# Leetcode笔记

## Array

### 1.1.1 027 Remove Element

Given an array and a value, remove all instances of that value in place and return the new length. The order of elements can be changed. It doesn't matter what you leave beyond the new length.

两种方法：1. 将需要删除的元素放在数组的末尾。考虑时想到如果末尾的值也等于value，那么交换到前面去不是错了吗。 其实，int i=0, 当前面的一个值和末尾值交换后，i并不加一，所以还会在原位置再判断刚刚换回来的值是不是满足条件。 这里用到int j = nums.length -1, 利用它来标记新数组的长度。 2. 想象成把原数组A的元素依次诺入到新数组B中，必然有B的长度小于A的长度，因为可以利用在A当做B来用。 从前向后，把数组中可以保留的保留下来，不可以保留的其位置会被后面满足条件的数占据，这样最后剩下的前面部分是满足条件的数组。Newindex 是满足条件的数组长度。

<http://blog.csdn.net/feliciafay/article/details/45006349>

### 1.1.2 283 Move Zeroes

设置两个指针. I用于遍历整个数组，j用于指向非零的数，最后j后面的数都置零。

### 217 containsDuplicate

使用HashSet类，它的add和contains方法。

原写法，会超时public boolean containsDuplicate(int[] nums) {

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

{

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

if (nums[i] == nums[j])

return true;

}

return false;

}

### 1.1.4 88 Merge Sorted Array

public void merge(int[] nums1, int m, int[] nums2, int n) {

int i=m-1,j=n-1,k=m+n-1;

while (i>-1&&j>-1) nums1[k--]=(nums1[i]>nums2[j]?nums1[i--]:nums2[j--]);

while (j>-1) nums1[k--]=nums2[j--];

}

### 1.1.5 169. Majority Element

　　给定一个数组，找这个数组中的主元素，主元素是元素出现次数大于⌊ n/2 ⌋的元素。

　　假设给定的数组非空，主元素都存在。

解题思路

　　用一个标记cnt记录某个元素出现的次数，如果后面的元素和它相同就加一，有一个元素和他不相同就减一，当cnt小于等于0时重新记录新的元素。

代码实现

算法实现类

public class Solution {

public int majorityElement(int[] num) {

int main = num[0]; // 用于记录主元素，假设第一个是主元素

int count = 1; // 用于抵消数的个数

for (int i = 1; i < num.length; i++) { // 从第二个元素开始到最后一个元素

if (main == num[i]) { // 如果两个数相同就不能抵消

count++; // 用于抵消的数据加1

} else {

if (count > 0) { // 如果不相同，并且有可以抵消的数

count--; // 进行数据抵消

} else { // 如果不相同，并且没有可以抵消的数

main = num[i]; // 记录最后不可以抵消的数

}

}

}

// 对于数组中可能没有主元素的情况，题中说明存在，此步可以省略。

// count = 0;

// for (int a : num) {

// if (a == main) {

// count++;

// }

// }

//

// if (count >= num.length / 2) {

// return main;

// } else {

// throw new RuntimeException("No majority element");

// }

return main;

}

}

### 1.1.6 229. Majority Element II

方法一:

观察可知，数组中至多可能会有2个出现次数超过 ⌊ n/3 ⌋ 的众数

记变量n1, n2为候选众数； c1, c2为它们对应的出现次数

遍历数组，记当前数字为num

若num与n1或n2相同，则将其对应的出现次数加1

否则，若c1或c2为0，则将其置为1，对应的候选众数置为num

否则，将c1与c2分别减1

最后，再统计一次候选众数在数组中出现的次数，若满足要求，则返回之。

方法二：

这类问题有一个很经典的算法叫做BM多数投票算法（Boyer-Moore Majority Vote algorithm），时间复杂度O(n)，空间复杂度O(1)。大体思路就是使用一个变量来计数当前出现次数最多的元素X。如果你发现了一个不等于X的数Y，那么计数变量减一。例如，如果数组中有5个X和4个Y，那么可以说X比Y多（5-4）个，或者说有4个X被Y抵消了。想进一步了解该算法的话，请参考http://goo.gl/64Nams。

