Find the Repeated Element

Akash Anand, Parth Kataria, Shruti Nanda
iit2019015@iiita.ac.in, iit20190016@iiita.ac.in, iit2019017@iiita.ac.in
IV semester,Department of Information Technology
Indian Institute of Information Technology, Allahabad

Abstract-This paper introduces the algorithms to find the one repeated element in a sorted array. The idea is to design,explain and analyse these algorithms and conclude with the most efficient algorithm in terms of time and space complexities.

1. INTRODUCTION

We are given a sorted array of n elements containing each element in the range from 1 to n-1 with only one element occurring twice and our task is to design the most efficient algorithm to find that repeating element.This report further contains:

1. Algorithm Design and Analysis
2. Experimental study and complexity.
3. Conclusion

2. ALGORITHM DESIGN

Let the size of the given sorted array be n therefore all elements of the array must be in the range of 1 to n-1.

Approach 1:Brute-force

```
function  brute(int a[],int n)
   for  i ← 0 to n-1 do
     if a[i]==a[i+1] then
        print a[i]
        break
   end for
end  function
```

It is a simple brute force method comparing adjacent elements and if for a case they are equal then it is our repeated element.

Approach 2:Hashing

```
function  hashing(int a[],int n)
   int hash[n]={0};
   for  i ← 0 to n do
     if hash[a[i]]==1 then
          print a[i]
          break
     else
          hash[a[i]]=1
   end for
end  function
```

In this approach a hash table is maintained for elements in the range 1 to n-1 with hash[i]=1 describing the element is present in the array and for an element if hash[i]=1 meaning it is repeated.

Approach 3:Divide and Conquer

```
function  bin(int a[],int low,int high)
  if low>high then
    return -1
  int mid = (low + high) / 2
  if a[mid]!=mid+1 then
     if mid>0 and a[mid]==a[mid-1] then
        return mid
     return bin(a,low,mid-1)
  return bin(a,mid+1,high)
end  function
```

In this approach we start by checking the middle element if it is repeating.Then we check the position of the middle element if it is proper then we check in the right half of the array else in the left half.

3. EXPERIMENTAL STUDY AND ANALYSIS

A priori analysis:

For Approach1:
1)Time Complexity:
Best case : Ω(1)
Average case : θ(n/2)
Worst case : O(n)
2)Space Complexity-O(1)

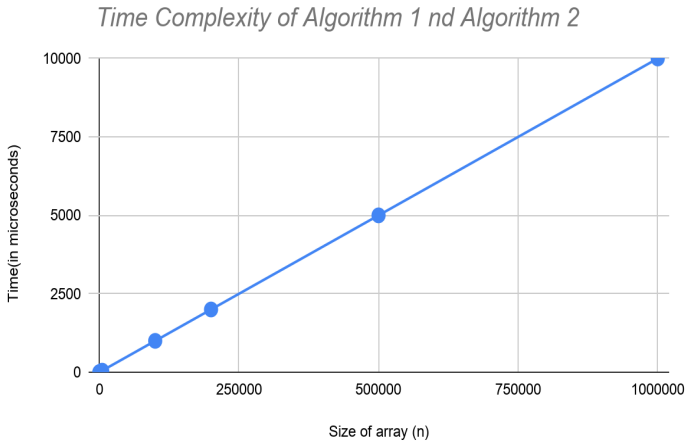
For Approach2:
1)Time Complexity:
Best case : Ω(1)
Average case : θ(n/2)
Worst case : O(n)
2)Space Complexity-O(n)

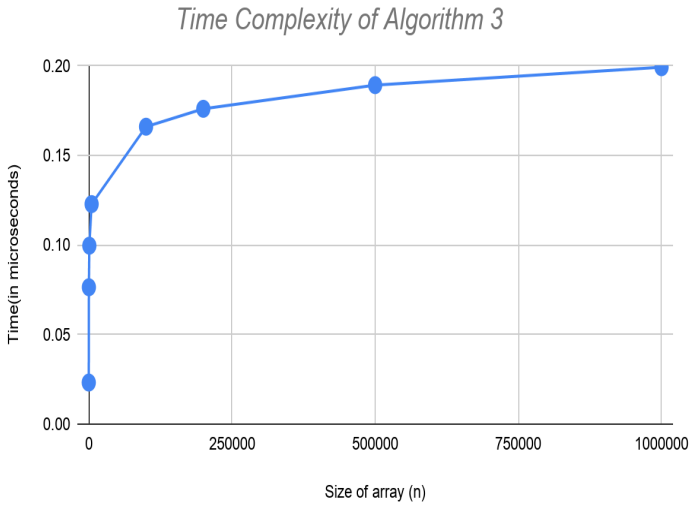
For Approach3:
1)Time Complexity:
Best case : Ω(1)
Average case : θ(logn)
Worst case : O(logn)
2)Space Complexity-O(1)

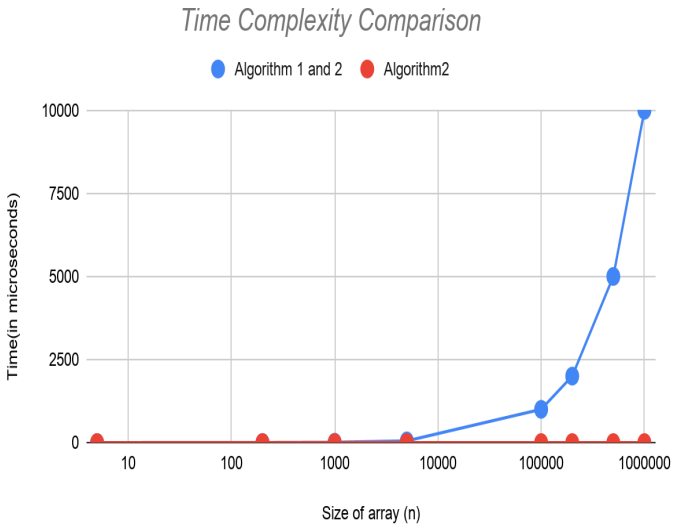
S no.	N	Algorithm 1 and 2	Algorithm2
1	5	0.05	0.02321928095
2	200	2	0.0764385619
3	1000	10	0.09965784285
4	5000	50	0.1228771238
5	100000	1000	0.1660964047
6	200000	2000	0.1760964047
7	500000	5000	0.1893156857
8	1000000	10000	0.1993156857

4. TIME COMPLEXITY

The algorithms were tested against random sorted arrays of variable sizes.The observation thus obtained from this experiment is represented in the form of individual and campirison graphs given below:







5. CONCLUSION

After observing and analysing the above algorithms we can conclude that the algorithm based on the third approach is much more effective than others based on time complexity and first and third approach are better than second in terms of space complexity.

6. REFERENCES

1. https://www.geeksforgeeks.org/find-repeating-element-sorted-array-size-n/