Majority Elements(N/3 times) | Find the elements that appears more than N/3 times in the array

Problem Statement: Given an array of N integers. Find the elements that appear more than **N/3** times in the array. If no such element exists, return an empty vector.

Pre-requisite: Majority Element(>N/2 times)

Examples

Example 1:
Input Format: N = 5, array[] = {1,2,2,3,2}
Result: 2
Explanation: Here we can see that the Count(1) = 1, Count(2) = 3 and Count(3) = 1.Therefore, the count of 2 is greater than N/3 times. Hence, 2 is the answer.

Example 2:
Input Format: N = 6, array[] = {11,33,33,11,33,11}
Result: 11 33
Explanation: Here we can see that the Count(11) = 3 and Count(33) = 3. Therefore, the count of both 11 and 33 is greater than N/3 times. Hence, 11 and 33 is the answer.

Brute Force Approach
Algorithm / Intuition

Solution:

Observation: How many integers can occur more than floor(N/3) times in the given array:

If we closely observe, in the given array, there can be a maximum of two integers that can occur more than floor(N/3) times. Let's understand it using the following scenario:
Assume there are 8 elements in the given array. Now, if there is any majority element, it should occur more than floor(8/3) = 2 times. So, the majority of elements should occur at least 3 times. Now, if we imagine there are 3 majority elements, then the total occurrence of them will be 3X3 = 9 i.e. greater than the size of the array. But this should not happen. That is why ***there can be a maximum of 2 majority elements.***

Naive Approach (Brute-force):

1. We will run a loop that will select the elements of the array one by one.
2. Now, for each unique 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/3), we will include it in our answer.
4. While traversing if we find any element that is already included in our answer, we will just skip it.

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

Code

```
import java.util.*;

public class MajorityEle {
    public static List<Integer> majorityElement(int []v) {
        int n = v.length; //size of the array
        List<Integer> ls = new ArrayList<>(); // list of answers

        for (int i = 0; i < n; i++) {
            //selected element is v[i]:
            // Checking if v[i] is not already
            // a part of the answer:
            if (ls.size() == 0 || ls.get(0) != 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/3:
                if (cnt > (n / 3))
                    ls.add(v[i]);
            }

            if (ls.size() == 2) break;
        }

        return ls;
    }

    public static void main(String args[]) {
        int[] arr = {11, 33, 33, 11, 33, 11};
        List<Integer> ans = majorityElement(arr);
        System.out.print("The majority elements are: ");
        for (int i = 0; i < ans.size(); i++) {
            System.out.print(ans.get(i) + " ");
        }
        System.out.println();
    }
}
```

Output: The majority elements are: 11 33

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 are using a list that stores a maximum of 2 elements. The space used is so small that it can be considered constant.

Better Approach (Using Hashing):

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:

The steps are as follows:

1. Use a hashmap and store the elements as <key, value> pairs. (Can also use frequency array based on the size of nums).
   Here the key will be the element of the array and the value will be the number of times it occurs.
2. Traverse the whole array and update the occurrence of each element.
3. After that, check the map if the value for any element is greater than the **floor of N/3**.
    1. If yes, include it in the list.
    2. Else iterate forward.
4. Finally, return the list.

Dry Run:

Let's take the example of arr[] = {10,20,40,40,40}, n=5.

First, we create an unordered map to store the count of each element.

Now traverse through the array

Found 10 at index 0, increase the value of key 10 in the map by 1.

Found 20 at index 1, increase the value of key 20 in the map by 1.

Found 40 at index 2, increase the value of key 40 in the map by 1.

Found 40 at index 3, increase the value of key 40 in the map by 1.

Found 40 at index 4, increase the value of key 40 in the map by 1.

Now, Our map will look like this:

```
10 -> 1
20 -> 1
40 -> 3
```

Now traverse through the map,

We found that the value of key 40 is greater than the floor(N/3). So, 40 is the answer.

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

Code

```
import java.util.*;

public class MajorityEle {
    public static List<Integer> majorityElement(int []v) {
        int n = v.length; //size of the array
        List<Integer> ls = new ArrayList<>(); // list of answers

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

        // least occurrence of the majority element:
        int mini = (int)(n / 3) + 1;

        //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);

            //checking if v[i] is
            // the majority element:
            if (mpp.get(v[i]) == mini) {
                ls.add(v[i]);
            }
            if (ls.size() == 2) break;
        }

        return ls;
    }

    public static void main(String args[]) {
        int[] arr = {11, 33, 33, 11, 33, 11};
        List<Integer> ans = majorityElement(arr);
        System.out.print("The majority elements are: ");
        for (int i = 0; i < ans.size(); i++) {
            System.out.print(ans.get(i) + " ");
        }
        System.out.println();
    }
}
```

Output: The majority elements are: 33 11

Complexity Analysis

Time Complexity: O(N*logN), 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).
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. We are also using a list that stores a maximum of 2 elements. That space used is so small that it can be considered constant.

