Majority Element

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.

Credits:
Special thanks to @ts for adding this problem and creating all test cases.

Solution 1

```
public class Solution {
    public int majorityElement(int[] num) {

        int major=num[0], count = 1;
        for(int i=1; i<num.length;i++){
            if(count==0){
                count++;
                major=num[i];
            }else if(major==num[i]){
                count++;
            }else count--;

        }
        return major;
    }
}
```

written by jojocat1010 original link here

Solution 2

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

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

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 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

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

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(n
ums.begin() + left, nums.begin() + right + 1, rm) ? lm : rm;
    }
};
```

Moore Voting Algorithm

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

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.

```cpp
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;
    }
};
```

written by jianchao.li.fighter original link here

Solution 3

This can be solved by Moore's voting algorithm. Basic idea of the algorithm is if we cancel out each occurrence of an element e with all the other elements that are different from e then e will exist till end if it is a majority element. Below code loops through each element and maintains a count of the element that has the potential of being the majority element. If next element is same then increments the count, otherwise decrements the count. If the count reaches 0 then update the potential index to the current element and sets count to 1.

```
int majorityElement(vector<int> &num) {
    int majorityIndex = 0;
    for (int count = 1, i = 1; i < num.size(); i++) {
        num[majorityIndex] == num[i] ? count++ : count--;
        if (count == 0) {
            majorityIndex = i;
            count = 1;
        }
    }

    return num[majorityIndex];
}
```

written by satyakam original link here