剑指offer解析（Java实现）

2019-01-18 18:32:16

以下题目按照牛客网在线编程排序，所有代码示例代码均已通过牛客网OJ。

### 二维数组的查找

#### 题目描述

在一个二维数组中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

public boolean Find(int target, int [][] arr) {  
  
}

#### 解析

暴力方法是遍历一遍二维数组，找到target就返回true，时间复杂度为O(M \* N)（对于M行N列的二维数组）。

由题可知输入的数据样本具有高度规律性（单独一行的数据来看是有序的，单独一列的数据来看也是有序的），因此考虑能否有一个比较基准在一次的比较中根据有序性淘汰不必再进行遍历比较的数。**有序**、**查找**，由此不难联想到二分查找，我们可以借鉴二分查找的思路，每次选出一个数作为比较基准进而淘汰掉一些不必比较的数。二分是选取数组的中位数作为比较基准的，因此能够保证每次都淘汰掉二分之一的数，那么此题中有没有这种特性的数呢？我们不妨举例观察一下：

不难发现上图中对角线上的数是其所在行和所在列形成的序列的中位数，不妨选取右上角的数作为比较基准，如果不相等，那么我们可以淘汰掉所有它左边的数或者它所有下面的，比如对于target = 6，因为(0,3)位置上的4 < 6，因此(0,3)位置及其同一行的左边的所有数都小于6因此可以直接淘汰掉，淘汰掉之后问题就变为了从剩下的三行中找target，这与原始问题是相似的，也就是说每一次都选取右上角的数据为比较基准然后淘汰掉一行或一列，直到某一轮被选取的数就是target或者已经淘汰得只剩下一个数的时候就一定能得出结果了，因此时间复杂度为被淘汰掉的行数和列数之和，即O(M + N)，经过分析后不难写出如下代码：

public boolean Find(int target, int [][] arr) {  
 //input check  
 if(arr == null || arr.length == 0 || arr[0] == null || arr[0].length == 0){  
 return false;  
 }  
 int i = 0, j = arr[0].length - 1;  
 while(i != arr.length - 1 && j != 0){  
 if(target > arr[i][j]){  
 i++;  
 }else if(target < arr[i][j]){  
 j--;  
 }else{  
 return true;  
 }  
 }  
  
 return target == arr[i][j];  
}

值得注意的是每次选取的数都是第一行最后一个数，因此前提是第一行有数，那么就对应着输入检查的arr[0] == null || arr[0].length == 0，这点比较容易忽略。

总结：经过分析其实不难发现，此题是在一维有序数组使用二分查找元素的一个变种，我们应该充分利用数据本身的规律性来寻找解题思路。

### 替换空格

#### 题目描述

请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

public String replaceSpace(StringBuffer str) {  
   
}

此题考查的是字符串这个数据结构的数组实现（对应的还有链表实现）的相关操作。

#### 解析

##### String.replace简单粗暴

如果可以使用API，那么可以很容易地写出如下代码：

public String replaceSpace(StringBuffer str) {  
 //input check  
 //null pointer  
 if(str == null){  
 return null;  
 }  
 //empty str or not exist blank  
 if(str.length() == 0 || str.indexOf(" ") == -1){  
 return str.toString();  
 }  
  
 for(int i = 0 ; i < str.length() ; i++){  
 if(str.charAt(i) == ' '){  
 str.replace(i, i + 1, "%20");  
 }  
 }  
  
 return str.toString();  
}

##### 时间O(n)，空间O(n)

但是如果面试官告诉我们不许使用封装好的替换函数，那么目的就是在考查我们对字符串**数组实现**方式的相关操作。由于是连续空间存储，因此需要在创建实例时指定大小，由于每个空格都使用%20替换，因此替换之后的字符串应该比原串多出空格数 \* 2个长度，实现如下：

public String replaceSpace(StringBuffer str) {  
 //input check  
 //null pointer  
 if(str == null){  
 return null;  
 }  
 //empty str or not exist blank  
 if(str.length() == 0 || str.indexOf(" ") == -1){  
 return str.toString();  
 }  
  
 char[] source = str.toString().toCharArray();  
 int blankCount = 0;  
 for(int i = 0 ; i < source.length ; i++){  
 blankCount = (source[i] == ' ') ? blankCount + 1 : blankCount;  
 }  
 char[] dest = new char[source.length + blankCount \* 2];  
 for(int i = source.length - 1, j = dest.length - 1 ; i >=0 && j >=0 ; i--, j--){  
 if(source[i] == ' '){  
 dest[j--] = '0';  
 dest[j--] = '2';  
 dest[j] = '%';  
 continue;  
 }else{  
 dest[j] = source[i];  
 }  
 }  
  
 return new String(dest);  
}

##### 时间O(n)，空间O(1)

如果还要求不能有额外空间，那我们就要考虑如何复用输入的字符串，如果我们从前往后遇到空格就将空格及其之后的两个位置替换为%20，势必会覆盖空格之后的两个字符，比如hello world会被替换成hello%20rld，因此我们需要在长度被扩展后的新串中从后往前确定每个索引上的字符。比如使用一个originalIndex指向原串中的最后一个字符索引，使用newIndex指向新串的最后一个索引，每次将originalIndex上的字符复制到newIndex上并且两个指针前移，如果originalIndex上的字符是空格，则将newIndex依次填充0,2,%，然后两者再前移，直到两者都到首索引位置。

public String replaceSpace(StringBuffer str) {  
 //input check  
 //null pointer  
 if(str == null){  
 return null;  
 }  
 //empty str or not exist blank  
 if(str.length() == 0 || str.indexOf(" ") == -1){  
 return str.toString();  
 }  
  
 int blankCount = 0;  
 for(int i = 0 ; i < str.length() ; i++){  
 blankCount = (str.charAt(i) == ' ') ? blankCount + 1 : blankCount;  
 }  
 int originalIndex = str.length() - 1, newIndex = str.length() - 1 + blankCount \* 2;  
 str.setLength(newIndex + 1); //需要重新设置一下字符串的长度，否则会报越界错误  
 while(originalIndex >= 0 && newIndex >= 0){  
 if(str.charAt(originalIndex) == ' '){  
 str.setCharAt(newIndex--, '0');  
 str.setCharAt(newIndex--, '2');  
 str.setCharAt(newIndex, '%');  
 }else{  
 str.setCharAt(newIndex, str.charAt(originalIndex));  
 }  
 originalIndex--;  
 newIndex--;  
 }  
  
 return str.toString();  
}

总结：要把思维打开，对于数组的操作我们习惯性的以for(int i = 0 ; i < arr.length ; i++)的形式从头到尾来操作数组，但是不要忽略了从尾到头遍历也有它的独到之处。

### 反转链表

#### 题目描述

输入一个链表，反转链表后，输出新链表的表头。

public ListNode ReverseList(ListNode head) {  
   
}

#### 解析

此题的难点在于无法通过一个单链表结点获取其前驱结点，因此我们不仅要在反转指针之前保存当前结点的前驱结点，还要保存当前结点的后继结点，并在下一次反转之前更新这两个指针。

/\*  
public class ListNode {  
 int val;  
 ListNode next = null;  
  
 ListNode(int val) {  
 this.val = val;  
 }  
}\*/  
public ListNode ReverseList(ListNode head) {  
 if(head == null || head.next == null){  
 return head;  
 }  
 ListNode pre = null, p = head, next;  
 while(p != null){  
 next = p.next;  
 p.next = pre;  
 pre = p;  
 p = next;  
 }  
  
 return pre;  
}

### 从尾到头打印链表

#### 题目描述

输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
   
}

#### 解析

此题的难点在于单链表只有指向后继结点的指针，因此我们无法通过当前结点获取前驱结点，因此不要妄想先遍历一遍链表找到尾结点然后再依次从后往前打印。

##### 递归，简洁优雅

由于我们通常是从头到尾遍历链表的，而题目要求从尾到头打印结点，这与前进后出的逻辑是相符的，因此你可以使用一个栈来保存遍历时走过的结点，再通过后进先出的特性实现从尾到头打印结点，但是我们也可以利用递归来帮我们压栈，由于递归简洁不易出错，因此面试中能用递归尽量用递归：只要当前结点不为空，就递归遍历后继结点，当后继结点为空时，递归结束，在递归回溯时将“当前结点”依次添加到集合中

/\*\*  
\* public class ListNode {  
\* int val;  
\* ListNode next = null;  
\*  
\* ListNode(int val) {  
\* this.val = val;  
\* }  
\* }  
\*  
\*/  
import java.util.ArrayList;  
public class Solution {  
 public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
 ArrayList<Integer> res = new ArrayList();  
 //input check  
 if(listNode == null){  
 return res;  
 }  
 recursively(res, listNode);  
 return res;  
 }  
  
 public void recursively(ArrayList<Integer> res, ListNode node){  
 //base case  
 if(node == null){  
 return;  
 }  
 //node not null  
 recursively(res, node.next);  
 res.add(node.val);  
 return;  
 }  
}

##### 反转链表

还有一种方法就是将链表指针都反转，这样将反转后的链表从头到尾打印就是结果了。需要注意的是我们不应该在访问用户数据时更改存储数据的结构，因此最后要记得反转回来：

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
 ArrayList<Integer> res = new ArrayList();  
 //input check  
 if(listNode == null){  
 return res;  
 }  
 return unrecursively(listNode);  
}  
  
public ArrayList<Integer> unrecursively(ListNode node){  
 ArrayList<Integer> res = new ArrayList<Integer>();  
 ListNode newHead = reverse(node);  
 ListNode p = newHead;  
 while(p != null){  
 res.add(p.val);  
 p = p.next;  
 }  
 reverse(newHead);  
 return res;  
}  
  
public ListNode reverse(ListNode node){  
 ListNode pre = null, cur = node, next;  
 while(cur != null){  
 //save predecessor  
 next = cur.next;  
 //reverse pointer  
 cur.next = pre;  
 //move to next  
 pre = cur;  
 cur = next;  
 }  
 //cur is null  
 return pre;  
}

总结：面试时能用递归就用递归，当然了如果面试官就是要考查你的指针功底那你也能just so so不是

### 重建二叉树

#### 题目描述

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,2,7,1,5,3,8,6}，则重建二叉树并返回。

public TreeNode reConstructBinaryTree(int [] pre,int [] in) {  
   
}

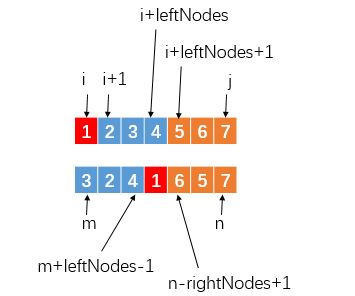
#### 解析

先序序列的特点是第一个数就是根结点而后是左子树的先序序列和右子树的先序序列，而中序序列的特点是先是左子树的中序序列，然后是根结点，最后是右子树的中序序列。因此我们可以通过先序序列得到根结点，然后通过在中序序列中查找根结点的索引从而得到左子树和右子树的结点数。然后可以将两序列都一分为三，对于其中的根结点能够直接重建，然后根据对应子序列分别递归重建根结点的左子树和右子树。这是一个典型的将复杂问题划分成子问题分步解决的过程。

递归体的定义，如上图先序序列的左子树序列是2,3,4对应下标1,2,3，而中序序列的左子树序列是3,2,4对应下标0,1,2，因此递归体接收的参数除了保存两个序列的数组之外，还需要指明需要递归重建的子序列分别在两个数组中的索引范围：TreeNode rebuild(int[] pre, int i, int j, int[] in, int m, int n)。然后递归体根据pre的i~j索引范围形成的先序序列和in的m~n索引范围形成的中序序列重建一棵树并返回根结点。

首先根结点就是先序序列的第一个数，即pre[i]，因此TreeNode root = new TreeNode(pre[i])可以直接确定，然后通过在in的m~n中查找出pre[i]的索引index可以求得左子树结点数leftNodes = index - m，右子树结点数rightNodes = n - index，如果左（右）子树结点数为0则表明左（右）子树为null，否则通过root.left = rebuild(pre, i' ,j' ,in ,m' ,n')来重建左（右）子树即可。

