[6 Suggested Solutions in C++ with Explanations](https://discuss.leetcode.com/topic/17446/6-suggested-solutions-in-c-with-explanations)

Well, if you have got this problem accepted, you may have noticed that there are 7 suggested solutions for this problem. The following passage will implement 6 of them except the O(n^2) brute force algorithm.

**Hash Table（36ms）**

The hash-table solution is very straightforward. We maintain a mapping from each element to its number of appearances. While constructing the mapping, we update the majority element based on the max number of appearances we have seen. Notice that we do not need to construct the full mapping when we see that an element has appeared more than n / 2 times.

The code is as follows, which should be self-explanatory.

**class** Solution {

**public**:

**int** **majorityElement**(vector<**int**>& nums) {

unordered\_map<**int**, **int**> counts;

**int** n = nums.size();

**for** (**int** i = 0; i < n; i++)

**if** (++counts[nums[i]] > n / 2)

**return** nums[i];

}

};

**Sorting（22ms）**

Since the majority element appears more than n / 2 times, the n / 2-th element in the sorted nums must be the majority element. This can be proved intuitively. Note that the majority element will take more than n / 2 positions in the sorted nums (cover more than half of nums). If the first of it appears in the 0-th position, it will also appear in the n / 2-th position to cover more than half of nums. It is similar if the last of it appears in the n - 1-th position. These two cases are that the contiguous chunk of the majority element is to the leftmost and the rightmost in nums. For other cases (imagine the chunk moves between the left and the right end), it must also appear in the n / 2-th position.

The code is as follows, being very short if we use the system nth\_element (thanks for [@qeatzy](https://discuss.leetcode.com/uid/22324) for pointing out such a nice function).

**class** Solution {

**public**:

**int** majorityElement(vector<**int**>& nums) {

nth\_element(nums.begin(), nums.begin() + nums.size() / 2, nums.end());

return nums[nums.size() / 2];

}

};

**Randomization（19ms）**

This is a really nice idea and works pretty well (16ms running time on the OJ, almost fastest among the C++ solutions). The proof is already given in the suggested solutions.

The code is as follows, randomly pick an element and see if it is the majority one.

**class** Solution {

**public**:

**int** **majorityElement**(vector<**int**>& nums) {

**int** n = nums.size();

srand(**unsigned**(time(NULL)));

**while** (true) {

**int** idx = rand() % n;

**int** candidate = nums[idx];

**int** counts = 0;

**for** (**int** i = 0; i < n; i++)

**if** (nums[i] == candidate)

counts++;

**if** (counts > n / 2) **return** candidate;

}

}

};

**Divide and Conquer（26ms）**

This idea is very algorithmic. However, the implementation of it requires some careful thought about the base cases of the recursion. The base case is that when the array has only one element, then it is the majority one. This solution takes 24ms.

**class** Solution {

**public**:

int majorityElement(vector<int>& nums) {

return majority(nums, 0, nums.size() - 1);

}

**private**:

int majority(vector<int>& nums, int left, int right) {

**if** (left == right) return nums[left];

int mid = left + ((right - left) >> 1);

int lm = majority(nums, left, mid);

int rm = majority(nums, mid + 1, right);

**if** (lm == rm) return lm;

return count(nums.begin() + left, nums.begin() + right + 1, lm) > count(nums.begin() + left, nums.begin() + right + 1, rm) ? lm : rm;

}

};

**Moore Voting Algorithm（29ms）**

A brilliant and easy-to-implement algorithm! It also runs very fast, about 20ms.

**class** Solution {

**public**:

**int** **majorityElement**(vector<**int**>& nums) {

**int** major, counts = 0, n = nums.size();

**for** (**int** i = 0; i < n; i++) {

**if** (!counts) {

major = nums[i];

counts = 1;

}

**else** counts += (nums[i] == major) ? 1 : -1;

}

**return** major;

}

};

**Bit Manipulation（23ms）**

Another nice idea! The key lies in how to count the number of 1's on a specific bit. Specifically, you need a mask with a 1 on the i-the bit and 0 otherwise to get the i-th bit of each element in nums. The code is as follows.

**class** Solution {

**public**:

**int** **majorityElement**(vector<**int**>& nums) {

**int** major = 0, n = nums.size();

**for** (**int** i = 0, mask = 1; i < 32; i++, mask <<= 1) {

**int** bitCounts = 0;

**for** (**int** j = 0; j < n; j++) {

**if** (nums[j] & mask) bitCounts++;

**if** (bitCounts > n / 2) {

major |= mask;

**break**;

}

}

}

**return** major;

}

};

## 4. Median of Two Sorted Arrays

这是一道考验基本功的题目。

要想在O(log(m+n))时间复杂度内解决问题，必然用到分治思想，也必然是递归程序。但是一开始递归函数的功能想错了，因为中位数分奇偶性 ，所以想写一个返回中间两个数值然后按照奇偶性判断解。但是这样写的函数代码繁杂，边界条件较多，不能有效解决题目。然后参考了discuss代码，将递归函数功能改为返回第k大的数字，调用两次不改变时间复杂度，易实现。

但是如果边界条件考虑不到位还是会错。

1.分别考虑两个有序序列前k/2个，但是可能长度没有k/2;

2.可以假使一个长度小，另一个长。

3.k=1情况必须要判断，否则会越界，错误。

## 240. Search a 2D Matrix II

我的行列二分解法，还行，不过还是太笨了点；discuss区的牛逼解法，O（m+n）复杂度，而且代码实现简单。

对于这种比较有某种特性的一维或二维数组，可以从四个特定位置的边界角入手，寻找特性简化求解。

## 493. Reverse Pairs

分治法的一般分析方法：一般能用分治在有效时间内解决问题的不会超过O（nlogn） 的算法复杂度，也即：T（n)=T（n/2）+O（kn） （k是常数） ，从这个角度分析转换问题是一个突破口。

## 327. Count of Range Sum

程序是多么优美！编程的艺术！

边界测试：

-2147483647,0,-2147483647,2147483647

-564

3864

对于一般算法复杂度为 O（nlogn）的分治算法，一般只有两种形式：

1.T(n)=T(n-1)+O(logn)

2.T(n)=2T(n/2)+O(n);

一般从这两个递推式中去分析解题方法。

第一种方法一般需要借助二叉搜索树、红黑树或者树状数组、线段树。一般C++用multiset、multimap(可能会用到distance(begin,end)函数求个数)

而第二种当用于某种有序关系、范围关系时，归并排序的运用是一种经典思想。运用序列有序，可以充分挖掘有序性质，从而减少计算，使得在O(n)时间内能从序列中得到答案。

本题和315. Count of Smaller Numbers After Self 及 493. Reverse Pairs 思想一致。

但是本题还在于能从区间和范围联想到sum方法。同时注意程序的写法。

另外注意：区间的范围应该是long long。