169. Majority Element

Easy Array Hash Table Divide and Conquer Counting Sorting

Problem Description

You are given an array called `nums` that has a certain number of elements, denoted as `n`. You have to find the **majority element** in this array. The majority element is defined as the one that appears more than `n / 2` times. It's guaranteed that there is always a majority element in the array you're given.

Intuition

The intuition behind the solution comes from the understanding that if an element occurs more than `n / 2` times, it means that element is present more often than all other elements combined. For every instance of a non-majority element, there must be more instances of the majority element. We can use this intuition with a clever algorithm known as the **Moore Voting Algorithm**.

The Moore Voting Algorithm works by keeping a current candidate for majority element and a counter. As it goes through the array, the algorithm either increases the count if the current number matches the candidate or decreases the count otherwise. When the count reaches zero, it means that up to that point, there is no majority element, and the algorithm selects the current number as the new candidate. The key insight is that the majority element's surplus count will withstand the count decrements due to non-majority elements.

Therefore, we initialize a counter `cnt` to zero and a majority element candidate `m` to none. Then, for each element `x` in `nums`, we make the following check:

- If `cnt` is zero, we assume the current element `x` can be a new majority candidate `m`, and we set `cnt` to 1.
- If `cnt` is not zero, we increase the count by 1 if `x` is equal to `m` (our current candidate), otherwise, we decrease the count.

After processing all elements of the array, our current candidate `m` is the majority element. Since we are guaranteed that a majority element exists, we don't need to verify `m` on a second pass.

Solution Approach

The solution uses the Moore Voting Algorithm, which is efficient in finding the majority element of an array when such an element definitely exists.

The algorithm works in a single pass and uses constant extra space. It consists of two main variables: the candidate `m` representing the presumed majority element, and the counter `cnt` indicating the strength of our presumption that `m` is indeed the majority element.

To implement the algorithm, we iterate through each element `x` of the array `nums`. During this iteration, the following logic is applied:

1. When `cnt` equals zero, there's no current candidate for majority element, or the previous candidate has been completely offset by other elements. So we assign the current element `x` to be the new candidate `m`, and set `cnt` to 1, because we start counting the occurrences of `m` again.
2. If `cnt` is not zero, it means there's a candidate set and we need to compare the current element `x` with `m`.
   - If `x` is equal to `m`, this means we've found another instance of our candidate, and we increment the counter `cnt` by 1, strengthening the candidacy of `m`.
   - If `x` is not equal to `m`, this means we have encountered an element that opposes our candidate. To denote this opposition, we decrement the counter `cnt` by 1.

By the end of the loop, despite all the increments and decrements, the surplus repetitions of the majority element ensure that `m` will remain as the candidate and `cnt` will be greater than zero.

Given the guarantee that a majority element always exists, `m` is the majority element at the end of this single pass, and we can return `m` as the answer.

It's important to note that if the problem statement didn't guarantee the existence of a majority element, a second pass would be necessary to confirm that our candidate `m` is indeed the majority by counting its total occurrences in `nums` and comparing it to `n / 2`.

Example Walkthrough

Assume we have an array `nums = [3, 3, 1, 3, 2, 3]`. We want to use the Moore Voting Algorithm to find the majority element. The majority element is the element that appears more than `n / 2` times in the array (`n` is the size of the array which in this case is 6, so `n / 2` is 3). The steps are as follows:

1. We initialize `cnt` to 0 and `m` to any value (let's choose `m = None` for the start).
2. Traverse the elements of `nums`:

- Start with the first element (`nums[0] = 3`).
  - Since `cnt` is 0, we set `m = 3` and `cnt = 1`.
- Move to the second element (`nums[1] = 3`).
  - Since `cnt` is not 0 and `m` is equal to `nums[1]`, we increment `cnt` to 2.
- Proceed to the third element (`nums[2] = 1`).
  - The current element is not equal to `m`, so we decrement `cnt` to 1.
- Next, the fourth element (`nums[3] = 3`).
  - The current element is equal to `m`, therefore `cnt` is incremented to 2.
- Then, the fifth element (`nums[4] = 2`).
  - This is not equal to `m`, and so `cnt` is decremented to 1.
- Finally, the sixth element (`nums[5] = 3`).
  - It is equal to `m`, meaning `cnt` gets incremented again, now `cnt = 2`.