这个题的难点也就在这里，即i',j',m',n'的值的确定，笔者曾在此困惑许久，建议通过leftNodes,rightNodes和i,j,m,n来确定：（这个时候了前往不要在脑子里面想这些下标对应关系！！一定要在纸上画，确保准确性和概括性）



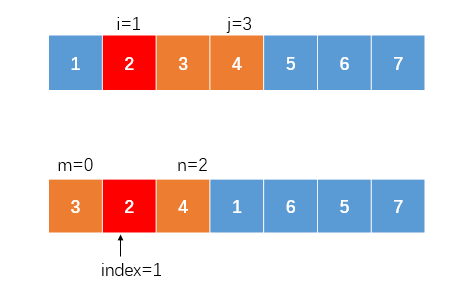
于是容易得出如下代码：

if(leftNodes == 0){  
 root.left = null;  
}else{  
 root.left = rebuild(pre, i + 1, i + leftNodes, in, m, m + leftNodes - 1);  
}  
if(rightNodes == 0){  
 root.right = null;  
}else{  
 root.right = rebuild(pre, i + leftNodes + 1, j, in, n - rightNodes + 1, n);  
}

笔者曾以中序序列的根节点索引来确定i',j',m',n'的对应关系写出如下**错误代码**：

if(leftNodes == 0){  
 root.left = null;  
}else{  
 root.left = rebuild(pre, i + 1, index, in, m, index - 1);  
}  
if(rightNodes == 0){  
 root.right = null;  
}else{  
 root.right = rebuild(pre, index + 1, j, in, index + 1, n);  
}

这种对应关系乍一看没错，但是不具有概括性（即囊括所有情况），比如对序列2,3,4、3,2,4重建时：



你看这种情况，上述错误代码还适用吗？原因就在于index是在in的m~n中选取的，与数组in是绑定的，和pre没有直接的关系，因此如果用index来表示i',j'自然是不合理的。

此题的正确完整代码如下：

/\*\*  
 \* Definition for binary tree  
 \* public class TreeNode {  
 \* int val;  
 \* TreeNode left;  
 \* TreeNode right;  
 \* TreeNode(int x) { val = x; }  
 \* }  
 \*/  
public class Solution {  
 public TreeNode reConstructBinaryTree(int [] pre,int [] in) {  
 if(pre == null || in == null || pre.length == 0 || in.length == 0 || pre.length != in.length){  
 return null;  
 }  
 return rebuild(pre, 0, pre.length - 1, in, 0, in.length - 1);  
 }  
   
 public TreeNode rebuild(int[] pre, int i, int j, int[] in, int m, int n){  
 int rootVal = pre[i], index = findIndex(rootVal, in, m, n);  
 if(index < 0){  
 return null;  
 }  
 int leftNodes = index - m, rightNodes = n - index;  
 TreeNode root = new TreeNode(rootVal);  
 if(leftNodes == 0){  
 root.left = null;  
 }else{  
 root.left = rebuild(pre, i + 1, i + leftNodes, in, m, m + leftNodes - 1);  
 }  
 if(rightNodes == 0){  
 root.right = null;  
 }else{  
 root.right = rebuild(pre, i + leftNodes + 1, j, in, n - rightNodes + 1, n);  
 }  
 return root;  
 }  
   
 public int findIndex(int target, int arr[], int from, int to){  
 for(int i = from ; i <= to ; i++){  
 if(arr[i] == target){  
 return i;  
 }  
 }  
 return -1;  
 }  
}

总结：

1. 对于复杂问题，一定要划分成若干子问题，逐一求解。比如二叉树问题，我们通常将其划分成头结点、左子树、右子树。
2. 对于递归过程的参数对应关系，尽量使用和数据样本本身没有直接关系的变量来表示。比如此题应该选取leftNodes和rightNodes来计算i',j',m',n'而不应该使用头结点在中序序列的下标index（它和in是绑定的，那么可能对pre就不适用了）。

### 用两个栈实现队列

#### 题目描述

用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

Stack<Integer> stack1 = new Stack<Integer>();  
Stack<Integer> stack2 = new Stack<Integer>();  
  
public void push(int node) {  
  
}  
  
public int pop() {  
   
}

#### 解析

这道题只要记住以下几点即可：

1. 一个栈（如stack1）只能用来存，另一个栈（如stack2）只能用来取
2. 当取元素时首先检查stack2是否为空，如果不空直接stack2.pop()，否则将stack1中的元素**全部倒入**stack2，如果倒入之后stack2仍为空则需要抛异常，否则stack2.pop()。

代码示例如下：

import java.util.Stack;  
  
public class Solution {  
 Stack<Integer> stack1 = new Stack<Integer>();  
 Stack<Integer> stack2 = new Stack<Integer>();  
   
 public void push(int node) {  
 stack1.push(node);  
 }  
   
 public int pop() {  
 if(stack2.empty()){  
 while(!stack1.empty()){  
 stack2.push(stack1.pop());  
 }  
 }  
 if(stack2.empty()){  
 throw new IllegalStateException("no more element！");  
 }  
 return stack2.pop();  
 }  
}

总结：只要取元素的栈不为空，取元素时直接弹出其栈顶元素即可，只有当其为空时才考虑将存元素的栈倒入进来，并且要一次性倒完。

### 旋转数组的最小数字

#### 题目描述

把一个数组最开始的**若干**个元素搬到数组的末尾，我们称之为数组的旋转。 输入一个**非减排序**的数组的一个旋转，输出旋转数组的最小元素。 例如数组{3,4,5,1,2}为{1,2,3,4,5}的一个旋转，该数组的最小值为1。 NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

public int minNumberInRotateArray(int [] arr) {  
   
}

#### 解析

此题需先认真审题：

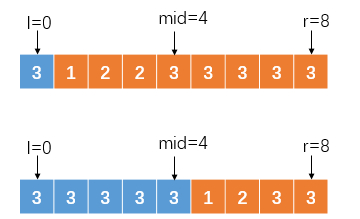
1. 若干，涵盖了一个元素都不搬的情况，此时数组是一个非减排序序列，因此首元素就是数组的最小元素。
2. 非减排序，并不代表是递增的，可能会出现若干相邻元素相同的情况，极端的例子是整个数组的所有元素都相同

由此不难得出如下input check：

public int minNumberInRotateArray(int [] arr) {  
 //input check  
 if(arr == null || arr.length == 0){  
 return 0;  
 }  
 //if only one element or no rotate  
 if(arr.length == 1 || arr[0] < arr[arr.length - 1]){  
 return arr[0];  
 }  
   
 //TODO  
}

上述的arr[0] < arr[arr.length - 1]不能写成arr[0] <= arr[arr.length - 1]，比如可能会有[1,2,3,3,4] -> [3,4,1,2,3] 的情况，这时你不能返回arr[0]=3。

如果走到了程序中的TODO，就可以考虑普遍情况下的推敲，数组可以被分成两部分：大于等于arr[0]的左半部分和小于等于arr[arr.length - 1]右半部分，我们不妨借助两个指针从数组的头、尾向中间靠近，这样就能利用二分的思想快速移动指针从而淘汰一些不在考虑范围之内的数。



如图，我们不能直接通过arr[mid]和arr[l]（或arr[r]）的比较（arr[mid] >= arr[l]）来决定移动l还是r到mid上，因为数组可能存在若干相同且相邻的数，因此我们还需要加上一个限制条件：arr[l + 1] >= arr[l] && arr[mid] >= arr[l]（对于r来说则是arr[r - 1] <= arr[r] && arr[mid] <= arr[r]），即当左半部分（右半部分）不止一个数时，我们才可能去移动l（r）指针。完整代码如下：

import java.util.ArrayList;  
public class Solution {  
 public int minNumberInRotateArray(int [] arr) {  
 //input check  
 if(arr == null || arr.length == 0){  
 return 0;  
 }  
 //if only one element or no rotate  
 if(arr.length == 1 || arr[0] < arr[arr.length - 1]){  
 return arr[0];  
 }  
   
 //has rotate, left part is big than right part  
 int l = 0, r = arr.length - 1, mid;  
 //l~r has more than 3 elements  
 while(r > l && r - l != 1){  
 //r-l >= 2 -> mid > l  
 mid = l + ((r - l) >> 1);  
 if(arr[l + 1] >= arr[l] && arr[mid] >= arr[l]){  
 l = mid;  
 }else{  
 r = mid;  
 }  
 }  
   
 return arr[r];  
 }  
}

总结：审题时要充分考虑数据样本的极端情况，以写出鲁棒性较强的代码。

### 斐波那契数列

#### 题目描述

大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0）。n<=39

public int Fibonacci(int n) {  
   
}

#### 解析

##### 递归方式

对于公式f(n) = f(n-1) + f(n-2)，明显就是一个递归调用，因此根据f(0) = 0和f(1) = 1我们不难写出如下代码：

public int Fibonacci(int n) {  
 if(n == 0 || n == 1){  
 return n;  
 }  
 return Fibonacci(n - 1) + Fibonacci(n - 2);  
}

##### 动态规划

在上述递归过程中，你会发现有很多计算过程是重复的：

**动态规划就在使用递归调用自上而下分析过程中发现有很多重复计算的子过程，于是采用自下而上的方式将每个子状态缓存下来，这样对于上层而言只有当需要的子过程结果不在缓存中时才会计算一次，因此每个子过程都只会被计算一次**。

public int Fibonacci(int n) {  
 if(n == 0 || n == 1){  
 return n;  
 }  
 //n1 -> f(n-1), n2 -> f(n-2)  
 int n1 = 1, n2 = 0;  
 //从f(2)开始算起  
 int N = 2, res = 0;  
 while(N++ <= n){  
 //每次计算后更新缓存，当然你也可以使用一个一维数组保存每次的计算结果，只额外空间复杂度就变为O(n)了  
 res = n1 + n2;  
 n2 = n1;  
 n1 = res;  
 }  
 return res;  
}

上述代码很多人都能写出来，只是没有意识到这就是动态规划。

总结：当你自上而下分析递归时发现有很多子过程被重复计算，那么就应该考虑能否通过自下而上将每个子过程的计算结果缓存下来。

### 跳台阶

#### 题目描述

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法（先后次序不同算不同的结果）。

public int JumpFloor(int target) {  
   
}

#### 解析

##### 递归版本

将复杂问题分解：复杂问题就是不断地将target减1或减2（对应跳一级和跳两级台阶）直到target变为1或2（对应只剩下一层或两层台阶）时我们能够很容易地得出结果。因此对于当前的青蛙而言，它能够选择的就是跳一级或跳二级，剩下的台阶有多少种跳法交给子过程来解决：

public int JumpFloor(int target) {  
 //input check  
 if(target <= 0){  
 return 0;  
 }  
 //base case  
 if(target == 1){  
 return 1;  
 }  
 if(target == 2){  
 return 2;  
 }  
 return JumpFloor(target - 1) + JumpFloor(target - 2);  
}

你会发现这其实就是一个斐波那契数列，只不过是从f(1) = 1,f(2) = 2开始的斐波那契数列罢了。自然你也应该能够写出动态规划版本。

#### 进阶问题

一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

#### 解析

##### 递归版本

本质上还是分解，只不过上一个是分解成两步，而这个是分解成n步：

public int JumpFloorII(int target) {  
 if(target <= 0){  
 return 0;  
 }  
 //base case，当target=0时表示某个分解分支跳完了所有台阶，这个分支就是一种跳法  
 if(target == 0){  
 return 1;  
 }  
   
 //本过程要收集的跳法的总数  
 int res = 0;  
 for(int i = 1 ; i <= target ; i++){  
 //本次选择，选择跳i阶台阶，剩下的台阶交给子过程，每个选择就代表一个分解分支  
 res += JumpFloorII(target - i);  
 }  
 return res;  
}