Optimal Approach
Algorithm / Intuition

Optimal Approach (Extended Boyer Moore's Voting Algorithm):

Approach:

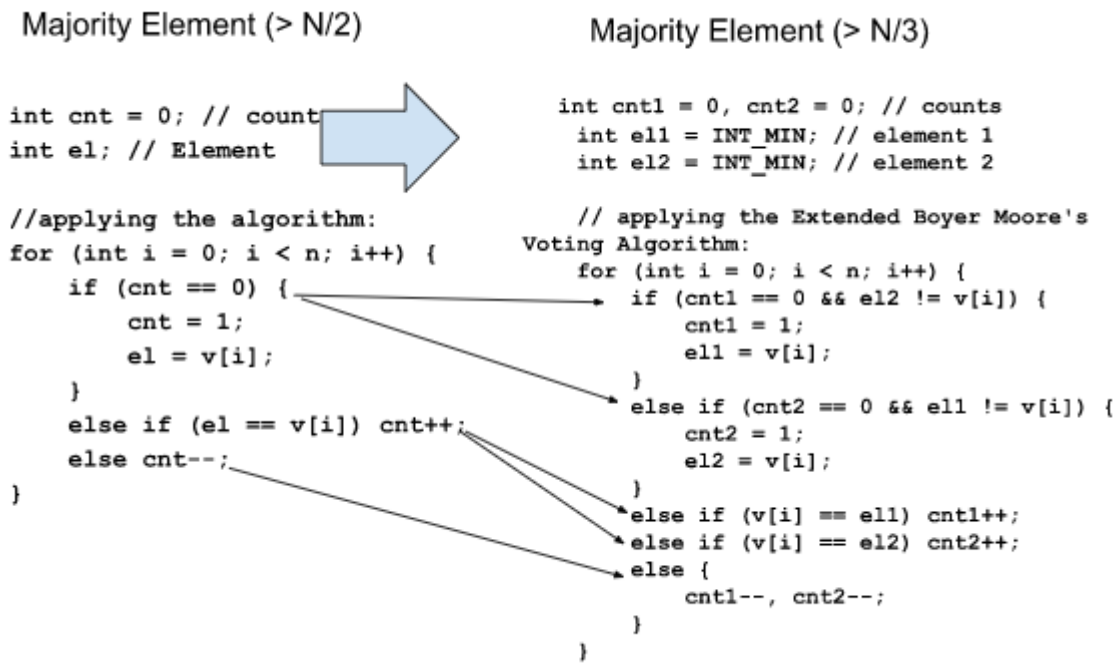
1. Initialize 4 variables:
   cnt1 & cnt2 – for tracking the counts of elements
   el1 & el2 – for storing the majority of elements.
2. Traverse through the given array.
    1. If **cnt1** is 0 and the current element is not el2 then store the current element of the array as **el1 along with increasing the cnt1 value by 1**.
    2. If **cnt2** is 0 and the current element is not el1 then store the current element of the array as **el2 along with increasing the cnt2 value by 1**.
    3. If the current element and **el1** are the same increase the **cnt1** by 1.
    4. If the current element and **el2** are the same increase the **cnt2** by 1.
    5. Other than all the above cases: decrease cnt1 and cnt2 by 1.
3. The integers present in **el1 & el2** should be the result we are expecting. So, using another loop, we will manually check their counts if they are greater than the floor(N/3).

Intuition: If the array contains the majority of elements, their occurrence must be greater than the floor(N/3). 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 counts of elements and the elements themselves for which we are tracking the counts.

