Majority Element

This study guide explains how to find the majority element in an array of integers, where the majority element appears more than `n / 2` times.

Problem Statement

Given an array of integers, find the element that appears more than `n / 2` times, where `n` is the length of the array.

For example, in the array `[2, 2, 3, 2, 2, 1, 2]`, the majority element is `2` because it appears 4 times, which is more than `n / 2` (7 / 2 = 3.5, rounded down to 3).

Brute Force Solution

The brute force approach involves picking an element and scanning through the entire array to count its occurrences.

1. Iterate through the array.
2. For each element, scan the entire array to count its occurrences.
3. If the count is greater than `n / 2`, return that element.
4. If no majority element is found, return -1.

```
for (int i = 0; i < array.size(); i++) {
    int count = 0;
    for (int j = 0; j < array.size(); j++) {
        if (array[i] == array[j]) {
            count++;
        }
    }
    if (count > array.size() / 2) {
        return array[i];
    }
}
return -1;
```

- **Time Complexity:** $O(n^2)$ due to nested loops.

Better Solution: Hashing

A better solution involves using hashing to count the occurrences of each element.

1. Declare a hashmap to store elements as keys and their counts as values.
2. Iterate through the array and update the count of each element in the hashmap.
3. Iterate through the hashmap and check if any element's count is greater than `n / 2`.

```
#include <iostream>
#include <vector>
#include <map>

int majorityElement(std::vector<int>& nums) {
    std::map<int, int> elementCount;
    for (int num : nums) {
        elementCount[num]++;
    }
    for (auto const& [key, val] : elementCount) {
        if (val > nums.size() / 2) {
            return key;
        }
    }
    return -1;
}
```

- Time Complexity: $O(n \log n)$ because the first loop is $O(n)$ and the second loop depends on the map implementation (usually logarithmic). If an unordered map is used, the average and best-case time complexity can be $O(n)$, but the worst-case remains $O(n)$.

- Space Complexity: $O(n)$ because, in the worst case (all unique elements), we store all elements in the hashmap.

Optimal Solution: Moore's Voting Algorithm

The most optimal solution is Moore's Voting Algorithm, which finds the majority element in linear time with constant space.

1. Initialize two variables: `element` and `count`. Initially, `count = 0` and `element` is uninitialized.
2. Iterate through the array.
   - If `count == 0`, assign the current element to `element` and set `count = 1`.
   - If the current element is equal to `element`, increment `count`.
   - If the current element is not equal to `element`, decrement `count`.
3. After the first pass, the `element` variable will hold the potential majority element.
4. Perform a second pass through the array to verify that `element` appears more than `n / 2` times.

```cpp
int majorityElement(std::vector<int>& nums) {
    int element = 0;
    int count = 0;
    for (int num : nums) {
        if (count == 0) {
            element = num;
            count = 1;
        } else if (num == element) {
            count++;
        } else {
            count--;
        }
    }
    int frequency = 0;
    for (int num : nums) {
        if (num == element) {
            frequency++;
        }
    }
    if (frequency > nums.size() / 2) {
        return element;
    }
    return -1;
}
```

Here's how the algorithm works with an example array `[7, 7, 5, 7, 5, 1, 5]`:

1. Initialize `element = 7` and `count = 1`.

2. Iterate through the array:

   - `7`: `count` increments to 2.
   - `5`: `count` decrements to 1.
   - `7`: `count` increments to 2.
   - `5`: `count` decrements to 1.
   - `1`: `count` decrements to 0. Since `count` is 0, `element` becomes `5` and `count` becomes 1.
   - `5`: `count` increments to 2.

3. The potential majority element is 5.

4. Verify 5's frequency.

- **Time Complexity:** $O(n)$ because it iterates through the array twice.
- **Space Complexity:** $O(1)$ because it uses only constant extra space.