##### 动态规划

这个动态规划就有一点难度了，**首先我们要确定缓存目标**，斐波那契数列中由于f(n)只依赖于f(n-1)和f(n-2)因此我们仅用两个缓存变量实现了动态规划，但是这里f(n)依赖的是f(0),f(1),f(2),...,f(n-1)，因此我们需要通过长度量级为n的表缓存前n个状态（int arr[] = new int[target + 1]，arr[target]表示f(n)）。**然后根据递归版本（通常是base case）确定哪些状态的值是可以直接确定的**，比如由if(target == 0){ return 1 }可知arr[0] = 1，从f(N = 1)开始的所有状态都需要依赖之前（f(n < N)）的所有状态：

int res = 0;  
for(int i = 1 ; i <= target ; i++){  
 res += JumpFloorII(target - i);  
}  
return res

因此我们可以据此自下而上计算出每个子状态的值：

public int JumpFloorII(int target) {  
 if(target <= 0){  
 return 0;  
 }  
  
 int arr[] = new int[target + 1];  
 arr[0] = 1;  
 for(int i = 1 ; i < arr.length ; i++){  
 for(int j = 0 ; j < i ; j++){  
 arr[i] += arr[j];  
 }  
 }  
  
 return arr[target];  
}

但这仍不是最优解，因为观察循环体你会发现，每次f(n)的计算都要从f(0)累加到f(n-1)，我们完全可以将这个累加值缓存起来preSum，每计算出一次f(N)之后都将缓存更新为preSum += f(N)。如此得到最优解：

public int JumpFloorII(int target) {  
 if(target <= 0){  
 return 0;  
 }  
  
 int arr[] = new int[target + 1];  
 arr[0] = 1;  
 int preSum = arr[0];  
 for(int i = 1 ; i < arr.length ; i++){  
 arr[i] = preSum;  
 preSum += arr[i];  
 }  
  
 return arr[target];  
}

### 矩形覆盖

#### 题目描述

我们可以用2\*1的小矩形横着或者竖着去覆盖更大的矩形。请问用n个2\*1的小矩形无重叠地覆盖一个2\*n的大矩形，总共有多少种方法？

public int RectCover(int target) {  
   
}

#### 解析

##### 递归版本

有了之前的历练，我们能很快的写出递归版本：先竖着放一个或者先横着放两个，剩下的交给递归处理：

//target 大矩形的边长，也是剩余小矩形的个数  
public int RectCover(int target) {  
 if(target <= 0){  
 return 0;  
 }  
 if(target == 1 || target == 2){  
 return target;  
 }  
 return RectCover(target - 1) + RectCover(target - 2);  
}

##### 动态规划

这仍然是个以f(1)=1,f(2)=2开头的斐波那契数列：

//target 大矩形的边长，也是剩余小矩形的个数  
public int RectCover(int target) {  
 if(target <= 0){  
 return 0;  
 }  
 if(target == 1 || target == 2){  
 return target;  
 }  
 //n\_1->f(n-1), n\_2->f(n-2)，从f(N=3)开始算起  
 int n\_1 = 2, n\_2 = 1, N = 3, res = 0;  
 while(N++ <= target){  
 res = n\_1 + n\_2;  
 n\_2 = n\_1;  
 n\_1 = res;  
 }  
  
 return res;  
}

### 二进制中1的个数

#### 题目描述

输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。

public int NumberOf1(int n) {  
   
}

#### 解析

题目已经给我们降低了难度：负数用补码（取反加1）表示表明输入的参数为均为正数，我们只需统计其二进制表示中1的个数、运算时只考虑无符号移位即可。

典型的判断某个二进制位上是否为1的方法是将该二进制数右移至该二进制位为最低位然后与1相与&，由于1的二进制表示中只有最低位为1其余位均为0，因此相与后的结果与该二进制位上的数相同。据此不难写出如下代码：

public int NumberOf1(int n) {  
 int count = 0;  
 for(int i = 0 ; i < 32 ; i++){  
 count += ((n >> i) & 1);  
 }  
 return count;  
}

当然了，还有一种比较秀的解法就是利用n = n & (n - 1)将n的二进制位中为1的最低位置为0（只要n不为0就说明含有二进位制为1的位，如此这样的操作能做多少次就说明有多少个二进制位为1的位）：

public int NumberOf1(int n) {  
 int count = 0;  
 while(n != 0){  
 count++;  
 n &= (n - 1);  
 }  
 return count;  
}

### 数值的整数次方

#### 题目描述

给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。

public double Power(double base, int exponent) {  
   
}

#### 解析

这是一道充满危险色彩的题，求职者可能会内心窃喜不假思索的写出如下代码：

public double Power(double base, int exponent) {  
 double res = 1;  
 for(int i = 1 ; i <= exponent ; i++){  
 res \*= base;  
 }  
 return res;  
}

但是你有没有想过底数base和幂exponent都是可正、可负、可为0的。如果幂为负数，那么底数就不能为0，否则应该抛出算术异常：

//是否是负数  
boolean minus = false;  
//如果存在分母  
if(exponent < 0){  
 minus = true;  
 exponent = -exponent;  
 if(base == 0){  
 throw new ArithmeticException("/ by zero");  
 }  
}

如果幂为0，那么根据任何不为0的数的0次方为1，0的0次方未定义，应该有如下判断：

//如果指数为0  
if(exponent == 0){  
 if(base != 0){  
 return 1;  
 }else{  
 throw new ArithmeticException("0^0 is undefined");  
 }  
}

剩下的就是计算乘方结果，但是不要忘了如果幂为负需要将结果取倒数：

//指数不为0且分母也不为0，正常计算并返回整数或分数  
double res = 1;  
for(int i = 1 ; i <= exponent ; i++){  
 res \*= base;  
}  
  
if(minus){  
 return 1/res;  
}else{  
 return res;  
}

也许你还可以锦上添花为幂乘方的计算引入二分计算（当幂为偶数时2^n = 2^(n/2) \* 2^(n/2)）：

public double binaryPower(double base, int exp){  
 if(exp == 1){  
 return base;  
 }  
 double res = 1;  
 res \*= (binaryPower(base, exp/2) \* binaryPower(base, exp/2));  
 return exp % 2 == 0 ? res : res \* base;  
}

### 调整数组顺序使奇数位于偶数前面

#### 题目描述

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的**相对位置不变**。

public void reOrderArray(int [] arr) {  
   
}

#### 解析

读题之后发现这个跟快排的partition思路很像，都是选取一个比较基准将数组分成两部分，当然你也可以以arr[i] % 2 == 0为基准将奇数放前半部分，将偶数放有半部分，但是虽然只需O(n)的时间复杂度但不能保证调整后奇数之间、偶数之间的相对位置：

public void reOrderArray(int [] arr) {  
 if(arr == null || arr.length == 0){  
 return;  
 }  
  
 int odd = -1;  
 for(int i = 0 ; i < arr.length ; i++){  
 if(arr[i] % 2 == 1){  
 swap(arr, ++odd, i);  
 }  
 }  
}  
  
public void swap(int[] arr, int i, int j){  
 int temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
}

涉及到排序稳定性，我们自然能够想到插入排序，从数组的第二个元素开始向后依次确定每个元素应处的位置，确定的逻辑是：将该数与前一个数比较，如果比前一个数小则与前一个数交换位置并在交换位置后继续与前一个数比较直到前一个数小于等于该数或者已达数组首部停止。

此题不过是将比较的逻辑由数值的大小改为：当前的数是否是奇数并且前一个数是偶数，是则递归向前交换位置。代码示例如下：

public void reOrderArray(int [] arr) {  
 if(arr == null || arr.length == 0){  
 return;  
 }  
  
 int odd = -1;  
 for(int i = 1 ; i < arr.length ; i++){  
 for(int j = i ; j >= 1 ; j--){  
 if(arr[j] % 2 == 1 && arr[j - 1] % 2 == 0){  
 swap(arr, j, j - 1);  
 }  
 }  
 }  
}

### 链表中倒数第K个结点

#### 题目描述

输入一个链表，输出该链表中倒数第k个结点。

public ListNode FindKthToTail(ListNode head,int k) {  
   
}

#### 解析

**倒数**，这又是一个从尾到头的遍历逻辑，而链表对从尾到头遍历是敏感的，前面我们有通过压栈/递归、反转链表的方式实现这个遍历逻辑，自然对于此题同样适用，但是那样未免太麻烦了，我们可以通过两个间距为（k-1）个结点的链表指针来达到此目的。

public ListNode FindKthToTail(ListNode head,int k) {  
 //input check  
 if(head == null || k <= 0){  
 return null;  
 }  
 ListNode tmp = new ListNode(0);  
 tmp.next = head;  
 ListNode p1 = tmp, p2 = tmp;  
 while(k > 0 && p1.next != null){  
 p1 = p1.next;  
 k--;  
 }  
 //length < k  
 if(k != 0){  
 return null;  
 }  
 while(p1 != null){  
 p1 = p1.next;  
 p2 = p2.next;  
 }  
   
 tmp = null; //help gc  
  
 return p2;  
}

这里使用了一个技巧，就是创建一个临时结点tmp作为两个指针的初始指向，以模拟p1先走k步之后，p2才开始走，没走时停留在初始位置的逻辑，有利于帮我们梳理指针在对应位置上的意义，这样当p1走到头时（p1=null），p2就是倒数第k个结点。

这里还有一个坑就是，笔者层试图为了简化代码将上述的9 ~ 12行写成如下偷懒模式而导致排错许久：

while(k-- > 0 && p1.next != null){  
 p1 = p1.next;  
}

原因是将k--写在while()中，无论判断是否通过都会执行k = k - 1，因此代码总是会在if(k != 0)处返回null，希望读者不要和笔者一样粗心。

总结：当遇到复杂的指针操作时，我们不妨试图多引入几个指针或者临时结点，以方便梳理我们的思路，加强代码的逻辑化，这些空间复杂度O(1)的操作通常也不会影响性能。

### 合并两个排序的链表

#### 题目描述

输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

public ListNode Merge(ListNode list1,ListNode list2) {  
   
}

#### 解析

public ListNode Merge(ListNode list1,ListNode list2) {  
 if(list1 == null || list2 == null){  
 return list1 == null ? list2 : list1;  
 }  
 ListNode newHead = list1.val < list2.val ? list1 : list2;  
 ListNode p1 = (newHead == list1) ? list1.next : list1;  
 ListNode p2 = (newHead == list2) ? list2.next : list2;  
 ListNode p = newHead;  
 while(p1 != null && p2 != null){  
 if(p1.val <= p2.val){  
 p.next = p1;  
 p1 = p1.next;  
 }else{  
 p.next = p2;  
 p2 = p2.next;  
 }  
 p = p.next;  
 }  
  
 while(p1 != null){  
 p.next = p1;  
 p = p.next;  
 p1 = p1.next;  
 }  
 while(p2 != null){  
 p.next = p2;  
 p = p.next;  
 p2 = p2.next;  
 }  
  
 return newHead;  
}

### 树的子结构

#### 题目描述

输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）

/\*\*  
public class TreeNode {  
 int val = 0;  
 TreeNode left = null;  
 TreeNode right = null;  
  
 public TreeNode(int val) {  
 this.val = val;  
  
 }  
  
}\*/  
public boolean HasSubtree(TreeNode root1,TreeNode root2) {  
 if(root1 == null || root2 == null){  
 return false;  
 }  
  
 return process(root1, root2);  
}

#### 解析

这是一道典型的分解求解的复杂问题。典型的二叉树分解：遍历头结点、遍历左子树、遍历右子树。首先按照root1和root2的值是否相等划分为两种情况：

1. 两个头结点的值相等，并且root2.left也是roo1.left的子结构（递归）、root2.right也是root1.right的子结构（递归），那么可返回true。
2. 否则，要看只有当root2为root1.left的子结构或者root2为root1.right的子结构时，才能返回true

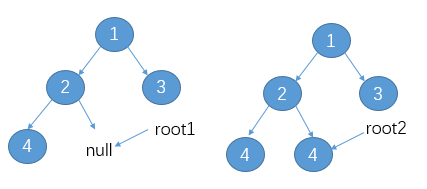
据上述两点很容易得出如下递归逻辑：

if(root1.val == root2.val){  
 if(process(root1.left, root2.left) && process(root1.right, root2.right)){  
 return true;  
 }  
}  
  
return process(root1.left, root2) || process(root1.right, root2);

