**Find the Majority Element that occurs more than N/2 times**

**Problem Statement:** Given an array of **N integers**, write a program to return an element that occurs more than **N/2** times in the given array. You may consider that such an element always exists in the array.

**Examples**

```
Example 1:
Input Format: N = 3, nums[] = {3,2,3}
Result: 3
Explanation: When we just count the occurrences of each number and compare with half of the size of
the array, you will get 3 for the above solution.

Example 2:
Input Format:  N = 7, nums[] = {2,2,1,1,1,2,2}

Result: 2

Explanation: After counting the number of times each element appears and comparing it with half of
array size, we get 2 as result.

Example 3:
Input Format:  N = 10, nums[] = {4,4,2,4,3,4,4,3,2,4}

Result: 4
```

Brute Force Approach
Algorithm / Intuition

**Naive Approach:**

**Approach:**

The steps are as follows:

1. We will run a loop that will select the elements of the array one by one.
2. Now, for each element, we will run another loop and count its occurrence in the given array.
3. If any element occurs more than the floor of (N/2), we will simply return it.

Code
```
import java.util.*;

public class MajorityEle2 {
    public static int majorityElement(int []v) {
        //size of the given array:
        int n = v.length;

        for (int i = 0; i < n; i++) {
            //selected element is v[i]
            int cnt = 0;
            for (int j = 0; j < n; j++) {
                // counting the frequency of v[i]
                if (v[j] == v[i]) {
                    cnt++;
                }
            }

            // check if frquency is greater than n/2:
            if (cnt > (n / 2))
                return v[i];
        }

        return -1;
    }

    public static void main(String args[]) {
        int[] arr = {2, 2, 1, 1, 1, 2, 2};
        int ans = majorityElement(arr);
        System.out.println("The majority element is: " + ans);

    }

}
```
Output: The majority element is: 2

Complexity Analysis

**Time Complexity:** O(N2), where N = size of the given array. **Reason:** For every element of the array the inner loop runs for N times. And there are N elements in the array. So, the total time complexity is O(N2). **Space Complexity:** O(1) as we use no extra space.

Better Approach
Algorithm / Intuition

**Solution 2 (Better):**

**Intuition:**

Use a better data structure to reduce the number of look-up operations and hence the time complexity. Moreover, we have been calculating the count of the same element again and again - so we have to reduce that also.

**Approach:**

1. Use a hashmap and store as *(key,* value) pairs. (Can also use frequency array based on the size of nums)
2. Here the key will be the element of the array and the value will be the number of times it occurs.
3. Traverse the array and update the value of the key. Simultaneously check if the value is greater than the **floor of N/2**.
    1. If yes, return the key
    2. Else iterate forward.

Code

```java
import java.util.*;

public class MajorityEle2 {
    public static int majorityElement(int []v) {
        //size of the given array:
        int n = v.length;

        //declaring a map:
        HashMap<Integer, Integer> mpp = new HashMap<>();

        //storing the elements with its occurnce:
        for (int i = 0; i < n; i++) {
            int value = mpp.getOrDefault(v[i], 0);
            mpp.put(v[i], value + 1);
        }

        //searching for the majority element:
        for (Map.Entry<Integer, Integer> it : mpp.entrySet()) {
            if (it.getValue() > (n / 2)) {
                return it.getKey();
            }
        }

        return -1;
    }

    public static void main(String args[]) {
        int[] arr = {2, 2, 1, 1, 1, 2, 2};
        int ans = majorityElement(arr);
        System.out.println("The majority element is: " + ans);

    }

}
```

Output: The majority element is: 2

Complexity Analysis

**Time Complexity:** O(N*logN) + O(N), where N = size of the given array.
**Reason:** We are using a map data structure. Insertion in the map takes logN time. And we are doing it for N elements. So, it results in the first term O(N*logN). The second O(N) is for checking which element occurs more than floor(N/2) times. If we use unordered_map instead, the first term will be O(N) for the best and average case and for the worst case, it will be O(N2).

