11. *print “Element not found”*

**Algorithm4:** sortMatrix(int matrix[MAX][MAX], int n, int m)

1. *temp[m*×*n] 0*
2. *s 0*
3. **for** *i 0 to n*
4. **for** *j 0 to m*
5. *temp[s] = matrix[i][j]*
6. *j++*
7. *s++*
8. *i++*
9. *temp\_size = sizeof(temp)/sizeof(temp[0])*
10. *mergeSort(temp, 0, temp\_size - 1)*
11. *s 0*
12. **for** *i 0 to n*
13. **for** *j 0 to m*
14. *matrix[i][j] = temp[s]*
15. *j++*
16. *s++*
17. *i++*

**Algorithm:** main(int matrix[m][n],int m,int n,int x)

1. *sortMatrix(matrix, n, m)*
2. *l 0*
3. *h (m\*n)-1*
4. *sortedMatrixSearch(matrix,l,h,x)*

APRIORI ANALYSIS

The complexity of the algorithm is directly related to the number of steps, size of the matrix and the complexity of the sorting algorithm. The sorting algorithm used in this matrix search is merge sort which is a divide and conquer algorithm which divides the problem into two halves and require time proportional to n\*log(n) where n is the size of the array. We will use this to calculate the overall complexity of the algorithm.

|  |  |  |
| --- | --- | --- |
| Step | Time | Iteration |
| 1 | 1 | 1 |
| 2 | 1 | 1 |
| 3 | 2 | n |
| 4 | 2 | m\*n |
| 5 | 1 | m\*n |
| 6 | 1 | m\*n |
| 7 | 1 | m\*n |
| 8 | 1 | n |
| 9 | 2 | 1 |
| 10 | 1 | (m\*n)\*log(m\*n) |
| 11 | 1 | 1 |
| 12 | 2 | n |
| 13 | 2 | m\*n |
| 14 | 1 | m\*n |
| 15 | 1 | m\*n |
| 16 | 1 | m\*n |
| 17 | 1 | n |

Table1: Complexity analysis of sortMatrix() function.

For searching the element from the sorted matrix, we used binary search. We assume the matrix to be 1-D array, hence the size would be m\*n. So, the complexity will be

1st step=> T(m\*n)=T(m\*n/2) + 1

2nd step=> T(m\*n/2)=T(m\*n/4) + 1 ……

[ T(m\*n/4) = T(m\*n/2^2) ]

3rd step=> T(m\*n/4)=T(m\*n/8) + 1 ……

[ T(m\*n/8)= T(m\*n/2^3) ]

.

.

kth step=> T(m\*n/2^k-1)=T(m\*n/2^k) + 1\*(k times)

Adding all the equations we get,

T(m\*n) = T(m\*n/2^k) + k\*1 ---eq(final)

=> m\*n/2^k= 1

=> m\*n=2^k

=> log(m\*n)=k [taken log(base 2) on both sides ]

Put k= log(m\*n) in eq(final)

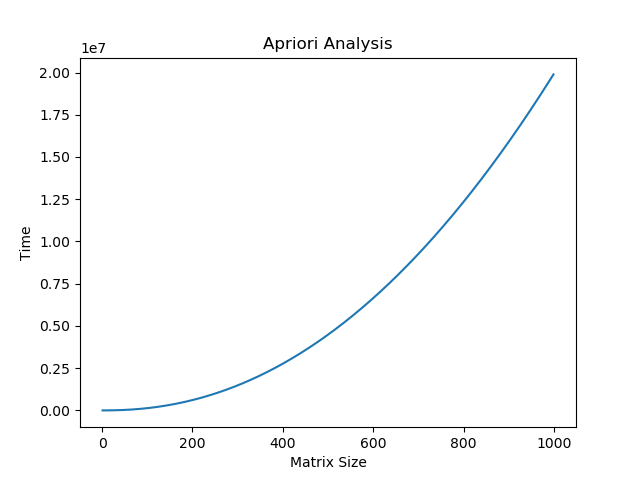
T(m\*n) = T(1) + log(m\*n)

T(m\*n) = 1 + log (m\*n)

[we know that T(1) = 1 , because it’s a base condition as we are left with only one element in the array and that is the element to be searched so we return 1]

Hence, the complexity of the sortedMatrixSearch function is

Now, the final complexity of the algorithm is the sum of the complexities of sortMatrix() function and sortedMatrixSearch() function. Hence, the complexity is

Figure1:Comparison of Time complexity for Apriori analysis of Algorithm.

V. EXPERIMENTAL ANALYSIS AND PROFILING

The program was run against various test-cases and the time required for each test-case was collected and then plotted against the size of the matrix(m\*n).

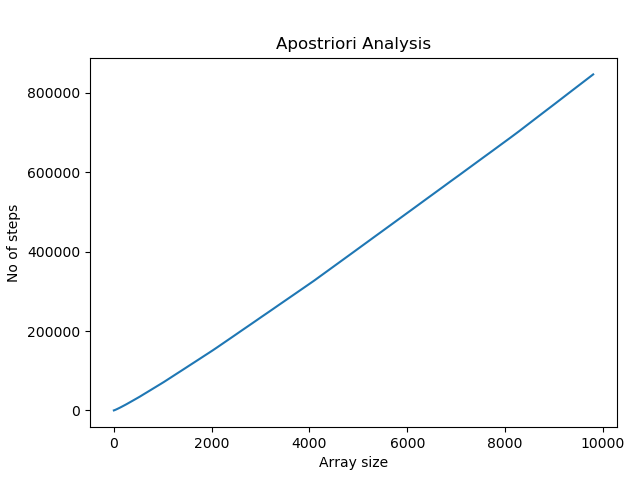


Figure2: Time complexity graph for Aposteriori analysis.

The above graph shows that the Apriori analysis is consistent with the Aposteriori analysis as the theoretical graph and the experimental graph has the same nature and hence we could say that the analysis made is true to a certain extent.

VI. CONCLUSION

From this paper we can conclude that the algorithm is an algorithm. The time complexity of merge sort is same in all cases, i.e., best, worst and average case which is . Similarly, the complexity of binary search is same in average and worst case i.e., . The best case complexity of binary search is . The final complexity of the algorithm is the sum of the complexities of sorting and searching algorithm which implies the complexity of the algorithm will remain the same in the best case as well because the best case complexity of merge sort is more than the binary search. Hence, the total complexity of the algorithm in all cases is .

VII. REFERENCES

[1.] Introduction to Algorithms by Tas H. Cormen, Charles E. Leiserson, Ronald L. Rivest.

[2.] https://en.wikipedia.org/wiki/Matrix\_ (mathematics)

[3.] https://www.geeksforgeeks.org/merge-sort/

[4.] https://www.geeksforgeeks.org/binary-search/

[5.] https://www.geeksforgeeks.org/search-element- sorted-matrix/