接下来确定递归的终止条件，如果某个子过程root2=null那么说明在自上而下的比较过程中root2的结点已被罗列比较完了，这时无论root1是否为null，该子过程都应该返回true：

if(root2 == null){  
 return true;  
}

但是如果root2 != null而root1 = null，则应返回false



if(root1 == null && root2 != null){  
 return false;  
}

完整代码如下：

public class Solution {  
 public boolean HasSubtree(TreeNode root1,TreeNode root2) {  
 if(root1 == null || root2 == null){  
 return false;  
 }  
  
 return process(root1, root2);  
 }  
  
 public boolean process(TreeNode root1, TreeNode root2){  
 if(root2 == null){  
 return true;  
 }  
 if(root1 == null && root2 != null){  
 return false;  
 }   
  
 if(root1.val == root2.val){  
 if(process(root1.left, root2.left) && process(root1.right, root2.right)){  
 return true;  
 }  
 }  
  
 return process(root1.left, root2) || process(root1.right, root2);  
 }  
}

### 二叉树的镜像

#### 题目描述

操作给定的二叉树，将其变换为源二叉树的镜像。

public void Mirror(TreeNode root) {  
   
}

#### 解析

由图可知获取二叉树的镜像就是将原树的每个结点的左右孩子交换一下位置（这个规律一定要会找），也就是说我们只需遍历每个结点并交换left,right的引用指向就可以了，而我们有成熟的先序遍历：

public void Mirror(TreeNode root) {  
 if(root == null){  
 return;  
 }  
  
 TreeNode tmp = root.left;  
 root.left = root.right;  
 root.right = tmp;  
 Mirror(root.left);  
 Mirror(root.right);  
}

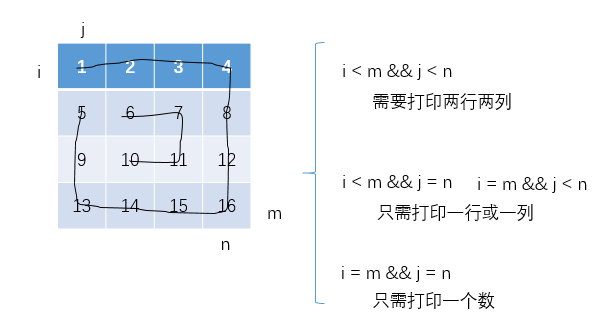
### 顺时针打印矩阵

#### 题目描述

输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下4 X 4矩阵： 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 则依次打印出数字1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

public ArrayList<Integer> printMatrix(int [][] matrix) {  
   
}

#### 解析



只要分析清楚了打印思路（左上角和右下角即可确定一条打印轨迹）后，此题主要考查条件控制的把握。只要给我一个左上角的点(i,j)和右下角的点(m,n)，就可以将这一圈的打印分解为四步：

但是如果左上角和右下角的点在一行或一列上那就没必要分解，直接打印改行或该列即可，打印的逻辑如下：

public void printEdge(int[][] matrix, int i, int j, int m, int n, ArrayList<Integer> res){  
 if(i == m && j == n){  
 res.add(matrix[i][j]);  
 return;  
 }  
  
 if(i == m || j == n){  
 //only one while will be execute  
 while(i < m){  
 res.add(matrix[i++][j]);  
 }  
 while(j < n){  
 res.add(matrix[i][j++]);  
 }  
 res.add(matrix[m][n]);  
 return;  
 }  
  
 int p = i, q = j;  
 while(q < n){  
 res.add(matrix[p][q++]);  
 }  
 //q == n  
 while(p < m){  
 res.add(matrix[p++][q]);  
 }  
 //p == m  
 while(q > j){  
 res.add(matrix[p][q--]);  
 }  
 //q == j  
 while(p > i){  
 res.add(matrix[p--][q]);  
 }  
 //p == i  
}

接着我们将每个圈的左上角和右下角传入该函数即可：

public ArrayList<Integer> printMatrix(int [][] matrix) {  
 ArrayList<Integer> res = new ArrayList<Integer>();  
 if(matrix == null || matrix.length == 0 || matrix[0] == null || matrix[0].length == 0){  
 return res;  
 }  
 int i = 0, j = 0, m = matrix.length - 1, n = matrix[0].length - 1;  
 while(i <= m && j <= n){  
 printEdge(matrix, i++, j++, m--, n--, res);  
 }  
 return res;  
}

### 包含min函数的栈

#### 题目描述

定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的min函数（时间复杂度应为O（1））。

public class Solution {  
  
   
 public void push(int node) {  
   
 }  
   
 public void pop() {  
   
 }  
   
 public int top() {  
   
 }  
   
 public int min() {  
   
 }  
}

#### 解析

最直接的思路是使用一个变量保存栈中现有元素的最小值，但这只对只存不取的栈有效，当弹出的值不是最小值时还没什么影响，但当弹出最小值后我们就无法获取当前栈中的最小值。解决思路是使用一个最小值栈，栈顶总是保存当前栈中的最小值，每次数据栈存入数据时最小值栈就要相应的将存入后的最小值压入栈顶：

private Stack<Integer> dataStack = new Stack();  
private Stack<Integer> minStack = new Stack();  
  
public void push(int node) {  
 dataStack.push(node);  
 if(!minStack.empty() && minStack.peek() < node){  
 minStack.push(minStack.peek());  
 }else{  
 minStack.push(node);  
 }  
}  
  
public void pop() {  
 if(!dataStack.empty()){  
 dataStack.pop();  
 minStack.pop();  
 }  
}  
  
public int top() {  
 if(!dataStack.empty()){  
 return dataStack.peek();  
 }  
 throw new IllegalStateException("stack is empty");  
}  
  
public int min() {  
 if(!dataStack.empty()){  
 return minStack.peek();  
 }  
 throw new IllegalStateException("stack is empty");  
}

### 栈的压入、弹出序列

#### 题目描述

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否可能为该栈的弹出顺序。假设压入栈的**所有数字均不相等**。例如序列1,2,3,4,5是某栈的压入顺序，序列4,5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。（注意：这两个序列的**长度是相等**的）

public boolean IsPopOrder(int [] arr1,int [] arr2) {  
   
}

#### 解析

可以使用两个指针i,j，初始时i指向压入序列的第一个，j指向弹出序列的第一个，试图将压入序列按照顺序压入栈中：

1. 如果arr1[i] != arr2[j]，那么将arr1[i]压入栈中并后移i（表示arr1[i]还没到该它弹出的时刻）
2. 如果某次后移i之后发现arr1[i] == arr2[j]，那么说明此刻的arr1[i]被压入后应该被立即弹出才会产生给定的弹出序列，于是不压入arr1[i]（表示压入并弹出了）并后移i，j也要后移（表示弹出序列的arr2[j]记录已产生，接着产生或许的弹出记录即可）。
3. 因为步骤2和3都会后移i，因此循环的终止条件是i到达arr1.length，此时若栈中还有元素，那么从栈顶到栈底形成的序列必须与arr2中j之后的序列相同才能返回true。

public boolean IsPopOrder(int [] arr1,int [] arr2) {  
 //input check  
 if(arr1 == null || arr2 == null || arr1.length != arr2.length || arr1.length == 0){  
 return false;  
 }  
 Stack<Integer> stack = new Stack();  
 int length = arr1.length;  
 int i = 0, j = 0;  
 while(i < length && j < length){  
 if(arr1[i] != arr2[j]){  
 stack.push(arr1[i++]);  
 }else{  
 i++;  
 j++;  
 }  
 }  
  
 while(j < length){  
 if(arr2[j] != stack.peek()){  
 return false;  
 }else{  
 stack.pop();  
 j++;  
 }  
 }  
  
 return stack.empty() && j == length;  
}

### 从上往下打印二叉树

#### 题目描述

从上往下打印出二叉树的每个节点，同层节点从左至右打印。

public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {  
   
}

#### 解析

使用一个队列来保存当前遍历结点的孩子结点，首先将根节点加入队列中，然后进行队列非空循环：

1. 从队列头取出一个结点，将该结点的值打印
2. 如果取出的结点左孩子不空，则将其左孩子放入队列尾部
3. 如果取出的结点右孩子不空，则将其右孩子放入队列尾部

public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {  
 ArrayList<Integer> res = new ArrayList<Integer>();  
 if(root == null){  
 return res;  
 }  
 LinkedList<TreeNode> queue = new LinkedList();  
 queue.addLast(root);  
 while(queue.size() > 0){  
 TreeNode node = queue.pollFirst();  
 res.add(node.val);  
 if(node.left != null){  
 queue.addLast(node.left);  
 }  
 if(node.right != null){  
 queue.addLast(node.right);  
 }  
 }  
  
 return res;  
}

### 二叉搜索树的后序遍历序列

#### 题目描述

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。

public boolean VerifySquenceOfBST(int [] sequence) {  
   
}

#### 解析

对于二叉树的后序序列，我们能够确定最后一个数就是根结点，还能确定的是前一半部分是左子树的后序序列，后一部分是右子树的后序序列。

遇到这种复杂问题，我们仍能采用三步走战略（根结点、左子树、右子树）：

1. 如果当前根结点的左子树**是BST**且其右子树也是BST，那么才可能是BST
2. 在1的条件下，如果左子树的**最大值**小于根结点且右子树的**最小值**大于根结点，那么这棵树就是BST

据此我们需要定义一个递归体，该递归体需要收集的信息如下：下层需要向我返回其最大值、最小值、以及是否是BST

class Info{  
 boolean isBST;  
 int max;  
 int min;  
 Info(boolean isBST, int max, int min){  
 this.isBST = isBST;  
 this.max = max;  
 this.min = min;  
 }  
}

递归体的定义如下：

public Info process(int[] arr, int start, int end){  
 if(start < 0 || end > arr.length - 1 || start > end){  
 throw new IllegalArgumentException("invalid input");  
 }  
 //base case : only one node  
 if(start == end){  
 return new Info(true, arr[end], arr[end]);  
 }  
  
 int root = arr[end];  
 Info left, right;  
 //not exist left child  
 if(arr[start] > root){  
 right = process(arr, start, end - 1);  
 return new Info(root < right.min && right.isBST,   
 Math.max(root, right.max), Math.min(root, right.min));  
 }  
 //not exist right child  
 if(arr[end - 1] < root){  
 left = process(arr, start, end - 1);  
 return new Info(root > left.max && left.isBST,   
 Math.max(root, left.max), Math.min(root, left.min));  
 }  
  
 int l = 0, r = end - 1;  
 while(r > l && r - l != 1){  
 int mid = l + ((r - l) >> 1);  
 if(arr[mid] > root){  
 r = mid;  
 }else{  
 l = mid;  
 }  
 }  
 left = process(arr, start, l);  
 right = process(arr, r, end - 1);  
 return new Info(left.isBST && right.isBST && root > left.max && root < right.min,   
 right.max, left.min);  
}

总结：二叉树相关的信息收集问题分步走：

1. 分析当前状态需要收集的信息
2. 根据下层传来的信息加工出当前状态的信息
3. 确定递归终止条件

### 二叉树中和为某一值的路径

#### 题目描述

输入一颗二叉树的跟节点和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下**一直到叶结点**所经过的结点形成一条路径。(注意: 在返回值的list中，数组长度大的数组靠前)

public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {  
   
}

#### 解析

审题可知，我们需要有一个自上而下从根结点到每个叶子结点的遍历思路，而先序遍历刚好可以拿来用，我们只需在来到当前结点时将当前结点值加入到栈中，在离开当前结点时再将栈中保存的当前结点的值弹出即可使用栈模拟保存自上而下经过的结点，从而实现在来到每个叶子结点时只需判断栈中数值之和是否为target即可。

public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {  
 ArrayList<ArrayList<Integer>> res = new ArrayList();  
 if(root == null){  
 return res;  
 }  
 Stack<Integer> stack = new Stack<Integer>();  
 preOrder(root, stack, 0, target, res);  
 return res;  
}  
  
public void preOrder(TreeNode root, Stack<Integer> stack, int sum, int target,   
 ArrayList<ArrayList<Integer>> res){  
 if(root == null){  
 return;  
 }  
  
 stack.push(root.val);  
 sum += root.val;  
 //leaf node  
 if(root.left == null && root.right == null && sum == target){  
 ArrayList<Integer> one = new ArrayList();  
 one.addAll(stack);  
 res.add(one);  
 }  
  
 preOrder(root.left, stack, sum, target, res);  
 preOrder(root.right, stack, sum, target, res);  
  
 sum -= stack.pop();  
}

### 复杂链表的复制

#### 题目描述

输入一个复杂链表（每个节点中有节点值，以及两个指针，一个指向下一个节点，另一个特殊指针指向任意一个节点），返回结果为复制后复杂链表的head。（注意，输出结果中请不要返回参数中的节点引用，否则判题程序会直接返回空）

