原 LeetCode169:Majority Element

2015年07月09日 19:29:49 阅读数：1989

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.

Hide Tags Divide and Conquer Array Bit Manipulation

这道题在剑指offer上见到过，使用一次遍历的方式就可以求解。做完以后看了下Discuss，发现还有好多种解法。主要有下面这么多种，图片来自于这里。

1. **Runtime: O(n^2) — Brute force solution**: Check each element if it is the majority element.
2. **Runtime: O(*n*), Space: O(*n*) — Hash table**: Maintain a hash table of the counts of each element, then find the most common one.
3. **Runtime: O(*n* log *n*) — Sorting**: Find the longest contiguous identical element in the array after sorting.
4. **Average runtime: O(*n*), Worst case runtime: Infinity — Randomization**: Randomly pick an element and check if it is the majority element. If it is not, do the rando pick again until you find the majority element. As the probability to pick the majority element is greater than 1/2, the expected number of attempts is < 2.
5. **Runtime: O(*n* log *n*) — Divide and conquer**: Divide the array into two halves, then find the majority element A in the first half and the majority element B in the second half. The global majority element must either be A or B. If A == B, then it automatically becomes the global majority element. If not, then both A and B are the candidates for the majority element, and it is suffice to check the count of occurrences for at most two candidates. The runtime complexity, T(*n*) = T(*n*/2) + 2*n* = O(*n* lo *n*).
6. **Runtime: O(*n*) — Moore voting algorithm**: We maintain a current candidate and a counter initialized to 0. As we iterate the array, we look at the current element x:
    1. If the counter is 0, we set the current candidate to x and the counter to 1.
    2. If the counter is not 0, we increment or decrement the counter based on whether x is the current candidate.

   After one pass, the current candidate is the majority element. Runtime complexity = O(*n*).
7. **Runtime: O(*n*) — Bit manipulation**: We would need 32 iterations, each calculating the number of 1's for the i^{th} bit of all *n* numbers. Since a majority must exist, therefore, either count of 1's > count of 0's or vice versa (but can never be equal). The majority number's i^{th} bit must be the one bit that has the greater count.

我针对其中的5种进行了实现，第1种由于时间复杂度太高没有实现，第4种由于存在最差的情况所以也没有实现。

解法一：Hash table

这种解法是使用一个hash表，键用来存放数组的元素，键对应的值存放元素出现的次数。遍历整个数组，查找它在hash表中是否出现，如果出现将出现次数加1，如果没有出现，将它插入hash表中，并设置它的出现次数为1。每次遍历到一个元素，判断它的出现次数是否超过了数组长度的一半，要是超过了就返回该元素。时间复杂度是O(n)，空间复杂度是O(n)。。
runtime:48ms

```
1       int majorityElement(vector<int>& nums) {
2               if(nums.size()==1)
3                   return nums[0];
4               map<int,int> tables;
5               for(int i=0;i<nums.size();i++)
6               {
7                   if(tables.count(nums[i]))
8                   {
9                       tables[nums[i]]++;
10                      if(tables[nums[i]]>nums.size()/2)
11                          return nums[i];
```

```
                }
                else
                {
                    tables[nums[i]]=1;
                }
            }


    }
```

解法二：Sorting

这种解法其实应该一开始就想到的，因为这种解法可以说是最简单的。对数组进行排序，那么出现次数超过一半的元素必定是数组中的中间元素，返回这个元素即可。时间复杂度是O(nlogn)，空间复杂度是O(1)。
runtime:40ms

```
    int majorityElement(vector<int>& nums) {
        sort(nums.begin(),nums.end());
        return nums[nums.size()/2];
    }
```