3. Despite the increment and decrement of `cnt`, at the end of the traversal, `m` remains `3`. Since `cnt` is greater than 0, we are left with `m = 3` as the candidate for the majority element.
4. As the problem guarantees the presence of a majority element, we don't need to check `m` in a second pass. We declare that the majority element is `3`.

By the end of the process, we successfully used the Moore Voting Algorithm to find the majority element which is `3` in the given example array.

Python Solution

```python
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        # Initialize the count and the candidate for majority element
        count = 0
        majority_candidate = None

        # Process each number in the list
        for num in nums:
            # If the current count is 0, we choose a new number as the potential majority candidate
            if count == 0:
                majority_candidate = num
                count = 1
            # If the current number is the same as the majority candidate, increase the count
            elif majority_candidate == num:
                count += 1
            # Otherwise, decrease the count
            else:
                count -= 1

        # The majority candidate is the number that remains after pairing off different elements
        return majority_candidate

```

Java Solution

```java
class Solution {

    // This method finds the majority element in an array, which is defined as the element that appears more than n/2 times
    public int majorityElement(int[] nums) {
        // Initialize count and candidate for majority element
        int count = 0;
        int candidate = 0;

        // Iterate over all elements in the array
        for (int num : nums) {
            // If count is zero, we choose the current element as the new candidate
            if (count == 0) {
                candidate = num;
                count = 1;
            } else {
                // If the current element is the same as the candidate, increment count
                if (num == candidate) {
                    count++;
                } else {
                    // If different, decrement count
                    count--;
                }
            }
        }

        // The candidate is the majority element, which is guaranteed to exist
        return candidate;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the vector header for using the vector container

class Solution {
public:
    // Function to find the majority element in the array
    // A majority element is an element that appears more than n/2 times in the array
    int majorityElement(vector<int>& nums) {
        int count = 0; // Counter for the number of times the current candidate is found
        int candidate = 0; // Variable to store the current potential majority element

        // Iterate through all the elements in the given array
        for (int num : nums) {
            if (count == 0) {
                // If the count is 0, we select the current element as our new candidate for majority element
                candidate = num;
                count = 1; // Set the count for this new candidate to 1
            } else {
                // If count is not 0, increment or decrement the count
                // Increment if the current element is the same as the candidate
                // Decrement if it is different
                count += (candidate == num) ? 1 : -1;
            }
        }

        // After the loop, the candidate is the majority element
        // (due to the problem's guarantee that a majority element always exists)
        return candidate;
    }
};

```

Typescript Solution

```typescript
function majorityElement(numbers: number[]): number {
    // Initialize a count variable to keep track of the frequency of the majority element.
    let count: number = 0;
    // Initialize a variable to hold the current majority element.
    let majorityElement: number = 0;

    // Iterate through each number in the numbers array.
    for (const number of numbers) {
        // If count is zero, we found a new possible majority element.
        if (count === 0) {
            majorityElement = number;
            count = 1;
        }
        // If the current number is the same as the majority element, increment the count.
        // Otherwise, decrement the count.
        else {
            count += (majorityElement === number) ? 1 : -1;
        }
    }

    // At the end of the loop, the majorityElement variable will contain the majority element.
    return majorityElement;
}

```

Time and Space Complexity

The given Python code snippet is an implementation of the Boyer-Moore Voting Algorithm, designed to find a majority element (an element that appears more than `n/2` times) from a list. The time complexity and space complexity of this algorithm are analyzed as follows:

Time Complexity:

The function iterates through the list `nums` exactly once. For each element `x` in `nums`, the code executes a constant number of operations, either incrementing, decrementing, or setting the count `cnt` and possibly updating the candidate majority element `m`. Thus, the number of operations is proportional to the length of `nums`, which is `n`. Therefore, the time complexity is `O(n)`.

Space Complexity:

The algorithm uses a fixed amount of extra space: two variables `cnt` and `m` to keep track of the current count and the potential majority element respectively. The amount of space used does not depend on the size of the input list, therefore, the space complexity is constant, `O(1)`.