/\*  
public class RandomListNode {  
 int label;  
 RandomListNode next = null;  
 RandomListNode random = null;  
  
 RandomListNode(int label) {  
 this.label = label;  
 }  
}  
\*/  
public class Solution {  
 public RandomListNode Clone(RandomListNode pHead)  
 {  
   
 }  
}

#### 解析

此题主要的难点在于random指针的处理。

##### 方法一：使用哈希表，额外空间O(n)

可以将链表中的结点都复制一份，用一个哈希表来保存，key是源结点，value就是副本结点，然后遍历key取出每个对应的value将副本结点的next指针和random指针设置好：

public RandomListNode Clone(RandomListNode pHead){  
 if(pHead == null){  
 return null;  
 }  
 HashMap<RandomListNode, RandomListNode> map = new HashMap();  
 RandomListNode p = pHead;  
 //copy  
 while(p != null){  
 RandomListNode cp = new RandomListNode(p.label);  
 map.put(p, cp);  
 p = p.next;  
 }  
 //link  
 p = pHead;  
 while(p != null){  
 RandomListNode cp = map.get(p);  
 cp.next = (p.next == null) ? null : map.get(p.next);  
 cp.random = (p.random == null) ? null : map.get(p.random);  
 p = p.next;  
 }  
  
 return map.get(pHead);  
}

##### 方法二：追加结点，额外空间O(1)

首先将每个结点复制一份并插入到对应结点之后，然后遍历链表将副本结点的random指针设置好，最后将源结点和副本结点分离成两个链表

public RandomListNode Clone(RandomListNode pHead){  
 if(pHead == null){  
 return null;  
 }  
  
 RandomListNode p = pHead;  
 while(p != null){  
 RandomListNode cp = new RandomListNode(p.label);  
 cp.next = p.next;  
 p.next = cp;  
 p = p.next.next;  
 }  
  
 //more than two node  
 //link random pointer  
 p = pHead;  
 RandomListNode cp;  
 while(p != null){  
 cp = p.next;  
 cp.random = (p.random == null) ? null : p.random.next;  
 p = p.next.next;  
 }  
  
 //split source and copy  
 p = pHead;  
 RandomListNode newHead = p.next;  
 //p != null -> p.next != null  
 while(p != null){  
 cp = p.next;  
 p.next = p.next.next;  
 p = p.next;  
 cp.next = (p == null) ? null : p.next;  
 }  
  
 return newHead;  
}

### 二叉搜索树与双向链表

#### 题目描述

输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

public TreeNode Convert(TreeNode root) {  
}

#### 解析

典型的二叉树分解问题，我们可以定义一个黑盒transform，它的目的是将二叉树转换成双向链表，那么对于一个当前结点root，首先将其前驱结点（BST中前驱结点指中序序列的前一个数值，也就是当前结点的左子树上最右的结点，如果左子树为空则没有前驱结点）和后继结点（当前结点的右子树上的最左结点，如果右子树为空则没有后继结点），然后使用黑盒transform将左子树和右子树转换成双向链表，最后将当前结点和左子树形成的链表链起来（通过之前保存的前驱结点）和右子树形成的链表链起来（通过之前保存的后继结点），整棵树的转换完毕。

public TreeNode Convert(TreeNode root) {  
 if(root == null){  
 return null;  
 }  
  
 //head is the most left node  
 TreeNode head = root;  
 while(head.left != null){  
 head = head.left;  
 }  
 transform(root);  
 return head;  
}  
  
//transform a tree to a double-link list  
public void transform(TreeNode root){  
 if(root == null){  
 return;  
 }  
 TreeNode pre = root.left, next = root.right;  
 while(pre != null && pre.right != null){  
 pre = pre.right;  
 }  
 while(next != null && next.left != null){  
 next = next.left;  
 }  
  
 transform(root.left);  
 transform(root.right);  
 //asume the left and right has transformed and what's remaining is link the root  
 root.left = pre;  
 if(pre != null){  
 pre.right = root;  
 }  
 root.right = next;  
 if(next != null){  
 next.left = root;  
 }  
}

### 字符串全排列

#### 题目描述

输入一个字符串,按字典序打印出该字符串中字符的所有排列。例如输入字符串abc,则打印出由字符a,b,c所能排列出来的所有字符串abc,acb,bac,bca,cab和cba。

#### 解析

定义一个递归体generate(char[] arr, int index, TreeSet<String> res)，其中char[] arr和index组合表示上层状态给当前状态传递的信息，即arr中0 ~ index-1是已生成好的串，现在你（当前状态）要确定index位置上应该放什么字符（你可以从index ~ arr.length - 1上任选一个字符），然后将index + 1应该放什么字符递归交给子过程处理，当某个状态要确定arr.length上应该放什么字符时说明0 ~ arr.length-1位置上的字符已经生成好了，因此递归终止，将生成好的字符串记录下来（这里由于要求不能重复且按字典序排列，因此我们可以使用JDK中红黑树的实现TreeSet来做容器）

public ArrayList<String> Permutation(String str) {  
 ArrayList<String> res = new ArrayList();  
 if(str == null || str.length() == 0){  
 return res;  
 }  
 TreeSet<String> set = new TreeSet();  
 generate(str.toCharArray(), 0, set);  
 res.addAll(set);  
 return res;  
}  
  
public void generate(char[] arr, int index, TreeSet<String> res){  
 if(index == arr.length){  
 res.add(new String(arr));  
 }  
 for(int i = index ; i < arr.length ; i++){  
 swap(arr, index, i);  
 generate(arr, index + 1, res);  
 swap(arr, index, i);  
 }  
}  
  
public void swap(char[] arr, int i, int j){  
 if(arr == null || arr.length == 0 || i < 0 || j > arr.length - 1){  
 return;  
 }  
 char tmp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = tmp;  
}

注意：上述代码的第19行有个坑，笔者曾因忘记写第19行而排错许久，由于你任选一个index ~ arr.length - 1位置上的字符与index位置上的交换并将交换生成的结果交给了子过程（第17,18行），但你不应该影响后续选取其他字符放到index位置上而形成的结果，因此需要再交换回来（第19行）

### 数组中出现次数超过一半的数

#### 题目描述

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为9的数组{1,2,3,2,2,2,5,4,2}。由于数字2在数组中出现了5次，超过数组长度的一半，因此输出2。如果不存在则输出0。

public int MoreThanHalfNum\_Solution(int [] arr) {  
}

#### 解析

##### 方法一：基于partition查找数组中第k大的数

如果我们将数组排序，最快也要O(nlogn)，排序后的中位数自然就是出现次数超过长度一半的数。

我们知道快排的partition操作能够将数组按照一个基准划分成小于部分和大于等于部分并返回这个基准在数组中的下标，虽然一次partition并不能使数组整体有序，但是能够返回随机选择的数在partition之后的下标index，这个下标标识了它是第index大的数，这也意味着我们要求数组中第k大的数不一定要求数组整体有序。

于是我们在首次对整个数组partition之后将返回的index与n/2进行比较，并调整下一次partition的范围直到index = n/2为止我们就找到了。

这个时间复杂度需要使用Master公式计算（计算过程参见 http://www.zhenganwen.top/62859a9a.html#Master%E5%85%AC%E5%BC%8F），**使用partition查找数组中第k大的数时间复杂度为O(n)**，最后不要忘了验证一下index = n/2上的数出现的次数是否超过了长度的一半。

public int MoreThanHalfNum\_Solution(int [] arr) {  
 if(arr == null || arr.length == 0){  
 return 0;  
 }  
 if(arr.length == 1){  
 return arr[0];  
 }  
  
 int index = partition(arr, 0, arr.length - 1);  
 int half = arr.length >> 1;// 0 <= half <= arr.length - 1  
 while(index != half){  
 index = index > k ? partition(arr, 0, index - 1) : partition(arr, index + 1, arr.length - 1);  
 }  
  
 int count = 0;  
 for(int i = 0 ; i < arr.length ; i++){  
 count = (arr[i] == arr[index]) ? ++count : count;  
 }  
  
 return (count > arr.length / 2) ? arr[index] : 0;  
}  
  
public int partition(int[] arr, int start, int end){  
 if(arr == null || arr.length == 0 || start < 0 || end > arr.length - 1 || start > end){  
 throw new IllegalArgumentException();  
 }  
 if(start == end){  
 return end;  
 }  
 int random = start + (int)(Math.random() \* (end - start + 1));  
 swap(arr, random, end);  
 int small = start - 1;  
 for(int i = start ; i < end ; i++){  
 if(arr[i] < arr[end]){  
 swap(arr, ++small, i);  
 }  
 }  
  
 swap(arr, ++small, end);  
  
 return small;  
}  
  
public void swap(int[] arr, int i, int j){  
 int t = arr[i];  
 arr[i] = arr[j];  
 arr[j] = t;  
}

##### 方法二

1. 使用一个target记录一个数，并使用count记录它出现的次数
2. 初始时target = arr[0]，count = 1，表示arr[0]出现了1次
3. 从第二个元素开始遍历数组，如果遇到的数不等于target就将count减1，否则加1
4. 如果遍历到某个数时，count为0了，那么就将target设置为该数，并将count置1，继续向后遍历

如果存在出现次数超过一半的数，那么必定是target最后一次被设置时的数。

public int MoreThanHalfNum\_Solution(int [] arr) {  
 if(arr == null || arr.length == 0){  
 return 0;  
 }  
 //此题需要抓住出现次数超过数组长度的一半这个点来想  
 //使用一个计数器，如果这个数出现一次就自增，否则自减，如果自减为0则更新被记录的数  
 //如果存在出现次数大于一半的数，那么最后一次被记录的数就是所求之数  
 int target = arr[0], count = 1;  
 for(int i = 1 ; i < arr.length ; i++){  
 if(count == 0){  
 target = arr[i];  
 count = 1;  
 }else{  
 count = (arr[i] == target) ? ++count : --count;  
 }  
 }  
  
 if(count == 0){  
 return 0;  
 }  
  
 //不要忘了验证！！！  
 count = 0;  
 for(int i = 0 ; i < arr.length ; i++){  
 count = (arr[i] == target) ? ++count : count;  
 }  
  
 return (count > arr.length / 2) ? target : 0;  
}

### 最小的k个数

#### 题目描述

输入n个整数，找出其中最小的K个数。例如输入4,5,1,6,2,7,3,8这8个数字，则最小的4个数字是1,2,3,4,。

public ArrayList<Integer> GetLeastNumbers\_Solution(int [] arr, int k) {  
   
}

#### 解析

与上一题的求数组第k大的数如出一辙，如果某次partition之后你得到了第k大的数的下标，那么根据partitin规则该下标左边的数均比该下标上的数小，最小的k个数自然就是此时的0~k-1下标上的数

public ArrayList<Integer> GetLeastNumbers\_Solution(int [] arr, int k) {  
 ArrayList<Integer> res = new ArrayList<Integer>();  
 if(arr == null || arr.length == 0 || k <= 0 || k > arr.length){  
 //throw new IllegalArgumentException();  
 return res;  
 }  
  
 int index = partition(arr, 0, arr.length - 1);  
 k = k - 1;  
 while(index != k){  
 index = index > k ? partition(arr, 0, index - 1) : partition(arr, index + 1, arr.length - 1);  
 }  
  
 for(int i = 0 ; i <= k ; i++){  
 res.add(arr[i]);  
 }  
  
 return res;  
}  
  
public int partition(int[] arr, int start, int end){  
 if(arr == null || arr.length == 0 || start < 0 || end > arr.length - 1 || start > end){  
 throw new IllegalArgumentException();  
 }  
 if(start == end){  
 return end;  
 }  
  
 int random = start + (int)(Math.random() \* (end - start + 1));  
 swap(arr, random, end);  
 int small = start - 1;  
 for(int i = start ; i < end ; i++){  
 if(arr[i] < arr[end]){  
 swap(arr, ++small, i);  
 }  
 }  
  
 swap(arr, ++small, end);  
 return small;  
}  
  
public void swap(int[] arr, int i, int j){  
 int t = arr[i];  
 arr[i] = arr[j];  
 arr[j] = t;  
}

