Majority Element

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
```

Approach 1: Brute Force

Intuition

We can exhaust the search space in quadratic time by checking whether each element is the majority element.

Algorithm

The brute force algorithm iterates over the array, and then iterates again for each number to count its occurrences. As soon as a number is found to have appeared more than any other can possibly have appeared, return it.

```
public int majorityElement(int[] nums) {

    int majorityCount = nums.length/2;


    for (int num : nums) {

        int count = 0;

        for (int elem : nums) {

            if (elem == num) {

                count += 1;

            }

        }
```

```
            if (count > majorityCount) {

                return num;

            }


        }

        return -1;

    }
```

Complexity Analysis

- Time complexity : $O(n^2)$
- Space complexity : $O(1)$

Approach 2: HashMap

Intuition

We know that the majority element occurs more n}{2} \ ⌊2n⌋ times, and a `HashMap` allows us to count element occurrences efficiently.

Algorithm

We can use a `HashMap` that maps elements to counts in order to count occurrences in linear time by looping over `nums`. Then, we simply return the key with maximum value.

Complexity Analysis

- Time complexity : $O(n)$
- Space complexity : $O(n)$

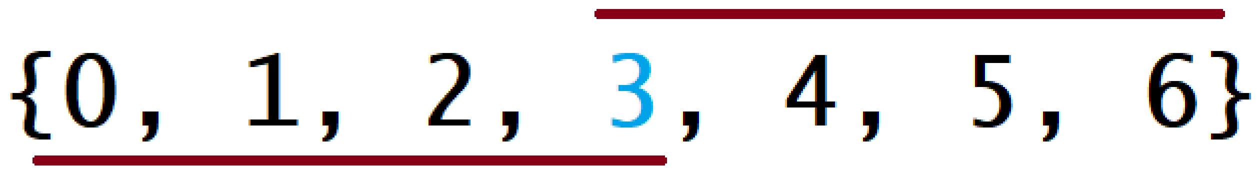
Approach 3: Sorting

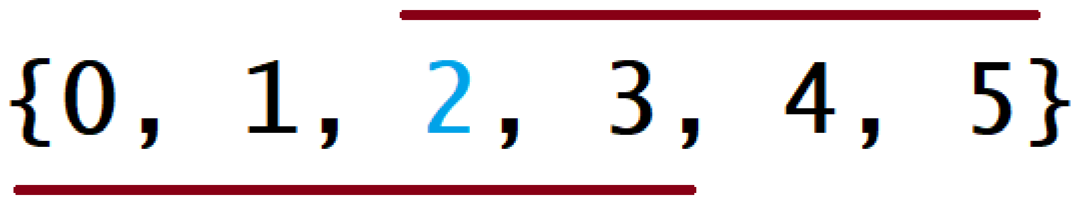
Intuition

If the elements are sorted in monotonically increasing (or decreasing) order, the majority element can be found at index {n}/{2 (and also {n}/{2} - 1, if n is even).

Algorithm

For this algorithm, we simply do exactly what is described: sort `nums`, and return the element in question. To see why this will always return the majority element (given that the array has one), consider the figure below (the top example is for an odd-length array and the bottom is for an even-length array):





```
class Solution {
   public int majorityElement(int[] nums) {
      Arrays.sort(nums);
      return nums[nums.length/2];
   }
}
```

Complexity Analysis

- Time complexity : $O(nlgn)$
- Space complexity : $O(1)$ or ($O(n)$

Approach 4: Boyer-Moore Voting Algorithm

Intuition

If we had some way of counting instances of the majority element as $+1$ and instances of any other element as -1 summing them would make it obvious that the majority element is indeed the majority element.

```
public int majorityElement(int[] nums) {
      int count=1;
```

```
        int majorityElement=nums[0];

        for(int i=1;i<nums.length;i++)

        {

            if(count==0)

            {

                majorityElement=nums[i];

            }

            if(nums[i]==majorityElement)

            {

                count++;

            }

            else

            {

                count--;

            }

        }

        return majorityElement;

    }
```

Complexity Analysis

- Time complexity : $O(n)$
- Space complexity : $O(1)$