Problem: Majority Element

Given an array nums of size n, the task is to return the majority element.

The majority element is defined as the element that appears more than ⌊n / 2⌋ times in the array. You can assume that the majority element always exists in the input array.

Example 1:

- **Input:** nums = [3, 2, 3]
- **Output:** 3

Example 2:

- **Input:** nums = [2, 2, 1, 1, 1, 2, 2]
- **Output:** 2

Constraints:

- n == nums.length
- 1 <= n <= 5 * 10^4
- -10^9 <= nums[i] <= 10^9

Follow-up: Could you solve the problem in linear time and O(1) space?

Solution Explanation

- To solve the problem of finding the majority element in an array where the majority element is defined as the element that appears more than ⌊n / 2⌋ times, we need an efficient approach that satisfies the constraints.

1. Understanding the Problem Requirements:

- The majority element is the one that appears more than half the time in the array. For example, if the length of the array n is 7, the majority element must appear more than ⌊7 / 2⌋ = 3 times.
- The problem guarantees that a majority element always exists.

2. Approaches to Solve the Problem:

- ***Brute Force:***
  - *Time Complexity:* $O(n^2)$
  - *Space Complexity:* O(1)
  - Check the count of each element in the array and see if it appears more than ⌊n / 2⌋ times. This approach is inefficient due to its time complexity.

- ***Using a Hash Map:***
  - *Time Complexity:* O(n)
  - *Space Complexity:* O(n)
  - Use a hash map to count the occurrences of each element. The element with a count greater than ⌊n / 2⌋ is the majority element. While this approach meets the time complexity requirement, it does not meet the space complexity constraint of O(1).

- ***Sorting:***
  - *Time Complexity:* O(n log n)
  - *Space Complexity:* O(1) or O(n) depending on the sorting algorithm
  - By sorting the array, the majority element will be at the middle index ⌊n / 2⌋. This approach is more efficient than brute force but does not meet the linear time constraint.

- ***Boyer-Moore Voting Algorithm:***
  - *Time Complexity:* O(n)
  - *Space Complexity:* O(1)
  - This algorithm is optimal for solving the problem within the given constraints. It works by maintaining a candidate for the majority element and a counter to track the number of votes the candidate has.

3. Boyer-Moore Voting Algorithm:

The Boyer-Moore Voting Algorithm is a powerful algorithm to solve this problem in linear time and constant space. Here's how it works:

- ***Initialization:***
  - Start with a candidate set to None and a count set to 0.

- ***Iterating through the Array:***
  - *For each element in the array:*
    - ✓ If the count is 0, set the current element as the candidate.
    - ✓ If the current element is the same as the candidate, increment the count.
    - ✓ If the current element is different from the candidate, decrement the count.

- ***Final Result:***
  - ➢ The candidate at the end of the iteration will be the majority element, as the algorithm effectively "cancels out" elements that are not the majority.

4. Detailed Example Walkthrough:

- ***Example 1:***
  - ➢ *Input:* nums = [3, 2, 3]
  - ➢ *Process:*
    - ✓ Start with candidate = None, count = 0.
    - ✓ First element is 3: count becomes 1, candidate = 3.
    - ✓ Second element is 2: count becomes 0, change candidate to None.
    - ✓ Third element is 3: count becomes 1, candidate = 3.
  - ➢ *Output:* 3

- ***Example 2:***
  - ➢ *Input:* nums = [2, 2, 1, 1, 1, 2, 2]
  - ➢ *Process:*
    - ✓ Start with candidate = None, count = 0.
    - ✓ First element is 2: count becomes 1, candidate = 2.
    - ✓ Second element is 2: count becomes 2, candidate = 2.
    - ✓ Third element is 1: count becomes 1, candidate = 2.
    - ✓ Fourth element is 1: count becomes 0, change candidate to None.
    - ✓ Fifth element is 1: count becomes 1, candidate = 1.
    - ✓ Sixth element is 2: count becomes 0, change candidate to None.
    - ✓ Seventh element is 2: count becomes 1, candidate = 2.
- ➢ *Output:* 2

5. Conclusion:

- The Boyer-Moore Voting Algorithm is the most efficient solution for finding the majority element in an array, meeting both the linear time complexity and constant space complexity requirements. This algorithm is optimal for problems where a majority element is guaranteed to exist and requires only a single pass through the array to determine the result.