### 连续子数组的最大和

#### 题目描述

HZ偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后,他又发话了:在古老的一维模式识别中,常常需要计算连续子向量的最大和,当向量全为正数的时候,问题很好解决。但是,如果向量中包含负数,是否应该包含某个负数,并期望旁边的正数会弥补它呢？例如:{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为8(从第0个开始,到第3个为止)。给一个数组，返回它的最大连续子序列的和，你会不会被他忽悠住？(子向量的长度至少是1)

public int FindGreatestSumOfSubArray(int[] arr) {  
   
}

#### 解析

##### 暴力解

暴力法是找出所有子数组，然后遍历求和，时间复杂度为O(n^3)

public int FindGreatestSumOfSubArray(int[] arr) {  
 if(arr == null || arr.length == 0){  
 return 0;  
 }  
 int max = Integer.MIN\_VALUE;  
  
 //start  
 for(int i = 0 ; i < arr.length ; i++){  
 //end  
 for(int j = i ; j < arr.length ; j++){  
 //sum  
 int sum = 0;  
 for(int k = i ; k <= j ; k++){  
 sum += arr[k];  
 }  
 max = Math.max(max, sum);  
 }  
 }  
  
 return max;  
}

##### 最优解

使用一个sum记录累加和，初始时为0，遍历数组：

1. 如果遍历到i时，发现sum小于0，那么丢弃这个累加和，将sum重置为0
2. 将当前元素累加到sum上，并更新最大和maxSum

public int FindGreatestSumOfSubArray(int[] arr) {  
 if(arr == null || arr.length == 0){  
 return 0;  
 }  
 int sum = 0, max = Integer.MIN\_VALUE;  
 for(int i = 0 ; i < arr.length ; i++){  
 if(sum < 0){  
 sum = 0;  
 }  
 sum += arr[i];  
 max = Math.max(max, sum);  
 }  
  
 return max;  
}

### 整数中1出现的次数（从1到n整数中1出现的次数）

#### 题目描述

求出1~13的整数中1出现的次数,并算出100~1300的整数中1出现的次数？为此他特别数了一下1~13中包含1的数字有1、10、11、12、13因此共出现6次,但是对于后面问题他就没辙了。ACMer希望你们帮帮他,并把问题更加普遍化,可以很快的求出任意非负整数区间中1出现的次数（从1 到 n 中1出现的次数）。

#### 解析

##### 遍历一遍不就完了吗

当然，你可从1遍历到n，然后将当前被遍历的到的数中1出现的次数累加到结果中可以很容易地写出如下代码：

public int NumberOf1Between1AndN\_Solution(int n) {  
 if(n < 1){  
 return 0;  
 }  
 int res = 0;  
 for(int i = 1 ; i <= n ; i++){  
 res += count(i);  
 }  
 return res;  
}  
  
public int count(int n){  
 int count = 0;  
 while(n != 0){  
 //取个位  
 count = (n % 10 == 1) ? ++count : count;  
 //去掉个位  
 n /= 10;  
 }  
 return count;  
}

但n多大就会循环多少次，这并不是面试官所期待的，这时我们就需要找规律看是否有捷径可走

##### 不用数我也知道

以51234这个数为例，我们可以先将51234划分成1~1234（去掉最高位）和1235~51234两部分来求解。下面先分析1235~51234这个区间的结果：

1. 所有的数中，1在最高位（万位）出现的次数

* 对于1235~51234，最高位为1时（即万位为1时）的数有10000~19999这10000个数，也就是说1在最高位（万位）出现的次数为10000，因此我们可以得出结论：如果最高位大于1，那么在最高位上1出现的次数为最高位对应的单位（本例中为一万次）；但如果最高位为1，比如1235~11234，那么次数就为去掉最高位之后的数了，11234去掉最高位后是1234，即1在最高位上出现的次数为1234

1. 所有的数中，1在非最高位上出现的次数

* 我们可以进一步将1235~51234按照最高位的单位划分成4个区间（能划分成几个区间由最高位上的数决定，这里最高位为5，所以能划分5个大小为一万子区间）：
  + 1235~11234
  + 11235~21234
  + 21235~31234
  + 31235~41234
  + 41235~51234
* 而每个数不考虑万位（因为1在万位出现的总次数在步骤1中已统计好了），其余四位（个、十、百、千）取一位放1（比如千位），剩下的3位从0~9中任意选（10 \* 10 \* 10），那么仅统计1在千位上出现的次数之和就是：5(子区间数) \* 10 \* 10 \* 10，还有百位、十位、个位，结果为：4 \* 10 \* 10 \* 10 \* 5。
* 因此非高位上1出现的总次数的计算通式为：(n-1) \* 10^(n-2) \* 十进制最高位上的数（其中n为十进制的总位数）
* 于是1235 ~ 51234之间所有的数的所有的位上1出现的次数的综合我们就计算出来了

剩下1 ~ 1234，你会发现这与1 ~ 51234的问题是一样的，因此可以做递归处理，即子过程也会将1 ~ 1234也分成1 ~ 234和235 ~ 1234两部分，并计算235~1234而将1~234又进行递归处理。

而递归的终止条件如下：

1. 如果1~n中的n：1 <= n <= 9，那么就可以直接返回1了，因为只有数1出现了一次1
2. 如果n == 0，比如将10000划分成的两部分是0 ~ 0(10000去掉最高位后的结果)和1 ~ 10000，那么就返回0

public int NumberOf1Between1AndN\_Solution(int n) {  
 if(n < 1){  
 return 0;  
 }  
 return process(n);  
}  
  
public int process(int n){  
 if(n == 0){  
 return 0;  
 }  
 if(n < 10 && n > 0){  
 return 1;  
 }  
 int res = 0;  
 //得到十进制位数  
 int bitCount = bitCount(n);  
 //十进制最高位上的数  
 int highestBit = numOfBit(n, bitCount);  
 //1、统计最高位为1时，共有多少个数  
 if(highestBit > 1){  
 res += powerOf10(bitCount - 1);  
 }else{  
 //highestBit == 1  
 res += n - powerOf10(bitCount - 1) + 1;  
 }  
 //2、统计其它位为1的情况  
 res += powerOf10(bitCount - 2) \* (bitCount - 1) \* highestBit;  
 //3、剩下的部分交给递归  
 res += process(n % powerOf10(bitCount - 1));  
 return res;  
}  
  
//返回10的n次方  
public int powerOf10(int n){  
 if(n == 0){  
 return 1;  
 }  
 boolean minus = false;  
 if(n < 0){  
 n = -n;  
 minus = true;  
 }  
 int res = 1;  
 for(int i = 1 ; i <= n ; i++){  
 res \*= 10;  
 }  
 return minus ? 1 / res : res;  
}  
  
public int bitCount(int n){  
 int count = 1;  
 while((n /= 10) != 0){  
 count++;  
 }  
 return count;  
}  
  
public int numOfBit(int n, int bit){  
 while(bit-- > 1){  
 n /= 10;  
 }  
 return n % 10;  
}

笔者曾纠结，对于一个四位数，每个位上出现1时都统计了一遍会不会有重复，比如11111这个数在最高位为1时的10000 ~ 19999统计了一遍，在统计非最高位的其他位上为1时又统计了4次，总共被统计了5次，而这个数1出现的次数也确实是5次，因此没有重复。

### 把数组排成最小的数

#### 题目描述

输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。例如输入数组{3，32，321}，则打印出这三个数字能排成的最小数字为321323。

#### 解析

这是一个贪心问题，你发现将数组按递增排序之后依次连接起来的结果并不是最优的结果，于是需要寻求贪心策略，对于这类最小数和最小字典序的问题而言，贪心策略是：如果3和32相连的结果大于32和3相连的结果，那么视作3比32大，最后我们需要按照按照这种策略将数组进行升序排序，以得到首尾相连之后的结果是最小数字（最小字典序）。

public String PrintMinNumber(int [] numbers) {  
 if(numbers == null || numbers.length == 0){  
 return "";  
 }  
 List<Integer> list = new ArrayList();  
 for(int num : numbers){  
 list.add(num);  
 }  
 Collections.sort(list, new MyComparator());  
 StringBuilder res = new StringBuilder("");  
 for(Integer integer : list){  
 res.append(integer.toString());  
 }  
 return res.toString();  
}  
  
class MyComparator implements Comparator<Integer>{  
 public int compare(Integer i1, Integer i2){  
 String s1 = i1.toString() + i2.toString();  
 String s2 = i2.toString() + i1.toString();  
 return Integer.parseInt(s1) - Integer.parseInt(s2);  
 }  
}

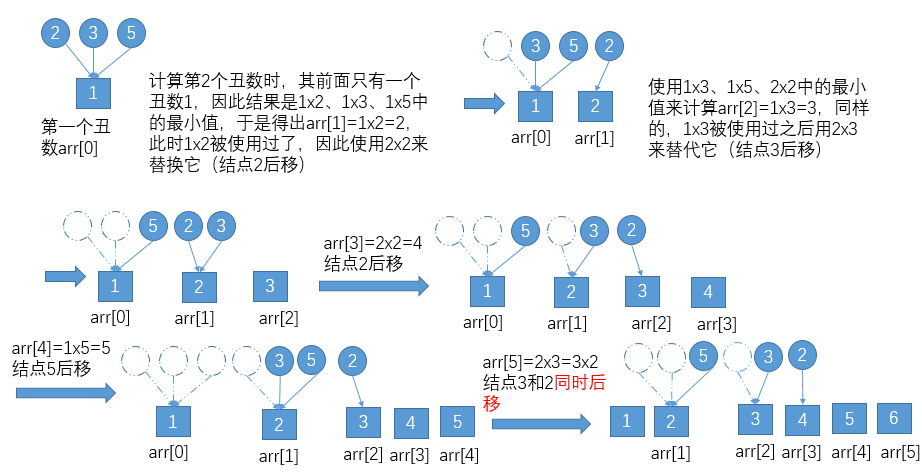
### 丑数

#### 题目描述

把只包含质因子2、3和5的数称作丑数（Ugly Number）。例如6、8都是丑数，但14不是，因为它包含质因子7。 习惯上我们把1当做是第一个丑数。求按从小到大的顺序的第N个丑数。

#### 解析

老实说，在《剑指offer》上看这道题的时候每太看懂，以至于第一遍在牛客网OJ这道题的时候都是背下来写上去的，直到这第二遍总结时才弄清整个思路，思路的核心就是第一个丑数是1（题目给的），此后的每一个丑数都是由之前的某个丑数与2或3或5的乘积得来



public int GetUglyNumber\_Solution(int index) {  
 if(index < 1){  
 //throw new IllegalArgumentException("index must bigger than one");  
 return 0;  
 }  
 if(index == 1){  
 return 1;  
 }  
  
 int[] arr = new int[index];  
 arr[0] = 1;  
 int indexOf2 = 0, indexOf3 = 0, indexOf5 = 0;  
 for(int i = 1 ; i < index ; i++){  
 arr[i] = Math.min(arr[indexOf2] \* 2, Math.min(arr[indexOf3] \* 3, arr[indexOf5] \* 5));  
 indexOf2 = (arr[indexOf2] \* 2 <= arr[i]) ? ++indexOf2 : indexOf2;  
 indexOf3 = (arr[indexOf3] \* 3 <= arr[i]) ? ++indexOf3 : indexOf3;  
 indexOf5 = (arr[indexOf5] \* 5 <= arr[i]) ? ++indexOf5 : indexOf5;  
 }  
  
 return arr[index - 1];  
}

### 第一个只出现一次的字符

#### 题目描述

在一个字符串(0<=字符串长度<=10000，**全部由字母组成**)中找到第一个只出现一次的字符,并返回它的位置, 如果没有则返回 -1（需要区分大小写）.