对于这道题，我们要找到是出现次数大于总数1/3的数，符合要求的数可以能有0个、1个、2个。我们可以在BM多数投票算法的基础上稍作改动，使用2个计数变量即可。其他原理是一样的。需要注意的是，在一轮“抵消”的过程结束后，我们只能保证此时得到的这2个元素是数量最多的，但不见得就超过了总数的1/3。因此，我们还要再分别对这两个数进行一次计数，看它们出现的次数是不是超过了1/3。

// Modified Boyer-Moore Algorithm

// O(n) time, O(1) space

public class Solution {

public List<Integer> majorityElement(int[] nums) {

List<Integer> res = new ArrayList<Integer>();

int count1 = 0;

int count2 = 0;

int majority1 = 0;

int majority2 = 1;

for(int n : nums){

if(count1 == 0) majority1 = n;

else if(count2 == 0) majority2 = n;

if(n == majority1) count1++;

else if(n == majority2) count2++;

else{

count1--;

count2--;

}

}

count1 = count2 = 0;

for(int n : nums){

if(n == majority1) count1++;

else if (n == majority2) count2++; //一开始写成了if， 这样若输入[1,1]，本应输出[1],结果输出了[1,1];

}

if(count1 > nums.length / 3) res.add(majority1);

if(count2 > nums.length / 3) res.add(majority2);

return res;

}

}

## Linked list

### 1.2.1 237 删除单链表的一个节点

没有给我们链表的起点，只给我们了一个要删的节点，

我们之前要删除一个节点的方法是要有其前一个节点的位置，然后将其前一个节点的next连向要删节点的下一个，然后delete掉要删的节点即可。

这道题的处理方法是先把当前节点的值用下一个节点的值覆盖了，然后我们删除下 一个节点即可。

### 1.2.2 203 删除单链表里所有值为val的节点。

写的时候用到 n.next = n.next.next. 所以涉及到链表长度为0,1,2的问题，可能出现nullpointer错误。 再加上判断n.next.val是不是 val ，这个链表长度在变化中，for循环和 if 判断不知道如何合在一起处理。

其实用递归，简单几行就可以了。

### 1.2.3 Reverse Linked List

自己写的 public ListNode reverseList(ListNode head) {

if(head == null || head.next == null)

return head;

while(head.next != null){

ListNode newhead = head.next;

newhead.next = head;

head = newhead;}

return head;

} //网站提示会超时，结果正确。其实举例运行一下也不对。两个单链不够用。

public ListNode reverseList(ListNode head) {

ListNode newHead = null;

while(head != null){

ListNode next = head.next;

head.next = newHead;

newHead = head;

head = next;

}

return newHead;

}

*// recursive solution 递归算法*

public ListNode reverseList(ListNode head) {

return reverseListInt(head, null);

}

public ListNode reverseListInt(ListNode head, ListNode newHead) {

if(head == null)

return newHead;

ListNode next = head.next;

head.next = newHead;

return reverseListInt(next, head);

}

### 1.2.4 Palindrome Linked List

思路：

1.如何判断一个链表是否是回文的？很简单将链表中的元素遍历出来并放进ArrayList中，然后可以像数组一样来判断该元素是否为回文的，时间复杂度O(n)，空间复杂度O(n)，可如何用O(n)的时间复杂度和O(1)的空间复杂度来解决呢？

2.是不是可以考虑将链表反转？可反转后还是链表啊，要是将链表分为前后两个部分呢，分为两个部分还是无法判断该链表是否为回文链表啊，那要是再将其中一个链表反转一下呢，It's done!好多时候，多想一步容易，再多想一步就困难了。

代码：

public boolean isPalindrome(ListNode head) {

if(head==null||head.next==null) return true;

ListNode p=head,temp=head,quick=head;

while(temp!=null&&quick!=null)

{ temp=temp.next;

if(quick.next==null) break;

quick=quick.next.next; } // 若p！=null，就可以写出p.next，顶多p.next = null;

temp=reverseList(temp);

while(temp!=null&&p!=null)

{ if(temp.val!=p.val)

return false;

temp=temp.next;

p=p.next; }

return true; //就算是奇数个字符，最后出现temp=null,跳出循环，也是返回true.

