majority element n by three times

Given an array of integers, return all integers that appear more than $floor(n/3)$ times.

For example, given the array `[1, 2, 1, 3, 1, 2, 2]`, where n = 7, $floor(7/3)$ = 2. Therefore, we need to return all integers that appear more than 2 times. In this case, 1 appears 3 times and 2 appears 3 times, so the answer is `[1, 2]`.

Maximum Number of Integers in the Answer

The answer will have a maximum of two integers. If n is 8, then $floor(n/3)$ is 2. Any element appearing more than two times must appear at least three times. If we have three integers each appearing three times, that's nine integers, but n is only eight. Therefore, at most, only two elements can appear greater than $floor(n/3)$ times.

Brute Force Solution

The brute force solution involves these steps:

1. Create an empty list.
2. Iterate through the array.
3. For each element, check if it's already in the list.
4. If not, count how many times it appears in the array.
5. If the count is greater than $floor(n/3)$, add it to the list.
6. If the list has two elements, stop.

Here's the pseudo code:

```
list = []
for i = 0 to n-1:
    if list is empty or list[0] != nums[i]:
        count = 0
        for j = 0 to n-1:
            if nums[j] == nums[i]:
                count++
        if count > n/3:
            list.add(nums[i])
    if list.size() == 2:
        break
return list
```

Time Complexity: $O(n^2)$

Space Complexity: $O(1)$

Better Solution (Hashing)

To improve from $O(n^2)$, we can use a hash map to keep track of how many times each number appeared.

1. Create an empty hash map.
2. Iterate through the array and update the count of each number in the hash map.
3. Iterate through the hash map and add numbers appearing more than $floor(n/3)$ times to a list.

Time Complexity: $O(n)$

Space Complexity: $O(n)$

Finding Majority Elements in an Array

Initial Setup and Iteration

- We're aiming to find elements that appear more than `floor(n/3)` times in an array.
- The key is to iterate through the array once, keeping track of counts and potential candidates.

Hash Map Approach

- Use a hash map to store elements and their frequencies.
- For each element:
  - If it's not in the map, add it with a count of 1.
  - If it's already in the map, increment its count.
- Crucially, the moment an element's count exceeds `n/3`, add it to the answer list.
- If the count increases beyond the minimum required (e.g., to 4 when 3 is the threshold), no need to re-add it.

Pseudocode

```
list answer;
map elementCounts;

for i from 0 to n-1:
    elementCounts[array[i]]++; // If element doesn't exist, it's initializ

    if elementCounts[array[i]] == floor(n/3) + 1:
        answer.add(array[i]);
```

- If an element isn't in the map, C++ automatically initializes its count to 0 before incrementing.

Time and Space Complexity

- Time complexity: $O(n)$ for iteration, plus potentially $O(logn)$ per map operation (or $O(1)$ for unordered maps).
- Using `unordered_map` can lead to $O(1)$ time complexity on average, but $O(n)$ in the worst case.
- Space complexity: $O(n)$ in the worst case, if all elements are distinct.

Optimization Note

- The interviewer might push for optimizing space complexity, as using a hash map can take $O(n)$ space if all elements are unique.

Code Submission Tips

- If submitting the "better solution," use the provided link.
- Be mindful of output requirements; the example asks for a sorted array.
- Sorting adds $O(klogk)$ time where k is the size of the array. In this example it is capped at size 2, so sorting only adds O(1) = constant time

Optimal Solution Intuition

- Drawing from the "greater than n/2 times" problem, the core idea is a cancellation logic.
- If an element appears a certain number of times, other elements must appear collectively more times to cancel it out.
- Adapt this logic for "greater than `floor(n/3)` times."

Adapting the Algorithm

- For `n/2`, you track one potential answer.
- For `n/3`, you can have up to two possible answers.
    - Use two counters (`count1`, `count2`) and two element holders (`element1`, `element2`).
- Iterate through the array:
    - If a counter is 0, set the corresponding element holder to the current element.
    - If the current element matches an element holder, increment its counter.
    - If it matches neither, decrement both counters.

Algorithm (Attempt 1 - Incorrect)

```
int count1 = 0, count2 = 0;
int element1, element2;

for i from 0 to n-1:
    if count1 == 0:
        count1 = 1;
        element1 = nums[i];
    else if count2 == 0:
        count2 = 1;
        element2 = nums[i];
    else if element1 == nums[i]:
        count1++;
    else if element2 == nums[i]:
        count2++;
    else:
        count1--;
        count2--;
```

Why the Initial Algorithm Fails

- The first attempt doesn't work due to a subtle flaw in how elements are assigned when counters are zero.
- If `element2` already holds a valid candidate, you shouldn't overwrite it just because `count1` is zero.
- The algorithm incorrectly reassigns `element1` or `element2` even when they hold valid candidates, leading to incorrect cancellations.

Optimal Solution for Finding Majority Elements

Here's an optimal solution to identify majority elements in an array, focusing on efficiency and clarity.

Edge Cases

Ensure that when considering an element as a potential majority element, it is not equal to the other potential majority element. For instance, if `nums[i]` is considered as `element1`, verify that it's not equal to `element2`, and vice versa.

```
nums[i] != element2
nums[i] != element1
```

Algorithm Steps

1. Initialization:
    - Initialize `count1` and `count2` to 0. These will track the counts of the potential majority elements.
    - Initialize `element1` and `element2` to `Integer.MIN_VALUE`. This assumes that the minimum integer value will not be present in the array, making them safe initial placeholders.
2. Iteration: Iterate through the array `V` from index 0 to its end.
    - If `counter1` is 0 and `V[i]` is not equal to `element2`, set `counter1` to 1 and `element1` to `V[i]`.
    - Else if `element1` is not equal to `V[i]`, set `counter2` to 1 and `element2` to `V[i]`.
    - Else if `V[i]` is equal to `element1`, increment `counter1`. If `V[i]` is equal to `element2`, increment `counter2`.
    - If none of the above conditions are met, decrement both `counter1` and `counter2`.
3. Manual Check:
    - Reinitialize `count1` and `count2` to 0.
    - Iterate through the array again to count the actual occurrences of `element1` and `element2`.
4. Verification: Check if `count1` and `count2` are greater than or equal to `n / 3 + 1`, where `n` is the size of the array.
    - If the condition is met, add the respective element to the answer list.
5. Sorting: Sort the answer list.
6. Return: Return the sorted list of majority elements.

Code Implementation

```cpp
vector<int> majorityElement(vector<int>& V) {
    int count1 = 0, count2 = 0;
    int element1 = INT_MIN, element2 = INT_MIN;
    int n = V.size();

    for (int i = 0; i < n; ++i) {
        if (count1 == 0 && element2 != V[i]) {
            count1 = 1;
            element1 = V[i];
        } else if (count1 != 0 && element1 != V[i] && count2 == 0) {
            count2 = 1;
            element2 = V[i];
        } else if (V[i] == element1) {
            count1++;
        } else if (V[i] == element2) {
            count2++;
        } else {
            count1--;
            count2--;
        }
    }

    vector<int> ans;
    count1 = 0;
    count2 = 0;

    for (int i = 0; i < n; ++i) {
        if (element1 == V[i]) {
            count1++;
        }
        if (element2 == V[i]) {
            count2++;
        }
    }

    if (count1 > n / 3) {
        ans.push_back(element1);
    }
    if (count2 > n / 3 && element1 != element2) {
        ans.push_back(element2);
    }

    sort(ans.begin(), ans.end());
    return ans;
}
```

Time and Space Complexity

Category	Complexity
Time Complexity	$O(2n)$
Space Complexity	$O(1)$