ARRAY

QUESTION 2

Majority Element

Write a function which takes an array and prints the majority element (if it exists), otherwise prints "No Majority Element". A ***majority element*** in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

Examples :

```
Input : {3, 3, 4, 2, 4, 4, 2, 4, 4}Output : 4

Input : {3, 3, 4, 2, 4, 4, 2, 4}Output : No Majority Element
```

METHOD 1 (Basic)

The basic solution is to have two loops and keep track of maximum count for all different elements. If maximum count becomes greater than n/2 then break the loops and return the element having maximum count. If maximum count doesn't become more than n/2 then majority element doesn't exist.

Below is the implementation of the above approach :

```
// Java  program to find Majority
// element in an array

import java.io.*;

class GFG {

// Function to find Majority element
// in an array
static void findMajority(int arr[], int n)
{
    int maxCount = 0;
    int index = -1; // sentinels
    for(int i = 0; i < n; i++)
    {
        int count = 0;
        for(int j = 0; j < n; j++)
        {
            if(arr[i] == arr[j])
            count++;
        }

        // update maxCount if count of
        // current element is greater
```

```
         if(count > maxCount)
         {
             maxCount = count;
             index = i;
         }
     }

     // if maxCount is greater than n/2
     // return the corresponding element
     if (maxCount > n/2)
     System.out.println (arr[index]);

     else
     System.out.println ("No Majority Element");
}

// Driver code
     public static void main (String[] args) {

         int arr[] = {1, 1, 2, 1, 3, 5, 1};
         int n = arr.length;

     // Function calling
     findMajority(arr, n);
     }
//This code is contributed by ajit.
}
```

Output : 1

Time Complexity : O(n*n).

Auxiliary Space : O(1).

METHOD 2 (Using Binary Search Tree)

Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if count of a node becomes more than n/2 then return.

The method works well for the cases where n/2+1 occurrences of the majority element is present in the starting of the array, for example {1, 1, 1, 1, 1, 2, 3, 4}.

Time Complexity : If a Binary Search Tree is used then time complexity will be O(n^2). If a self-balancing-binary-search tree is used then O(nlogn)

Auxiliary Space : O(n)

METHOD 3 (Using Moore's Voting Algorithm)

This is a two step process.

NOTE : This Method only works when we are given that majority element do exist in the array , otherwise this method won't work , as in the problem definition we said that majority element may or may not exist but for applying this approach you can assume that majority element do exist in the given input array

1. The first step gives the element that may be majority element in the array. If there is a majority element in an array, then this step will definitely return majority element, otherwise it will return candidate for majority element.
2. Check if the element obtained from above step is majority element.This step is necessary as we are not always sure that element return by first step is majority element.

1. Finding a Candidate :

The algorithm for first phase that works in O(n) is known as Moore's Voting Algorithm. Basic idea of the algorithm is that if we cancel out each occurrence of an element *e* with all the other elements that are different from *e* then *e* will exist till end if it is a majority element.

```
findCandidate(a[], size)
1.  Initialize index and count of majority element
     maj_index = 0, count = 1
2.  Loop for i = 1 to size – 1
    (a) If a[maj_index] == a[i]
          count++
    (b) Else
        count--;
    (c) If count == 0
          maj_index = i;
          count = 1
3.  Return a[maj_index]
```

Above algorithm loops through each element and maintains a count of a[maj_index]. If the next element is same then increment the count, if the next element is not same then decrement the count, and if the count reaches 0 then changes the maj_index to the current element and set the count again to 1. So, the first phase of the algorithm gives us a candidate element.

In the second phase we need to check if the candidate is really a majority element. Second phase is simple and can be easily done in O(n). We just need to check if count of the candidate element is greater than n/2.

Example :

Let the array be A[] = 2, 2, 3, 5, 2, 2, 6

- Initialize maj_index = 0, count = 1
- Next element is 2, which is same as a[maj_index] => count = 2

- Next element is 3, which is different from a[maj_index] => count = 1
- Next element is 5, which is different from a[maj_index] => count = 0
- Since count = 0, change candidate for majority element to 5 => maj_index = 3, count = 1
- Next element is 2, which is different from a[maj_index] => count = 0
- Since count = 0, change candidate for majority element to 2 => maj_index = 4
- Next element is 2, which is same as a[maj_index] => count = 2
- Next element is 6, which is different from a[maj_index] => count = 1
- Finally candidate for majority element is 2.

First step uses Moore's Voting Algorithm to get a candidate for majority element.

2. Check if the element obtained in step 1 is majority element or not :

```
• Find the candidate for majority
• If candidate is majority. i.e., appears more than n/2 times.
                Print the candidate
•   Else
      ▪       Print "No Majority Element"
```

```
/* Program for finding out majority element in an array */

class MajorityElement
{
    /* Function to print Majority Element */
    void printMajority(int a[], int size)
    {
        /* Find the candidate for Majority*/
        int cand = findCandidate(a, size);

        /* Print the candidate if it is Majority*/
        if (isMajority(a, size, cand))
            System.out.println(" " + cand + " ");
        else
            System.out.println("No Majority Element");
```

```
}

/* Function to find the candidate for Majority */
int findCandidate(int a[], int size)
{
    int maj_index = 0, count = 1;
    int i;
    for (i = 1; i < size; i++)
    {
        if (a[maj_index] == a[i])
            count++;
        else
            count--;
        if (count == 0)
        {
            maj_index = i;
            count = 1;
        }
    }
    return a[maj_index];
}

/* Function to check if the candidate occurs more
   than n/2 times */
boolean isMajority(int a[], int size, int cand)
{
    int i, count = 0;
    for (i = 0; i < size; i++)
    {
            if (a[i] == cand)
                count++;
        }
        if (count > size / 2)
            return true;
        else
            return false;
    }

    /* Driver program to test the above functions */
    public static void main(String[] args)
    {
        MajorityElement majorelement = new MajorityElement();
        int a[] = new int[]{1, 3, 3, 1, 2};
        int size = a.length;
        majorelement.printMajority(a, size);
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output: No Majority Element

Time Complexity: O(n)

Auxiliary Space : O(1)

METHOD 4 (Using Hashmap) : This method is somewhat similar to Moore voting algorithm in terms of time complexity, but in this case there is no need of second step of Moore voting algorithm.But as usual, here space complexity becomes O(n).

In Hashmap(key-value pair), at value,maintain a count for each element(key) and whenever count is greater than half of array length, we are just returning that key(majority element).

Time Complexity : O(n)

Auxiliary Space : O(n)

Below is the implementation.

```
import java.util.HashMap;

/* Program for finding out majority element in an array */

class MajorityElement
{
    private static void findMajority(int[] arr)
    {
        HashMap<Integer,Integer> map = new HashMap<Integer, Integer>();

        for(int i = 0; i < arr.length; i++) {
            if (map.containsKey(arr[i])) {
                    int count = map.get(arr[i]) +1;
                    if (count > arr.length /2) {
                        System.out.println("Majority found :- " + arr[i]);
                        return;
                    } else
                        map.put(arr[i], count);

            }
            else
                map.put(arr[i],1);
            }
            System.out.println(" No Majority element");
    }


    /* Driver program to test the above functions */
    public static void main(String[] args)
    {
        int a[] = new int[]{2,2,2,2,5,5,2,3,3};

        findMajority(a);
    }
}
// This code is contributed by  karan malhotra
```

Output:

```
Majority found :- 2
```