}

public ListNode reverseList(ListNode head) { //翻转链表

ListNode newHead =null;

if(head == null) return newHead;

while (head!=null){

ListNode next = head.next;

head.next = newHead;

newHead = head;

head = next;

}

return newHead;}

### 1.2.5 Valid Palindrome 回文验证

Java-native

public class Solution {

public boolean isPalindrome(String s) {

if(s == null) return false;

if(s.length() < 2) return true;

char[] charArray = s.toCharArray();

int len = s.length();

int i=0;

int j=len-1;

while(i<j){

char left, right;

while(i<len-1 && !isAlpha(left) && !isNum(left)){

i++;

left = charArray[i];

}

while(j>0 && !isAlpha(right) && !isNum(right)){

j--;

right = charArray[j];

}

if(i >= j)

break;

left = charArray[i];

right = charArray[j];

if(!isSame(left, right)){

return false;

}

i++;

j--;

}

return true;

}

public boolean isAlpha(char a){

if((a >= 'a' && a <= 'z') || (a >= 'A' && a <= 'Z')){

return true;

}else{

return false;

}

}

public boolean isNum(char a){

if(a >= '0' && a <= '9'){

return true;

}else{

return false;

}

}

public boolean isSame(char a, char b){

if(isNum(a) && isNum(b)){

return a == b;

}else if(Character.toLowerCase(a) == Character.toLowerCase(b)){

return true;

}else{

return false;

}

}

}

**\*Java Solution 2 - Using Stack 还未看**

**public boolean isPalindrome(String s) {**

**s = s.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();**

**int len = s.length();**

**if (len < 2)**

**return true;**

**Stack<Character> stack = new Stack<Character>();**

**int index = 0;**

**while (index < len / 2) {**

**stack.push(s.charAt(index));**

**index++;**

**}**

**if (len % 2 == 1)**

**index++;**

**while (index < len) {**

**if (stack.empty())**

**return false;**

**char temp = stack.pop();**

**if (s.charAt(index) != temp)**

**return false;**

**else**

**index++;**

**}**

**return true;**

**}**

**Java Solution 3 - Using Two Pointers**

if(s==null||s.length()==0) return false;

s = s.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();

System.out.println(s);

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

if(s.charAt(i) != s.charAt(s.length() - 1 - i))

return false;

}

return true;

### 1.2.6 Reverse linked list II

原代码：

if (head == null) return head;

ListNode partHead = head;

for(int i=1; i<m; i++)

partHead = partHead.next;

ListNode newPHead = partHead;

for(int j=m; j<=n; j++){

ListNode next = partHead.next;

partHead.next = newPHead;

newPHead = partHead;

partHead = next;

}

//面临一个问题就是：newPHead 怎么嵌入到原head中。即需要相应位置前面的节点指向newPHead, newPHead 的最后一个节点指向剩下的第一个节点。

其实就是需要新建一个指针，它应该是在newPHead的前面，由它指向newPHead.

答案: ?

public ListNode reverseBetween(ListNode head, int m, int n) {

if (head == null) return head;

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode pre = dummy;

for(int i = 0; i<m-1; i++)

pre = pre.next;

ListNode start = pre.next;

ListNode then = start.next;

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

start.next = then.next;

then.next = pre.next;

pre.next = then;

then = start.next;

}

return dummy.next;

}

### 1.2.7 83. Remove Duplicates from Sorted List

错误答案：

if(head == null || head.next == null) return head;

ListNode newHead = head; //证明这种方式来修改单链，再返回是可行的

while( newHead.next != null){

while(newHead.val == newHead.next.val)

newHead.next = newHead.next.next;

newHead = newHead.next;

} //输入1,1,1,2,3,3,5

return head; //输入 [1,1,1,2,3,3,5]时正确，但是输入【1,1】时会有nullpointer错误。 因为运行完newHead = newHead.next后，newhead 为空，不存在newHead.next。

//return newHead; // 结果是5

正确答案：

if(head == null || head.next == null) return head;

