<https://leetcode.com/problems/majority-element/description/>

Given an array nums of size n, return *the majority element*.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the 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 in O(1) space?

**Attempt 1: 2025-04-09**

**Refer to Deepseek**

**Problem Description**

Given an array nums of size n, return the majority element. The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

**Approaches**

**1. Hash Map (Count Frequencies)**

Use a hash map to count the frequency of each element.

Return the element with frequency > n/2.

**2. Sorting**

Sort the array.

The majority element will always be at index n/2.

**3. Boyer-Moore Voting Algorithm (Optimal)**

Initialize a candidate and a count.

Traverse the array:

If count is 0, set current element as candidate.

If current element == candidate, increment count.

Else decrement count.

The candidate will be the majority element.

**Solution Code (Boyer-Moore Voting Algorithm)**

class Solution {

public int majorityElement(int[] nums) {

int count = 0;

Integer candidate = null;

for (int num : nums) {

if (count == 0) {

candidate = num;

}

count += (num == candidate) ? 1 : -1;

}

return candidate;

}

}

**Explanation**

**Boyer-Moore Voting Algorithm**

Initialization: Start with count = 0 and no candidate.

First Pass:

When count is 0, we set the current number as our candidate.

For each subsequent number:

If it matches the candidate, increment count.

If it doesn't match, decrement count.

Result: The candidate remaining at the end is guaranteed to be the majority element.

**Why It Works**

**The algorithm works because the majority element occurs more than n/2 times.**

**Even if the majority element is canceled out by other elements in some parts of the array, it will still have a net positive count at the end.**

**Time and Space Complexity**

Time Complexity: O(n) - Single pass through the array.

Space Complexity: O(1) - Uses constant extra space.

**Alternative Solutions**

**1. Hash Map Solution**

class Solution {

public int majorityElement(int[] nums) {

Map<Integer, Integer> counts = new HashMap<>();

for (int num : nums) {

counts.put(num, counts.getOrDefault(num, 0) + 1);

if (counts.get(num) > nums.length / 2) {

return num;

}

}

return -1;

}

}

**2. Sorting Solution**

class Solution {

public int majorityElement(int[] nums) {

Arrays.sort(nums);

return nums[nums.length / 2];

}

}

The Boyer-Moore algorithm is preferred for its O(n) time and O(1) space complexity, while the hash map solution uses O(n) space and the sorting solution has O(n log n) time complexity.

**Refer to**

[L229.Majority Element II (Ref.L169)](note://WEB19bfa59a8b961fd8917415c984dc8b48)

[L2404.Most Frequent Even Element (Ref.L451)](note://WEBe76f4087aec16b39aa766194657a60d9)