Majority Element II(Majority Element (>N/3 times))

Given an integer array of size `n`, find all elements that appear more than `⌊ n/3 ⌋` times.

Example 1:

```
Input: nums = [3,2,3]

Output: [3]
```

Example 2:

```
Input: nums = [1]

Output: [1]
```

Example 3:

```
Input: nums = [1,2]

Output: [1,2]
```

Approach 1: Naïve

Traverse the array and count the elements occurrence

Time : O(n^2)

Space= O(1)

Approach 2 : HashMap

1. Add all elements with their frequencies into map
2. Traverse each element in the array and check its frequency is >n/3

```
public List<Integer> majorityElement(int[] nums) {
    int n=nums.length;
    HashMap<Integer,Integer> map=new HashMap<>();
    for(int i=0;i<n;i++)
    {
       map.put(nums[i],map.getOrDefault(nums[i],0)+1);
    }
    List<Integer> res=new ArrayList<>();
   for(int i=0;i<n;i++)
   {
      int count=map.get(nums[i]);
```

```
        if(count>n/3)
        {
           res.add(nums[i]);
           map.put(nums[i],0);
        }
     }
      return res;

   }
```

Time: O(n)

Space: O(n)

Approach 3: Moore's Voting algorithm

The idea is based on Moore's Voting algorithm. We first find two candidates. Then we check if any of these two candidates is actually a majority. Below is the solution for above approach.

```
static int appearsNBy3(int arr[], int n)
    {
        int count1 = 0, count2 = 0;

        // take the integers as the maximum
        // value of integer hoping the integer
        // would not be present in the array
        int first =  Integer.MIN_VALUE;;
        int second = Integer.MAX_VALUE;

        for (int i = 0; i < n; i++) {

            // if this element is previously
            // seen, increment count1.
            if (first == arr[i])
                count1++;

            // if this element is previously
            // seen, increment count2.
            else if (second == arr[i])
                count2++;

            else if (count1 == 0) {
                count1++;
                first = arr[i];
```

```
            }

            else if (count2 == 0) {
                count2++;
                second = arr[i];
            }

            // if current element is different
            // from both the previously seen
            // variables, decrement both the
            // counts.
            else {
                count1--;
                count2--;
            }
        }

        count1 = 0;
        count2 = 0;

        // Again traverse the array and
        // find the actual counts.
        for (int i = 0; i < n; i++) {
            if (arr[i] == first)
                count1++;

            else if (arr[i] == second)
                count2++;
        }

        if (count1 > n / 3)
            return first;

        if (count2 > n / 3)
            return second;

        return -1;
    }
```

Complexity Analysis:

- **Time Complexity**: **O(n)**
  First pass of the algorithm takes complete traversal over the array contributing to O(n) and another O(n) is consumed in checking if count1 and count2 is greater than floor(n/3) times.
- **Space Complexity: O(1)**
  As no extra space is required so space complexity is constant