ListNode newHead = head;

while( new Head.next != null){

if(newHead.val == newHead.next.val)

newHead.next = newHead.next.next;

else

newHead = newHead.next;

} 换成if else,当发现重复时，将newHead指向下一个的下一个，之前为了避免下一个的下一个依然等于newHEAD，所以设置了while循环。事实上，此时可以不移动newhead,只有当前后两个不同时，再将newhead移动。

return head;

递归算法：

If(head == null || head.next == null) return head;

Head.next = deleteDuplicates(head.next);

Return heas.val == head.next.val ? head.next : head;

### 160. Intersection of Two Linked Lists

方法一： 先把两条单链截为同一长度，因为如果后面部分长度不同，是不可能交叉的。

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

int la = length(headA) ;

int lb = length(headB);

while (la > lb){

headA = headA.next;

la --;

}

while (lb > la){

headB = headB.next;

lb --;

}

while (headA != headB){ //这里比较的是节点，而不是节点的值 headA.val

headA = headA.next;

headB = headB.next;

}

return headA;

}

private int length(ListNode head){

int length = 0;

while(head != null){

head = head.next;

length ++;

}

return length;

}

方法二： Time O(n+m), space O(1)

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if( null==headA || null==headB )

return null;

ListNode curA = headA, curB = headB;

while( curA!=curB){

curA = curA==null?headB:curA.next;

curB = curB==null?headA:curB.next;

}

return curA;

}

### 21. Merge Two Sorted Lists

if (l1== null) return l2;

if (l2== null) return l1;

if(l1.val<l2.val){

l1.next = mergeTwoLists(l1.next,l2);

return l1;

}

else{

l2.next = mergeTwoLists(l1,l2.next);

return l2;

}

}

### 1.2.10 328. Odd Even Linked List

思路：

1 小于3个元素的，直接返回原有head

2 大于等于3个元素的 while循环{

2.1 建立odd 链表 2.2 建立even链表}

odd链表尾部指针指向even链表的头部

3 返回head

public ListNode oddEvenList(ListNode head) {

if(head == null || head.next == null)

return head;

ListNode odd=head, even=head.next, evenHead=head.next;

while(even!=null&& even.next!=null){

odd.next = even.next;

odd = odd.next;

even.next = odd.next;

even = even.next;

}

odd.next = evenHead;

return head;

}

## tree

### 1.3.1 104 求二叉树的最大深度

递归思路： f(r)=0 , r=null

F(r)= 1+max(f(r.left),f(r.right))

运行时从左开始遍历，属于后序遍历（一个节点处的工作实在它的诸儿子节点被计算后进行的）。

根的深度为0。

Root ==null 树为空

Root.left == null && root.right == null 树没有左子树和右子树。 此时返回1，指的树只有一个根节点。

### 1.3.2 111 求二叉树的最小深度

if (root == null)

return 0;

if (root.right == null && root.left == null)

return 1;

if (root.right == null)

return 1+ minDepth(root.left);

if (root.left == null)

return 1+ minDepth(root.right);

return 1+ ( Math.min(minDepth(root.left),minDepth(root.right)));

// minDepth(root.left) <minDepth(root.right) ? minDepth(root.left) :minDepth(root.right)); 用这种方法会超时

错误代码1：int dep =0;

if (root == null)

return 0;

if (root.right == null || root.left == null)

return dep ++ ;

return (minDepth(root.right)< minDepth(root.left) ? minDepth(root.right) : minDepth(root.left));

错误代码2：if (root == null)

return 0;

if (root.right == null && root.left == null)

return 1;

// if (root.right == null || root.left == null)

// return 2; 如果不加这行代码，对于树【1,2】，计算结果为1，实际上应该为2。 如果加这行代码，对于树【1,2，null，3，null，4，null，5，null】，计算结果为2， 实际应该为5。其实某一子树为空时，应该返回另一子树的值。

### 1.3.3 100 判断两个二叉树是否相等。

凡是二叉树的，都要用到递归。 写递归，要先写递归最后终止的条件，一般是null 的情况。 然后再写一般情形。 本题代码最后的return true 容易忘记，即两棵树完全一样，就会运行完所有的代码，再执行 return true 语句。

public class Solution {

public boolean isSameTree(TreeNode p, TreeNode q) {

if (p == null && q == p)

return true;

if (p == null && q !=p)

return false;

if (q == null && p !=q)

return false;

if(p.val != q.val)

return false;

else {

if( !isSameTree(p.right,q.right))

return false;

if( !isSameTree(p.left,q.left))

return false;

