# LeetCode面试题整理

## 基础

### 只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

示例 1:

输入: [2,2,1]

输出: 1

示例 2:

输入: [4,1,2,1,2]

输出: 4

实现代码：

1. 异或操作。（相同的数字经过异或运算后结果为0；任何数字与0进行异或运算都是该数字本身）；O(n)，O(1)

class Solution {

public int singleNumber(int[] nums) {

int num = 0;

for(int i=0; i<nums.length; i++){

num = num ^ nums[i];

}

return num;

}

}

1. 利用HashSet去重 ；HashSet的add方法是通过HashMap来实现的，HashSet中的元素都是HashMap中的key，HashMap的put方法调用的 resize() 、 putTreeVal() 等方法本身也是O(n2)的时间复杂度。因此总的时间复杂度为O(n2)，空间复杂度为O(n)

import java.util.HashSet;

import java.util.Set;

class Solution {

public int singleNumber(int[] nums) {

Set<Integer> numSet = new HashSet<>();

int len = nums.length;

for(int i=0; i<len; i++){

int num = nums[i];

if(!numSet.add(num)){

numSet.remove(num);

}

}

return numSet.iterator().next();

}

}

1. 先排序后比较；O(n2)，O(1)

class Solution {

public int singleNumber(int[] nums) {

int len = nums.length;

for(int i=0; i<len; i++){

int idx = i;

for(int j=i; j<len; j++){

if(nums[j]<nums[idx]){

idx = j;

}

}

int tmp = nums[i];

nums[i] = nums[idx];

nums[idx] = tmp;

}

for(int i=0; i<len-1; i+=2){

if(nums[i] != nums[i+1])

return nums[i];

}

return nums[len-1];

}

}

### 求众数

给定一个大小为 n 的数组，找到其中的众数。众数是指在数组中出现次数大于 ⌊ n/2 ⌋ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在众数。

示例 1:

输入: [3,2,3]

输出: 3

示例 2:

输入: [2,2,1,1,1,2,2]

输出: 2

实现代码：

1. 先排序后直接返回位于n/2（向下取整）下标的元素。O(n2) （使用选择排序，快速排序为O(n log n)）。

import java.util.Set;

import java.util.HashSet;

class Solution {

public int majorityElement(int[] nums) {

int len = nums.length;

for(int i=0; i<len; i++){

int idx = i;

for(int j=i; j<len; j++){

if(nums[j] < nums[idx])

idx = j;

}

int tmp = nums[idx];

nums[idx] = nums[i];

nums[i] = tmp;

}

return nums[len/2];

}

}

1. 使用HashMap或HashTable，遍历数组，查找是否在HashMap或HashTable中出现，出现则加1，出现次数大于n/2时，返回。 O(n)；O(n)。
2. 分治法。O(n)
3. 成对匹配。摩尔投票算法。成对删除。O(n)；O(1)。

class Solution {

public int majorityElement(int[] nums) {

int res = nums[0];

int count = 1;

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

if (res == nums[i]){

count++;

}

else {

count--;

if (count == 0){

res = nums[++i];

count++;

}

}

}

return res;

}

}

##### Boyer-Moore majority vote algorithm(摩尔投票算法)

###### 简介

Boyer-Moore majority vote algorithm(摩尔投票算法)是一种在线性时间O(n)和空间复杂度的情况下，在一个元素序列中查找包含最多的元素。它是以Robert S.Boyer和J Strother Moore命名的，1981年发明的，是一种典型的流算法(streaming algorithm)。

在它最简单的形式就是，查找最多的元素，也就是在输入中重复出现超过一半以上(n/2)的元素。如果序列中没有最多的元素，算法不能检测到正确结果，将输出其中的一个元素之一。

当元素重复的次数比较小的时候，对于流算法不能在小于线性空间的情况下查找频率最高的元素。

###### 算法描述

算法在局部变量中定义一个序列元素(m)和一个计数器(i)，初始化的情况下计数器为0. 算法依次扫描序列中的元素，当处理元素x的时候，如果计数器为0，那么将x赋值给m，然后将计数器(i)设置为1，如果计数器不为0，那么将序列元素m和x比较，如果相等，那么计数器加1，如果不等，那么计数器减1。处理之后，最后存储的序列元素(m)，就是这个序列中最多的元素。

如果不确定是否存储的元素m是最多的元素，还可以进行第二遍扫描判断是否为最多的元素。

###### perudocode

* Initialize an element m and a counter i with i = 0
* For each element x of the input sequence:
  + if i = 0, then assign m = x and i = 1
  + else if m = x, then assign i = i + 1
  + else assign i = i − 1
* Return m

###### 算法举例

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.

###### 实现代码

class Solution {

public:

// moore majority vote algorithm

int majorityElement(vector<int>& nums) {

int m;

int count = 0;

for (int i = 0; i < nums.size(); i++) {

if (count == 0) {

m = nums[i];

count++;

} else if (nums[i] == m) {

count++;

} else

count--;

}

return m;

}

};

还有一个类似的算法题，就是判断一个序列中，某个元素的个数是否超过n/2，其中一种解法就是利用分治算法。还可以用上面找到的摩尔投票算法，第一遍扫描输出一个存储的元素，然后还需要进行第二遍扫描来判断元素在序列中是否确实超过n/2了。 因为一个元素超过一半，最后肯定会留下，但是最后留下的不一定超过一半，所以要扫描第二遍。

###### 代码实现

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

class Solution {

public:

// divide and conquer

int majorityElement(vector<int>& nums, int majority) {

if (nums.size() <= 2) {

int num = 0;

for (int i = 0; i < nums.size(); i++) {

if (nums[i] == majority)

num++;

}

return num;

}

int middle = floor(nums.size() / 2);

vector<int> left(nums.begin(), nums.begin() + middle);

vector<int> right(nums.begin() + middle, nums.end());

int left\_num = majorityElement(left, majority);

int right\_num = majorityElement(right, majority);

return left\_num + right\_num;

}

// moore majority vote algorithm

// 判断majority 是否大于一半以上,不含等于。

int moore\_majority\_vote\_algorithm(vector<int>& nums, int majority) {

int m;

int counter = 0;

// 第一轮扫描

for (int i = 0; i < nums.size(); i++) {

if (counter == 0) {

counter = 1;

m = nums[i];

} else if (m != nums[i]) {

counter--;

} else

counter++;

}

if (m != majority)

return -1;

int new\_counter = 0;

// 第二轮扫描

for (int i = 0; i < nums.size(); i++) {

if (nums[i] == m)

new\_counter++;

}

return new\_counter;

}

};

int main() {

int num[] = {2, 2, 1, 1, 1, 2};

vector<int> vec(num, num + 6);

Solution\* solution = new Solution();

int counter;

cout << (counter = solution->majorityElement(vec, 2)) << endl;

//cout << (counter = solution->moore\_majority\_vote\_algorithm(vec, 2)) << endl;

if (counter > floor(vec.size() / 2)) {

cout << "Yes" << endl;

} else

cout << "No" << endl;

return 0;

}