#### 解析

可以从头遍历字符串，并使用一个表记录每个字符第一次出现的位置（初始时表中记录的位置均为-1），如果记录当前被遍历字符出现的位置时发现之前已经记录过了（通过查表，该字符的位置不是-1而是大于等于0的一个有效索引），那么当前字符不在答案的考虑范围内，通过将表中该字符的出现索引标记为-2来标识。

遍历一遍字符串并更新表之后，再遍历一遍字符串，如果发现某个字符在表中对应的记录是一个有效索引（大于等于0），那么该字符就是整个串中第一个只出现一次的字符。

由于题目标注字符串全都由字母组成，而字母可以使用ASCII码表示且ASCII范围为0~255，因此使用了一个长度为256的数组来实现这张表。用字母的ASCII值做索引，索引对应的值就是字母在字符串中第一次出现的位置（初始时为-1，第一次遇到时设置为出现的位置，重复遇到时置为-2）。

public int FirstNotRepeatingChar(String str) {  
 if(str == null || str.length() == 0){  
 return -1;  
 }  
 //全部由字母组成  
 int[] arr = new int[256];  
 for(int i = 0 ; i < arr.length ; i++){  
 arr[i] = -1;  
 }  
 for(int i = 0 ; i < str.length() ; i++){  
 int ascii = (int)str.charAt(i);  
 if(arr[ascii] == -1){  
 //set index of first apearance  
 arr[ascii] = i;  
 }else if(arr[ascii] >= 0){  
 //repeated apearance, don't care  
 arr[ascii] = -2;  
 }  
 //arr[ascii] == -2 -> do not care  
 }  
  
 for(int i = 0 ; i < str.length() ; i++){  
 int ascii = (int)str.charAt(i);  
 if(arr[ascii] >= 0){  
 return arr[ascii];  
 }  
 }  
  
 return -1;  
}

### 数组中的逆序对

#### 题目描述

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数P。并将P对1000000007取模的结果输出。 即输出P%1000000007

public int InversePairs(int [] arr) {  
 if(arr == null || arr.length <= 1){  
 return 0;  
 }  
 return mergeSort(arr, 0, arr.length - 1).pairs;  
}

#### 输入描述

1. 题目保证输入的数组中没有相同的数字
2. 数据范围：对于%50的数据,size<=10^4；对于%75的数据,size<=10^5；对于%100的数据,size<=2\*10^5

#### 解析

借助归并排序的流程，将归并流程中前一个数组的数比后一个数组的数小的情况记录下来。

归并的原始逻辑是根据输入的无序数组返回一个新建的排好序的数组：

public int[] mergeSort(int[] arr, int start, int end){  
 if(arr == null || arr.length == 0 || start < 0 || end > arr.length - 1 || start > end){  
 throw new IllegalArgumentException();  
 }  
 if(start == end){  
 return new int[]{ arr[end] };  
 }  
  
 int[] arr1 = mergeSort(arr, start, mid);  
 int[] arr2 = Info right = mergeSort(arr, mid + 1, end);  
 int[] copy = new int[arr1.length + arr2.length];  
 int p1 = 0, p2 = 0, p = 0;  
  
 while(p1 < arr1.length && p2 < arr2.length){  
 if(arr1[p1] > arr2[p2]){  
 copy[p++] = arr1[p1++];  
 }else{  
 copy[p++] = arr2[p2++];  
 }  
 }  
 while(p1 < arr1.length){  
 copy[p++] = arr1[p1++];  
 }  
 while(p2 < arr2.length){  
 copy[p++] = arr2[p2++];  
 }  
 return copy;  
}

而我们需要再此基础上对子状态收集的信息进行改造，假设左右两半部分分别有序了，那么进行merge的时候，不应是从前往后复制了，这样当arr1[p1] > arr2[p2]的时候并不知道arr2的p2后面还有多少元素是比arr1[p1]小的，要想一次比较就统计出arr2中所有比arr1[p1]小的数需要将p1,p2从arr1,arr2的尾往前遍历：

而将比较后较大的数移入辅助数组的逻辑还是一样。这样当前递归状态需要收集左半子数组和右半子数组的变成有序过程中记录的逆序对数和自己merge记录的逆序对数之和就是当前状态要返回的信息，并且merge后形成的有序辅助数组也要返回。

public int InversePairs(int [] arr) {  
 if(arr == null || arr.length <= 1){  
 return 0;  
 }  
 return mergeSort(arr, 0, arr.length - 1).pairs;  
}  
  
class Info{  
 int arr[];  
 int pairs;  
 Info(int[] arr, int pairs){  
 this.arr = arr;  
 this.pairs = pairs;  
 }  
}  
  
public Info mergeSort(int[] arr, int start, int end){  
 if(arr == null || arr.length == 0 || start < 0 || end > arr.length - 1 || start > end){  
 throw new IllegalArgumentException();  
 }  
 if(start == end){  
 return new Info(new int[]{arr[end]}, 0);  
 }  
  
 int pairs = 0;  
 int mid = start + ((end - start) >> 1);  
 Info left = mergeSort(arr, start, mid);  
 Info right = mergeSort(arr, mid + 1, end);  
 pairs += (left.pairs + right.pairs) % 1000000007;  
  
 int[] arr1 = left.arr, arr2 = right.arr, copy = new int[arr1.length + arr2.length];  
 int p1 = arr1.length - 1, p2 = arr2.length - 1, p = copy.length - 1;  
  
 while(p1 >= 0 && p2 >= 0){  
 if(arr1[p1] > arr2[p2]){  
 pairs += (p2 + 1);  
 pairs %= 1000000007;  
 copy[p--] = arr1[p1--];  
 }else{  
 copy[p--] = arr2[p2--];  
 }  
 }  
  
 while(p1 >= 0){  
 copy[p--] = arr1[p1--];  
 }  
 while(p2 >= 0){  
 copy[p--] = arr2[p2--];  
 }  
  
 return new Info(copy, pairs % 1000000007);  
}

### 两个链表的第一个公共结点

#### 题目描述

输入两个链表，找出它们的第一个公共结点。

public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {  
  
}

#### 解析

首先我们要分析两个链表的组合状态，根据有环、无环相互组合只可能会出现如下几种情况：

于是我们首先要判断两个链表是否有环，判断链表是否有环以及有环链表的入环结点在哪已有前人给我们总结好了经验：

1. 使用一个快指针和一个慢指针同时从首节点出发，快指针一次走两步而慢指针一次走一步，如果两指针相遇则说明有环，否则无环
2. 如果两指针相遇，先将快指针重新指向首节点，然后两指针均一次走一步，再次相遇时的结点就是入环结点

public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {  
 //若其中一个链表为空则不存在相交问题  
 if(pHead1 == null || pHead2 == null){  
 return null;  
 }  
 ListNode ringNode1 = ringNode(pHead1);  
 ListNode ringNode2 = ringNode(pHead2);  
 //如果一个有环，另一个无环  
 if((ringNode1 == null && ringNode2 != null) ||  
 (ringNode1 != null && ringNode2 == null)){  
 return null;  
 }  
 //如果两者都无环，判断是否共用尾结点  
 else if(ringNode1 == null && ringNode2 == null){  
 return firstCommonNode(pHead1, pHead2, null);  
 }  
 //剩下的情况就是两者都有环了  
 else{  
 //如果入环结点相同，那么第一个相交的结点肯定在入环结点之前  
 if(ringNode1 == ringNode2){  
 return firstCommonNode(pHead1, pHead2, ringNode1);  
 }  
 //如果入环结点不同，看能否通过ringNode1的后继找到ringNode2  
 else{  
 ListNode p = ringNode1;  
 while(p.next != ringNode1){  
 p = p.next;  
 if(p == ringNode2){  
 break;  
 }  
 }  
 //如果能找到，那么第一个相交的结点既可以是ringNode1也可以是ringNode2  
 return (p == ringNode2) ? ringNode1 : null;  
 }  
 }  
}  
  
//查找两链表的第一个公共结点，如果两链表无环，则传入common=null，如果都有环且入环结点相同，那么传入common=入环结点  
public ListNode firstCommonNode(ListNode pHead1, ListNode pHead2, ListNode common){  
 ListNode p1 = pHead1, p2 = pHead2;  
 int len1 = 1, len2 = 1, gap = 0;  
 while(p1.next != common){  
 p1 = p1.next;  
 len1++;  
 }  
 while(p2.next != common){  
 p2 = p2.next;  
 len2++;  
 }  
 //如果是两个无环链表，要判断一下是否有公共尾结点  
 if(common == null && p1 != p2){  
 return null;  
 }  
 gap = len1 > len2 ? len1 - len2 : len2 - len1;  
 //p1指向长链表，p2指向短链表  
 p1 = len1 > len2 ? pHead1 : pHead2;  
 p2 = len1 > len2 ? pHead2 : pHead1;  
 while(gap-- > 0){  
 p1 = p1.next;  
 }  
 while(p1 != p2){  
 p1 = p1.next;  
 p2 = p2.next;  
 }  
 return p1;  
}  
  
//判断链表是否有环，没有返回null，有则返回入环结点（整个链表是一个环时入环结点就是头结点）  
public ListNode ringNode(ListNode head){  
 if(head == null){  
 return null;  
 }  
 ListNode p1 = head, p2 = head;  
 while(p1.next != null && p1.next.next != null){  
 p1 = p1.next.next;  
 p2 = p2.next;  
 if(p1 == p2){  
 break;  
 }  
 }  
  
 if(p1.next == null || p1.next.next == null){  
 return null;  
 }  
  
 p1 = head;  
 while(p1 != p2){  
 p1 = p1.next;  
 p2 = p2.next;  
 }  
 //可能整个链表就是一个环，这时入环结点就是头结点！！！  
 return p1 == p2 ? p1 : head;  
}

### 数字在排序数组中出现的次数

#### 题目描述

统计一个数字在排序数组中出现的次数。

public int GetNumberOfK(int [] array , int k) {  
  
}

#### 解析

我们可以分两步解决，先找出数值为k的连续序列的左边界，再找右边界。可以采用二分的方式，以查找左边界为例：如果arr[mid]小于k那么移动左指针，否则移动右指针（初始时左指针指向-1，而右指针指向尾元素arr.length），当两个指针相邻时，左指针及其左边的数均小于k而右指针及其右边的数均大于或等于k，因此此时右指针就是要查找的左边界，同样的方式可以求得右边界。

值得注意的是，笔者曾将左指针初始化为0而右指针初始化为arr.length - 1，这与指针指向的含义是相悖的，因为左指针指向的元素必须是小于k的，而我们并不能保证arr[0]一定小于k，同样的我们也不能保证arr[arr.length - 1]一定大于等于k。

还有一点就是如果数组中没有k这个算法是否依然会返回一个正确的值（0），这也是需要验证的。

public int GetNumberOfK(int [] arr , int k) {  
 if(arr == null || arr.length == 0){  
 return 0;  
 }  
 if(arr.length == 1){  
 return (arr[0] == k) ? 1 : 0;  
 }  
  
 int start, end, left, right;  
 for(start = -1, end = arr.length ; end > start && end - start != 1 ;){  
 int mid = start + ((end - start) >> 1);  
 if(arr[mid] >= k){  
 end = mid;  
 }else{  
 start = mid;  
 }  
 }  
 left = end;  
 for(start = -1, end = arr.length; end > start && end - start != 1 ;){  
 int mid = start + ((end - start) >> 1);  
 if(arr[mid] > k){  
 end = mid;  
 }else{  
 start = mid;  
 }  
 }  
 right = start;  
 return right - left + 1;  
}

### 二叉树的深度

#### 题目描述

输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

public int TreeDepth(TreeNode root) {  
}

#### 解析

1. 将TreeDepth看做一个黑盒，假设利用这个黑盒收集到了左子树和右子树的深度，那么当前这棵树的深度就是前面两者的最大值加1
2. base case，如果当前是一棵空树，那么深度为0

public class Solution {  
 public int TreeDepth(TreeNode root) {  
 if(root == null){  
 return 0;  
 }  
 return Math.max(TreeDepth(root.left), TreeDepth(root.right)) + 1;  
 }  
}