**Space Complexity:** O(N) as we are using a map data structure.

Optimal Approach
Algorithm / Intuition

**Optimal Approach: Moore's Voting Algorithm:**

**Intuition:**

If the array contains a majority element, its occurrence must be greater than the floor(N/2). Now, we can say that the count of minority elements and majority elements is equal up to a certain point in the array. So when we traverse through the array we try to keep track of the count of elements and the element itself for which we are tracking the count.

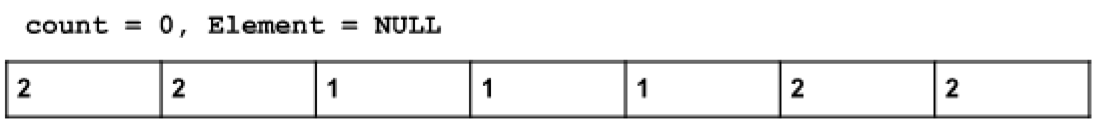
After traversing the whole array, we will check the element stored in the variable. If the question states that the array must contain a majority element, the stored element will be that one but if the question does not state so, then we need to check if the stored element is the majority element or not. If not, then the array does not contain any majority element.

**Approach:**

1. Initialize 2 variables:
   **Count** – for tracking the count of element
   **Element** – for which element we are counting
2. Traverse through the given array.
   1. If **Count** is 0 then store the current element of the array as **Element**.
   2. If the current element and **Element** are the same increase the **Count** by 1.
   3. If they are different decrease the **Count** by 1.
3. The integer present in **Element** should be the result we are expecting

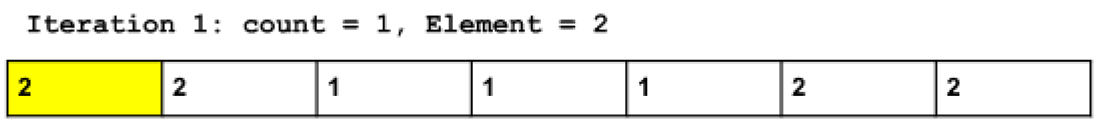
**Dry Run:**

The yellow-colored element represents the current element.



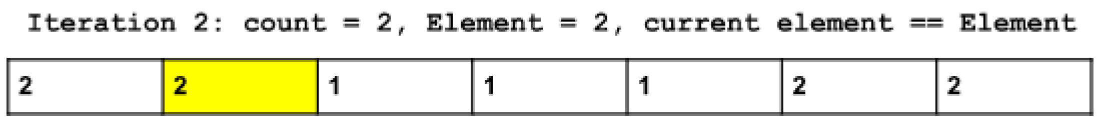
count = 0, Element = NULL

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 1 | 2 | 2 |



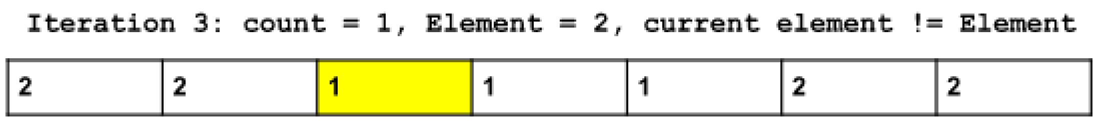
Iteration 1: count = 1, Element = 2

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 1 | 2 | 2 |



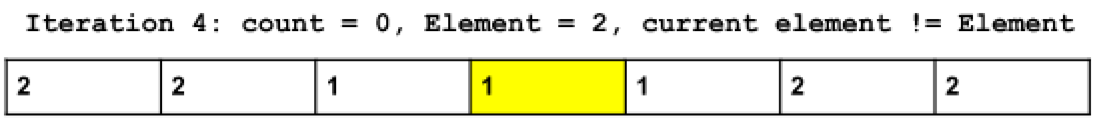
Iteration 2: count = 2, Element = 2, current element == Element