}

return true;

}

}

上面的代码相对下面的来说，其实运行速度是一样的。

简化的代码：

public boolean isSameTree(TreeNode p, TreeNode q) {

if(p==null && q==null) return true;

if(p==null || q==null) return false;

return (p.val==q.val) && isSameTree(p.left,q.left) && isSameTree(p.right,q.right);

}

### 1.3.4 226 invert binary Tree (自己写的哦)

public class Solution {

public TreeNode invertTree(TreeNode root) {

if (root == null)

return root;

TreeNode tmp = root.left;

root.left = root.right;

root.right = tmp; //只要二叉树root ！= null，那么就可以使用

// root.right和root.left的写法，因为至多他们是NULL.

invertTree(root.left);

invertTree(root.right);

return root; //不要忘记返回这个

}

}

简化代码：public class Solution {

public TreeNode invertTree(TreeNode root) {

if (root == null)

return root;

TreeNode tmp = root.left;

root.left = invertTree(root.right);

root.right = invertTree(tmp);

return root;

}

}

### 1.3.5 102 Binary Tree Level Order Traversal

### 1.3.6 235 Lowest Common Ancestor of a Binary Search Tree

利用起二叉查找树的性质.

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if(root.val>p.val && root.val>q.val)

return lowestCommonAncestor(root.left,p,q);

if(root.val<p.val && root.val<q.val)

return lowestCommonAncestor(root.right,p,q);

else

return root; // return root 这个特殊情况放在最后

}

### 1.3.7 236 Lowest Common Ancestor of a Binary Tree

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if(root == null || root == p || root == q) return root;

TreeNode left = lowestCommonAncestor(root.left,p,q);

TreeNode right = lowestCommonAncestor(root.right,p,q);

if(left != null && right!=null) return root;

return (left !=null? left:right);

}

### 1.3.8 110. Balanced Binary Tree

public boolean isBalanced(TreeNode root) {

if(root == null){

return true;

}

// 如果子树高度差大于1，则不平衡

if(Math.abs(depth(root.left)-depth(root.right)) > 1){

return false;

}

// 递归检查左子树和右子树的平衡性

return isBalanced(root.left) && isBalanced(root.right);

}

// 帮助方法，返回树的高度

private int depth(TreeNode root){

if(root == null){

return 0;

}

return 1 + Math.max(depth(root.left), depth(root.right));

}

### 1.3.9 101. Symmetric Tree depth-first search

public boolean isSymmetric(TreeNode root) {

return root == null || isMirror(root.left, root.right);

}

public boolean isMirror(TreeNode p, TreeNode q) {

return p == null && q == null ||

p != null && q != null &&

p.val == q.val &&

isMirror(p.left, q.right) &&

isMirror(p.right, q.left);

}

## 1.4 String

### 1.4.1 171 Excel Sheet Column Number

String 类的length()和charAt(i)的方法，而不是s.length和s[i] ,数组才可以用。

int sum = 0;

int tmp = 0;

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

tmp = s.charAt(i) -'A' + 1;

sum = 26\*sum + tmp; 26进制的换算。

}

原想的是将string转成数组。里面的代码没想出来。

char[] titles = s.toCharArray();

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

}

### 1.4.2 Valid anagram

if((s.length()!=t.length())) return false;

int[] alphabet = new int[26]; //新建int[] 数组

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

alphabet[s.charAt(i)-'a']++;

alphabet[t.charAt(i)-'a']--;

}

For(int I : alphabet)

if (i!=0) return false; //数组可以使用foreach增强循环

return true;