Intuition Behind Moore's Voting Algorithm

The Moore's Voting Algorithm works on the principle that if a majority element exists (i.e., an element that appears more than `n / 2` times), it will always be the last surviving element after canceling out other elements.

> **Metaphor**: Think of it like an election where voters supporting different candidates pair off and cancel each other out. The candidate with more than half the votes will always have some supporters left over.

If `count` reaches zero, it means that up to that point, no element has appeared more than `n / 2` times, and thus all elements encountered so far have effectively canceled each other out.

Consider a portion of the array `[7, 7, 5, 7, 5, 1]`. Here, 7 appears three times, and the other elements (5 and 1) appear three times in total. The count becomes zero, signifying that 7 is not a majority element in this subarray because it got canceled out.

If an element appears more than `n / 2` times, it cannot be fully canceled out by other elements in the array. Therefore, the algorithm correctly identifies the majority element in linear time and constant space.

Moore's Voting Algorithm Walkthrough

Let's dive into how Moore's Voting Algorithm works with an example array:

Initially, let's say we have an array and we want to find the majority element (if it exists).

The algorithm iterates through the array, keeping track of a `count` and a potential `element`.

Iteration Example

1. Start with `count = 0`.
2. Move through the array:

- If `count` is `0`, set the current element as the potential majority element and set `count = 1`.
- If the current element is the same as the potential majority element, increment `count`.
- If the current element is different, decrement `count`.
- If `count` returns to zero, that means that the majority has been cancelled out by an equal number of non-majority elements.

Example Array Explained

Consider the array `[5, 5, 7, 7, 5, 5, 5, 5, 7, 7, 5, 5, 1, 1, 1]`.

1. Start with 5, `count = 1`.
2. Next 5, `count = 2`.
3. Encounter 7, `count = 1`.
4. Encounter 7, `count = 0`. Since count is zero we will initialize the count to 1 again and the element to be 5, restarting the array check from that position onwards.
5. The 5's and 7's cancel each other out, setting `count` back to `0`.
6. The algorithm continues, re-initializing `count` and the potential majority element as needed.

If, after the first pass, `count` is greater than `0`, the remaining `element` is a potential majority element.

Verifying the Result

After applying Moore's Voting Algorithm, it's crucial to verify if the potential majority element is indeed a majority. To do this, iterate through the array and count the occurrences of the potential majority element. If it appears more than `n/2` times (where `n` is the length of the array), it is indeed the majority element.

For example, if the potential majority element is `5`, count how many times `5` appears in the array. If it appears more than `n/2` times, `5` is the majority element.

Intuition Behind the Algorithm

> The intuition behind Moore's Voting Algorithm is that if a majority element exists (appears more than `n/2` times), it will not be canceled out by other elements.

In other words, the majority element will always have a count greater than `0` at the end of the first pass (or after being re-initialized).

Code Implementation

Here's how you can implement Moore's Voting Algorithm:

```cpp
int findMajorityElement(vector<int>& v) {
    int count = 0;
    int element;

    for (int i = 0; i < v.size(); ++i) {
        if (count == 0) {
            count = 1;
            element = v[i];
        } else if (v[i] == element) {
            count++;
        } else {
            count--;
        }
    }

    int counterOne = 0;
    for (int i = 0; i < v.size(); ++i) {
        if (v[i] == element) {
            counterOne++;
        }
    }

    if (counterOne > v.size() / 2) {
        return element;
    } else {
        return -1; // or handle the case where no majority element exists
    }
}
```

Time and Space Complexity

- Time Complexity: $O(n)$ because it involves a single loop through the array. If you need to verify the element, it will be $O(2n)$, which simplifies to $O(n)$.
- Space Complexity: $O(1)$ because it uses a constant amount of extra space (just a few variables).

Problem Conditions

- If the problem states that a majority element always exists, you only need to perform the first pass of Moore's Voting Algorithm.
- If the problem states that a majority element may or may not exist, you need to verify the potential majority element after the first pass.