Given an array of size *n*, find the majority element. The majority element is the element that appears **more than** ⌊ n/2 ⌋ times.

You may assume that the array is non-empty and the majority element always exist in the array.

**Example 1:**

**Input:** [3,2,3]

**Output:** 3

**Example 2:**

**Input:** [2,2,1,1,1,2,2]

**Output:** 2

给你一个array，有一个数重复了n/2多次，找到这个数

方法1：弄一个hashMap，把数作为key，次数作为value，刷新就好， 然后再找到value大于一半的key，

耗时为O（n）

方法2：给array排序，最后中位数必然是我们要的数，因为次数比n/2更多

class Solution {

public int majorityElement(int[] nums) {

Arrays.sort(nums);

return nums[nums.length/2];

}

}

array自带sort

方法3：Conquer and divide:

class Solution {

private int countInRange(int[] nums, int num, int lo, int hi) { //输入数组，要查的数，开头，结尾

int count = 0;

for (int i = lo; i <= hi; i++) {

if (nums[i] == num) {

count++;

}

}

return count;

}

private int majorityElementRec(int[] nums, int lo, int hi) {

// base case; the only element in an array of size 1 is the majority

// element.

if (lo == hi) { //开头等于结尾，return这个元素，也就是分治法永远的第一步，

return nums[lo];

}

// recurse on left and right halves of this slice.

int mid = (hi-lo)/2 + lo; //找到中位数

int left = majorityElementRec(nums, lo, mid); left循环

int right = majorityElementRec(nums, mid+1, hi); right循环

// if the two halves agree on the majority element, return it.

if (left == right) { //如果左边的majority数等于右边的majority数，随便return随便一个

return left;

} //例如只剩两个的时候，就默认return左边的

// otherwise, count each element and return the "winner".

int leftCount = countInRange(nums, left, lo, hi); 计算majority数的次数 //你怎么知道哪一个是majority数呢？通过前面的left,rec 注意了，这里是lo,hi，而不是中位数，要遍历当前整个array

int rightCount = countInRange(nums, right, lo, hi);

return leftCount > rightCount ? left : right; 哪边大return哪个

}

public int majorityElement(int[] nums) {

return majorityElementRec(nums, 0, nums.length-1);

}

}

例如323，

mid位1

分成32 与3

第一个左边是3， 第二个是3，相等，return 3

又如32212

分为322， 12

然后 32 2 1 2

32得到3， 右边是2，

不一样，分别数，得到2更多，因此322是2，

右边是1，

不一样，整个遍历，2更多

我自己写的答案

class Solution {

public int majorityElement(int[] nums) {

return majRec(nums,0,nums.length-1);

}

private static int count(int[] nums,int number, int start,int end){

int counter=0;

for(int i=start;i<=end;i++){ //注意这里是小于等于，因为end已经是index了

if (nums[i]==number) counter++;

}

return counter;

}

private static int majRec(int[] nums,int start,int end){

if(start==end) return nums[start];

int mid=(end-start)/2+start;

int left=majRec(nums,start,mid);

int right=majRec(nums,mid+1,end);

if(left==right) return left;

int leftCount=count(nums,left,start,end);

int rightCount=count(nums,right,start,end);

if (leftCount>rightCount) return left;

else return right;

}

}

方法4：Boyer-Moore, 也是这题最优解

该算法时间复杂度为O(n)，空间复杂度为O(1)，

我们需要记录2个值：

candidate，初值可以为任何数

count，初值为0

之后，对于数组中每一个元素，首先判断count是否为0，若为0，则把candidate设置为当前元素。之后判断candidate是否与当前元素相等，若相等则count+=1，否则count-=1。

原理解析

举个例子，我们的输入数组为[1,1,0,0,0,1,0]，那么0就是多数元素。

首先，candidate被设置为第一个元素1，count也变成1，由于1不是多数元素，所以当扫描到数组某个位置时，count一定会减为0。在我们的例子中，当扫描到第四个位置时，count变成0.

count 值变化过程：

[1,2,1,0……

当count变成0时，对于每一个出现的1，我们都用一个0与其进行抵消，所以我们消耗掉了与其一样多的0，而0是多数元素，这意味着当扫描到第四个位置时，我们已经最大程度的消耗掉了多数元素。 这时不会中断for循环，因为count又一次等于0了，因此第五个数就是我们新candidate候选人

然而，对于数组从第五个位置开始的剩余部分，0依然是其中的多数元素(注意，多数元素出现次数大于⌊ n/2 ⌋，而我们扫描过的部分中多数元素只占一般，那剩余部分中多数元素必然还是那个数字)。如果之前用于抵消的元素中存在非多数元素，那么数组剩余部分包含的多数元素就更多了。

//解释：哪怕你前面全部都是用majority抵消，majority后面还是占多数，因为majority大>n/2，

类似的，假设第一个数字就是多数元素，那么当count减为0时，我们消耗掉了与多数元素一样多的非多数元素，那么同样道理，数组剩余部分中的多数元素数值不变。

这两种情况证明了关键的一点：数组中从candidate被赋值到count减到0的那一段可以被去除，余下部分的多数元素依然是原数组的多数元素。我们可以不断重复这个过程，直到扫描到数组尾部，那么count必然会大于0，而且这个count对应的candinate就是原数组的多数元素。

//注意：这个方法只能用于most数大于一半的情况下，例如12121212000，最后剩下的事0，但其实12最多，因为（注意，多数元素出现次数大于⌊ n/2 ⌋，而我们扫描过的部分中多数元素只占一般，那剩余部分中多数元素必然还是那个数字)

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;

}

}