### 1.4.3 13 Roman to Integer

public int romanToInt(String s) {

//：Ⅰ（1）Ⅴ（5）Ⅹ（10）L（50）C（100）D（500）M（1000）

// rules:位于大数的后面时就作为加数；位于大数的前面就作为减数

//eg：Ⅲ=3,Ⅳ=4,Ⅵ=6,ⅩⅨ=19,ⅩⅩ=20,ⅩLⅤ=45,MCMⅩⅩC=1980

If(s == null || s.length() == 0) return 0;

Int len = s.length();

HashMap<Character,Integer> map = new HashMap<Character,Interger>();

Map.put(‘I’,1);

map.put('V',5);

map.put('X',10);

map.put('L',50);

map.put('C',100);

map.put('D',500);

map.put('M',1000);

int result = map.get(s.charAt(len -1)); // 在s=ⅩLⅤ中，s.charAt(2)=V,s.charAt(0)=X

int pivot = result;

for(int I = len -2; I >= 0; i—){

int curr = map.get(s.charAt(i)); // curr =10( L), pivot =5( V).

if(curr >= pivot){

result +=curr; //result = 15

}else{

result -=curr;

}

pivot = curr;

}

return result;

}}

### 1.4.4 Integer to Roman

int[] values = {1000,900,500,400,100,90,50,40,10,9,5,4,1};

String[] strs = {"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};

StringBuilder sb = new StringBuilder();

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

while(num >= values[i]) {

num -= values[i];

sb.append(strs[i]);

}

}

return sb.toString();

}

## Map & Table

## 1.6 Queue & Stacks

### 1.6.1 232. Implement Queue using Stacks

正确答案：

class MyQueue {

Stack<Integer> input = new Stack();

Stack<Integer> output = new Stack();

public void push(int x) {

input.push(x);

}

public void pop() {

peek();//先把放进input栈的所有元素转移到output栈里，才能弹出正确元素

output.pop();

}

public int peek() { //完成input栈向output栈传送值

if (output.empty()) //最初output栈是没有元素的

while (!input.empty()) //假如input栈里面已经push进去了元素

output.push(input.pop()); //只要input栈里面还有元素，就把它弹出然后放到output栈里面

return output.peek();

}

public boolean empty() {

return input.empty() && output.empty(); //栈为空时，empty()=true

}

}

错误答案：

Stack<Integer> input = new Stack();

Stack<Integer> output = new Stack();

public void push(int x) {

input.push(x);}

// Removes the element from in front of queue.

public void pop() {

output.pop();

}

// Get the front element.

public int peek() {

return output.peek();

}

// Return whether the queue is empty.

public boolean empty() {

if((!input.isEmpty()) && (!output.isEmpty())

return false;

return true;

}

### 1.6.2 225. Implement Stack using Queues

用一个队列即可.

Queue<Integer> q = new LinkedList<Integer>();

public void push(int x) {

q.add(x);

}

// Removes the element on top of the stack.

public void pop() {

for(int i = 1;i<q.size();i++)

q.add(q.remove());

q.remove();

}

// Get the top element.

public int top() {

for(int i = 1;i<q.size();i++)

q.add(q.remove());

int ret = q.remove();

q.add(ret);

return ret;

}

// Return whether the stack is empty.

public boolean empty() {

return q.isEmpty();

}

另解: Queue<Integer> queue;

public MyStack()

{

this.queue=new LinkedList<Integer>();

}

// Push element x onto stack.

public void push(int x)

{

queue.add(x);

for(int i=0;i<queue.size()-1;i++)

{

queue.add(queue.poll());

}

}

// Removes the element on top of the stack.

public void pop()

{

queue.poll();

}

// Get the top element.

public int top()

{

return queue.peek();

}

// Return whether the stack is empty.

public boolean empty()

{

return queue.isEmpty();

}

## Bit

### 1.7.1 231. Power of Two

public boolean isPowerOfTwo(int n) {

return n>0 && ((n&(n-1))==0);

}

### 1.7.2 191. Number of 1 Bits

public int hammingWeight(int n) {

int ones = 0;

while(n!=0){

ones = ones + (n&1);

n = n>>>1;

}

return ones;

}

### 190. Reverse Bits

public int reverseBits(int n) {

int result = 0;

for(int i =0; i<32;i++){

result += n&1;

n >>>= 1;

if(i<31)

result <<= 1;

}

return result;

}