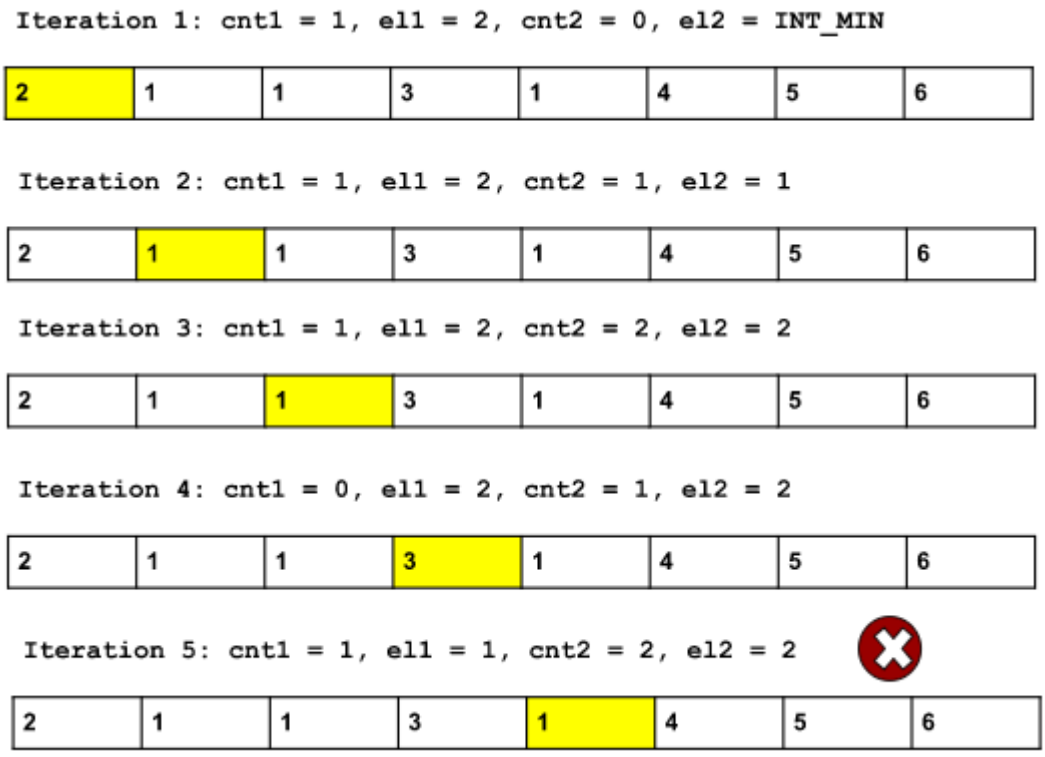
After traversing the whole array, we will check the elements stored in the variables. Then we need to check if the stored elements are the majority elements or not by manually checking their counts.

Note: This intuition is simply the logic of cancellation i.e. a variation of Moore's Voting Algorithm that we used in the problem Majority Element (> N/2).

The projection will be the following:



Edge Case: Why we are adding extra checks like el2 != v[i] and el1 != v[i] in the first if statements? Let's understand it using an example: Assume the given array is: {2, 1, 1, 3, 1, 4, 5, 6}. Now apply the algorithm without the checks:



We can clearly notice that in iteration 5, el1 and el2 both are set to 1 as cnt1 becomes 0 in iteration 4. But this is incorrect. So, to avoid this edge case, we are checking if the current element is already included in our elements, and if it is, we will not again include it in another variable.

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

Code

```
import java.util.*;

public class MajorityEle {
    public static List<Integer> majorityElement(int []v) {
        int n = v.length; //size of the array

        int cnt1 = 0, cnt2 = 0; // counts
        int el1 = Integer.MIN_VALUE; // element 1
        int el2 = Integer.MIN_VALUE; // element 2

        // applying the Extended Boyer Moore's Voting Algorithm:
        for (int i = 0; i < n; i++) {
            if (cnt1 == 0 && el2 != v[i]) {
                cnt1 = 1;
                el1 = v[i];
            } else if (cnt2 == 0 && el1 != v[i]) {
                cnt2 = 1;
                el2 = v[i];
            } else if (v[i] == el1) cnt1++;
            else if (v[i] == el2) cnt2++;
            else {
                cnt1--; cnt2--;
            }
        }

        List<Integer> ls = new ArrayList<>(); // list of answers

        // Manually check if the stored elements in
        // el1 and el2 are the majority elements:
        cnt1 = 0; cnt2 = 0;
        for (int i = 0; i < n; i++) {
            if (v[i] == el1) cnt1++;
            if (v[i] == el2) cnt2++;
        }

        int mini = (int)(n / 3) + 1;
        if (cnt1 >= mini) ls.add(el1);
        if (cnt2 >= mini) ls.add(el2);

        // Uncomment the following line
        // if it is told to sort the answer array:
        //Collections.sort(ls); //TC --> O(2*log2) ~ O(1);

        return ls;
    }

    public static void main(String args[]) {
        int[] arr = {11, 33, 33, 11, 33, 11};
        List<Integer> ans = majorityElement(arr);
        System.out.print("The majority elements are: ");
        for (int i = 0; i < ans.size(); i++) {
            System.out.print(ans.get(i) + " ");
        }
        System.out.println();
    }
}
```

Output: The majority elements are: 11 33

Complexity Analysis

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

Space Complexity: O(1) as we are only using a list that stores a maximum of 2 elements. The space used is so small that it can be considered constant.