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 1 | 2 | 2 |



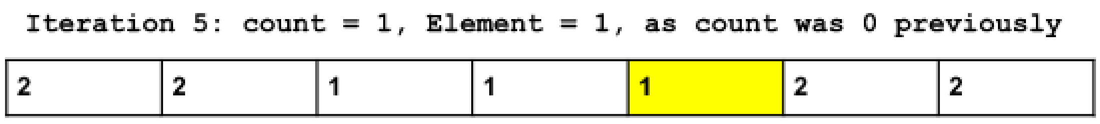
Iteration 3: count = 1, Element = 2, current element != Element

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 1 | 2 | 2 |



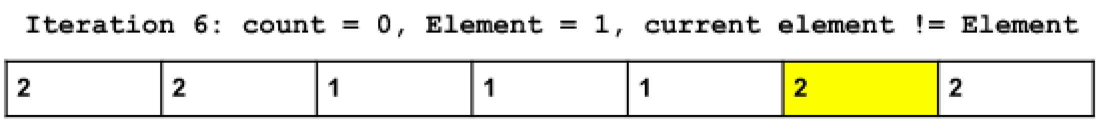
Iteration 4: count = 0, Element = 2, current element != Element

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 1 | 2 | 2 |



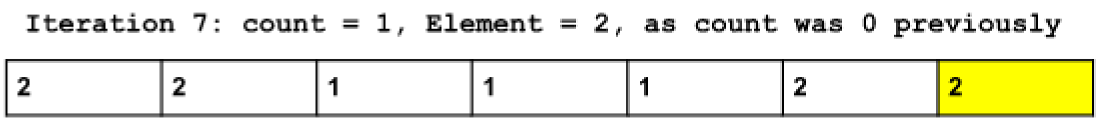
Iteration 5: count = 1, Element = 1, as count was 0 previously

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 1 | 2 | 2 |



Iteration 6: count = 0, Element = 1, current element != Element

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 1 | 2 | 2 |



Iteration 7: count = 1, Element = 2, as count was 0 previously

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 1 | 1 | 2 | 2 |

Basically, we are trying to keep track of the occurrences of the majority element and minority elements dynamically. That is why, in iteration 4, the count becomes 0 as the occurrence of Element and the occurrence of the other elements are the same. So, they canceled each other. This is how the process works. The element with the most occurrence will remain and the rest will cancel themselves.

Here, we can see that 2 is the majority element. But if in this array, the last two elements were 3, then the Element variable would have stored 3 instead of 2. For that, we need to check if the Element is the majority element by traversing the array once more. But if the question guarantees that the given array contains a majority element, then we can bet the Element will store the majority one.

**Note:** *For a better understanding of intuition, please watch the video at the bottom of the page.*

Code

```
import java.util.*;

public class MajorityEle2 {
    public static int majorityElement(int []v) {
        //size of the given array:
        int n = v.length;
        int cnt = 0; // count
        int el = 0; // Element

        //applying the algorithm:
        for (int i = 0; i < n; i++) {
            if (cnt == 0) {
                cnt = 1;
                el = v[i];
            } else if (el == v[i]) cnt++;
            else cnt--;
        }

        //checking if the stored element
        // is the majority element:
        int cnt1 = 0;
        for (int i = 0; i < n; i++) {
            if (v[i] == el) cnt1++;
        }

        if (cnt1 > (n / 2)) return el;
        return -1;
    }

    public static void main(String args[]) {
        int[] arr = {2, 2, 1, 1, 1, 2, 2};
        int ans = majorityElement(arr);
        System.out.println("The majority element is: " + ans);

    }

}
```

Output: The majority element is: 2

Complexity Analysis

**Time Complexity:** O(N) + O(N), where N = size of the given array.
**Reason:** The first O(N) is to calculate the count and find the expected majority element. The second one is to check if the expected element is the majority one or not.

**Note:** *If the question states that the array must contain a majority element, in that case, we do not need the second check. Then the time complexity will boil down to O(N).*

**Space Complexity:** O(1) as we are not using any extra space.