解法三：Divide and conquer

这道题的提示是使用分治法或位操作来求解，最开始我思考时也是把数组分成分成两部分，但是在原始数组中出现次数超过一半的元素不一定在子数组中出现次数超过一半，到这里就不知道如果进行后续的操作了。看了上面的解释发现正确的思考方法应该是这样的：
将数组分成两部分，寻找第一个部分中出现次数超过一半的元素为A，第二个部分出现次数超过一半的元素为B，如果A==B，那么A就是这个数组中出现次数超过一半的元素，如果A!=B，那么A和B都可能是这个数组中出现次数超过一半的元素，那么重新遍历这个数组，记录A和B出现的次数，返回出现次数多的元素，时间复杂度T(n)=2T(n/2)+2n=O(nlogn)。

runtime:20ms

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

        return helper(nums,0,nums.size()-1);
    }

    int helper(vector<int> &nums,int begin,int end)
    {
        if(begin==end)
           return nums[begin];
        if(begin<end)
        {
            int mid=begin+(end-begin)/2;
            int left=helper(nums,begin,mid);
            int right=helper(nums,mid+1,end);
            if(left==right)
               return left;
            else
            {
                int leftCount=0;
                int rightCount=0;
                for(int i=begin;i<=end;i++)
                {
                    if(nums[i]==left)
                       leftCount++;
                    else if(nums[i]==right)
                       rightCount++;
                }

               if(leftCount<rightCount)
                       return right;
               else if(leftCount>rightCount)
                       return left;
            }
        }
    }
```

解法四：Moore voting alogrithm

这种解法就是在《剑指offer》上看到的那种解法。它本质上也是一种分治法，只不过在编程时使用了一些技巧，结果没那么容易看出来了。
算法的思想如下：每次从数组中找出一对不同的元素，将它们从数组中删除，直到遍历完整个数组。由于这道题已经说明一定存在一个出现次数超过一半的元素，所以遍历完数组后数组中一定会存在至少一个元素。
上面就是这种算法的思想，删除操作可以在常数时间内完成，但是查找不同的元素无法在常数时间内完成，这里有一个技巧。
在算法执行过程中，使用常量空间来记录一个候选元素c以及它的出现次数f(c)，c即为当前阶段出现次数超过一半的元素。在遍历开始之前，该元素c为空，f(c)=0。然后开始遍历数组A时：

1. 如果f(c)为0，表示当前并没有候选元素，也就是说之前的遍历过程中没有找到超过一半的元素。那么，如果超过一半的元素c存在，那么c在剩下的子数组中，出现的次数也一定超过一半。因次我们可以将原始问题转化成它的子问题。此时c赋值为当前元素，同时f(c)=1。
2. 如果当前A[i]==c，那么f(c)+=1。（没有找到相同的元素，只需要把相同的元素累加起来）
3. 如果当前元素A[i]!=c，那么f(c)-=1（相当于删除一个c），不对A[i]做任何处理（相当于删除A[i]）
   如果遍历结束之后，f(c)不为0，那么再次遍历一遍数组，如果c真正出现的频率，上面算法的时间复杂度是O(n)，空间复杂度为O(1)。这种方法的分析来自这里。

runtime:20ms

```
    int majorityElement(vector<int>& nums) {
        int count=0;
        int result=nums[0];
        for(int i=0;i<nums.size();i++)
        {
            if(count==0||nums[i]==result)
            {
                count++;
                result=nums[i];
            }
            else
                count--;
        }
        return result;
    }
```

解法五：Bit manipulation

还可以使用位操作来求解这道题。因为一个整数在32为机器上只有32位，那么可以使用一个长度为32的数组来记录输入数组中每1位中1的个数和0的个数，由于存在一个出现次数超过一半的元素，那么取第i位中出现次数多的（0或者1）即可以构成超过数组一半元素。
runtime:40ms

```
   int majorityElement(vector<int>& nums) {
        int **table=new int*[32];
        int result=0;
        for(int i=0;i<32;i++)
        {
            table[i]=new int[2]();
        }
        for(int i=0;i<nums.size();i++)
        {
            for(int j=0;j<32;j++)
            {
                int pos=1<<j&nums[i]?1:0;
                table[j][pos]++;
            }
        }
        for(int i=0;i<32;i++)
        {
            if(table[i][1]>table[i][0])
                result+=1<<i;
        }
        return result;
    }
```

个人分类： LeetCode

所属专栏： LeetCode