### 平衡二叉树

#### 题目描述

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

public boolean IsBalanced\_Solution(TreeNode root) {  
  
}

#### 解析

判断当前这棵树是否是平衡二叉所需要收集的信息：

1. 左子树、右子树各自是平衡二叉树吗（需要收集子树是否是平衡二叉树）
2. 如果1成立，还需要收集左子树和右子树的高度，如果高度相差不超过1那么当前这棵树才是平衡二叉树（需要收集子树的高度）

class Info{  
 boolean isBalanced;  
 int height;  
 Info(boolean isBalanced, int height){  
 this.isBalanced = isBalanced;  
 this.height = height;  
 }  
}

递归体的定义：（这里高度之差不超过1中的left.height - right.height == 0容易被忽略）

public boolean IsBalanced\_Solution(TreeNode root) {  
 return process(root).isBalanced;  
}  
  
public Info process(TreeNode root){  
 if(root == null){  
 return new Info(true, 0);  
 }  
 Info left = process(root.left);  
 Info right = process(root.right);  
 if(!left.isBalanced || !right.isBalanced){  
 //如果左子树或右子树不是平衡二叉树，那么当前这棵树肯定也不是，树高度信息也就没用了  
 return new Info(false, 0);  
 }  
 //高度之差不超过1  
 if(left.height - right.height == 1 || left.height - right.height == -1 ||  
 left.height - right.height == 0){  
 return new Info(true, Math.max(left.height, right.height) + 1);  
 }  
 return new Info(false, 0);  
}

### 数组中只出现一次的数字

#### 题目描述

一个整型数组里除了两个数字之外，其他的数字都出现了偶数次。请写程序找出这两个只出现一次的数字。

#### 解析

如果没有解过类似的题目，思路比较难打开。面试官可能会提醒你，如果是让你求一个整型数组里只有一个数只出现了一次而其它数出现了偶数次呢？你应该联想到：

1. **偶数次相同的数异或的结果是0**
2. **任何数与0异或的结果是它本身**

于是将数组从头到尾求异或和便可得知结果。那么对于此题，能否将数组分成这样的两部分呢：每个部分只有一个数出现了一次，其他的数都出现偶数次。

如果我们仍将整个数组从头到尾求异或和，那结果应该和这两个只出现一次的数的异或结果相同，目前我们所能依仗的也就是这个结果了，能否靠这个结果将数组分成想要的两部分？

由于两个只出现一次的数（用A和B表示）异或结果A ^ B肯定不为0，那么A ^ B的二进制表示中肯定包含数值为1的bit位，而这个位上的1肯定是由A或B提供的，也就是说我们能根据**这个bit位上的数是否为1**来区分A和B，那剩下的数呢？

由于剩下的数都出现偶数次，因此相同的数都会被分到一边（按照某个bit位上是否为1来分）。

public void FindNumsAppearOnce(int [] arr,int num1[] , int num2[]) {  
 if(arr == null || arr.length <= 1){  
 return;  
 }  
 int xorSum = 0;  
 for(int num : arr){  
 xorSum ^= num;  
 }  
 //取xorSum二进制表示中低位为1的bit位，将其它的bit位 置0  
 //比如：xorSum = 1100，那么 (1100 ^ 1011) & 1100 = 0100，只剩下一个为1的bit位  
 xorSum = (xorSum ^ (xorSum - 1)) & xorSum;  
  
 for(int num : arr){  
 num1[0] = (num & xorSum) == 0 ? num1[0] ^ num : num1[0];  
 num2[0] = (num & xorSum) != 0 ? num2[0] ^ num : num2[0];  
 }  
}

### 和为S的连续正数序列

#### 题目描述

小明很喜欢数学,有一天他在做数学作业时,要求计算出9~16的和,他马上就写出了正确答案是100。但是他并不满足于此,他在想究竟有多少种连续的正数序列的和为100(至少包括两个数)。没多久,他就得到另一组连续正数和为100的序列:18,19,20,21,22。现在把问题交给你,你能不能也很快的找出所有和为S的连续正数序列? Good Luck!

public ArrayList<ArrayList<Integer> > FindContinuousSequence(int sum) {  
  
}

#### 输出描述

输出所有和为S的连续正数序列。序列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序

#### 解析

将1 ~ (S / 2 + 1)区间的数n依次加入到队列中（因为从S/2 + 1之后的任意两个正数之和都大于S）：

1. 将n加入到队列queue中并将队列元素之和queueSum更新，更新queueSum之后如果发现等于sum，那么将此时的队列快照加入到返回结果res中，并弹出队首元素（**保证下次入队操作时队列元素之和是小于sum的**）
2. 更新queueSum之后如果发现大于sum，那么循环弹出队首元素直到queueSum <= Sum，如果循环弹出之后发现queueSum == sum那么将队列快照加入到res中，并弹出队首元素（**保证下次入队操作时队列元素之和是小于sum的**）；如果queueSum < sum那么入队下一个n

于是有如下代码：

public ArrayList<ArrayList<Integer>> FindContinuousSequence(int sum) {  
 ArrayList<ArrayList<Integer>> res = new ArrayList();  
 if(sum <= 1){  
 return res;  
 }  
 LinkedList<Integer> queue = new LinkedList();  
 int n = 1, halfSum = (sum >> 1) + 1, queueSum = 0;  
 while(n <= halfSum){  
 queue.addLast(n);  
 queueSum += n;  
 if(queueSum == sum){  
 ArrayList<Integer> one = new ArrayList();  
 one.addAll(queue);  
 res.add(one);  
 queueSum -= queue.pollFirst();  
 }else if(queueSum > sum){  
 while(queueSum > sum){  
 queueSum -= queue.pollFirst();  
 }  
 if(queueSum == sum){  
 ArrayList<Integer> one = new ArrayList();  
 one.addAll(queue);  
 res.add(one);  
 queueSum -= queue.pollFirst();  
 }  
 }  
 n++;  
 }  
  
 return res;  
}

我们发现11~15和20~24行的代码是重复的，于是可以稍微优化一下：

public ArrayList<ArrayList<Integer>> FindContinuousSequence(int sum) {  
 ArrayList<ArrayList<Integer>> res = new ArrayList();  
 if(sum <= 1){  
 return res;  
 }  
 LinkedList<Integer> queue = new LinkedList();  
 int n = 1, halfSum = (sum >> 1) + 1, queueSum = 0;  
 while(n <= halfSum){  
 queue.addLast(n);  
 queueSum += n;  
 if(queueSum > sum){  
 while(queueSum > sum){  
 queueSum -= queue.pollFirst();  
 }  
 }  
 if(queueSum == sum){  
 ArrayList<Integer> one = new ArrayList();  
 one.addAll(queue);  
 res.add(one);  
 queueSum -= queue.pollFirst();  
 }  
 n++;  
 }  
  
 return res;  
}

### 和为S的两个数字

#### 题目描述

输入一个递增排序的数组和一个数字S，在数组中查找两个数，使得他们的和正好是S，如果有多对数字的和等于S，输出两个数的乘积最小的。

public ArrayList<Integer> FindNumbersWithSum(int [] arr,int sum) {  
   
}

#### 输出描述

对应每个测试案例，输出查找到的两个数，如果有多对，输出乘积最小的两个。

#### 解析

使用指针l,r，初始时l指向首元素，r指向尾元素，当两指针元素之和不等于sum且r指针在l指针右侧时循环：

1. 如果两指针元素之和大于sum，那么将r指针左移，试图减小两指针之和
2. 如果两指针元素之和小于sum，那么将l右移，试图增大两指针之和
3. 如果两指针元素之和等于sum那么就可以返回了，或者r跑到了l的左边表名没有和sum的两个数，也可以返回了。

public ArrayList<Integer> FindNumbersWithSum(int [] arr,int sum) {  
 ArrayList<Integer> res = new ArrayList<Integer>();  
 if(arr == null || arr.length <= 1 ){  
 return res;  
 }  
 int l = 0, r = arr.length - 1;  
 while(arr[l] + arr[r] != sum && r > l){  
 if(arr[l] + arr[r] > sum){  
 r--;  
 }else{  
 l++;  
 }  
 }  
 if(arr[l] + arr[r] == sum){  
 res.add(arr[l]);  
 res.add(arr[r]);  
 }  
 return res;  
}

### 旋转字符串

#### 题目描述

汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。对于一个给定的字符序列S，请你把其循环左移K位后的序列输出。例如，字符序列S=”abcXYZdef”,要求输出循环左移3位后的结果，即“XYZdefabc”。是不是很简单？OK，搞定它！

public String LeftRotateString(String str,int n) {  
   
}

#### 解析

将开头的一段子串移到串尾：将开头的子串翻转一下、将剩余的子串翻转一下，最后将整个子串翻转一下。按理来说应该输入char[] str的，这样的话这种算法不会使用额外空间。

public String LeftRotateString(String str,int n) {  
 if(str == null || str.length() == 0 || n <= 0){  
 return str;  
 }  
 char[] arr = str.toCharArray();  
 reverse(arr, 0, n - 1);  
 reverse(arr, n, arr.length - 1);  
 reverse(arr, 0, arr.length - 1);  
 return new String(arr);  
}  
  
public void reverse(char[] str, int start, int end){  
 if(str == null || str.length == 0 || start < 0 || end > str.length - 1 || start >= end){  
 return;  
 }  
 for(int i = start, j = end ; j > i ; i++, j--){  
 char tmp = str[i];  
 str[i] = str[j];  
 str[j] = tmp;  
 }  
}

### 翻转单词顺序列

#### 题目描述

牛客最近来了一个新员工Fish，每天早晨总是会拿着一本英文杂志，写些句子在本子上。同事Cat对Fish写的内容颇感兴趣，有一天他向Fish借来翻看，但却读不懂它的意思。例如，“student. a am I”。后来才意识到，这家伙原来把句子单词的顺序翻转了，正确的句子应该是“I am a student.”。Cat对一一的翻转这些单词顺序可不在行，你能帮助他么？

public String LeftRotateString(String str,int n) {  
   
}

#### 解析

先将整个字符串翻转，最后按照标点符号或空格一次将句中的单词翻转。注意：由于最后一个单词后面没有空格，因此需要单独处理！！！

public String ReverseSentence(String str) {  
 if(str == null || str.length() <= 1){  
 return str;  
 }  
 char[] arr = str.toCharArray();  
 reverse(arr, 0, arr.length - 1);  
 int start = -1;  
 for(int i = 0 ; i < arr.length ; i++){  
 if(arr[i] != ' '){  
 //初始化start  
 start = (start == -1) ? i : start;  
 }else{  
 //如果是空格，不用担心start>i-1，reverse会忽略它  
 reverse(arr, start, i - 1);  
 start = i + 1;  
 }  
 }  
 //最后一个单词，这里比较容易忽略！！！  
 reverse(arr, start, arr.length - 1);  
  
 return new String(arr);  
}  
  
public void reverse(char[] str, int start, int end){  
 if(str == null || str.length == 0 || start < 0 || end > str.length - 1 || start >= end){  
 return ;  
 }  
 for(int i = start, j = end ; j > i ; i++, j--){  
 char tmp = str[i];  
 str[i] = str[j];  
 str[j] = tmp;  
 }  
}

### 扑克牌顺子

#### 题目描述

LL今天心情特别好,因为他去买了一副扑克牌,发现里面居然有2个大王,2个小王(一副牌原本是54张^\_^)...他随机从中抽出了5张牌,想测测自己的手气,看看能不能抽到顺子,如果抽到的话,他决定去买体育彩票,嘿嘿！！“红心A,黑桃3,小王,大王,方片5”,“Oh My God!”不是顺子.....LL不高兴了,他想了想,决定大\小 王可以看成任何数字,并且A看作1,J为11,Q为12,K为13。上面的5张牌就可以变成“1,2,3,4,5”(大小王分别看作2和4),“So Lucky!”。LL决定去买体育彩票啦。 现在,要求你使用这幅牌模拟上面的过程,然后告诉我们LL的运气如何， 如果牌能组成顺子就输出true，否则就输出false。为了方便起见,你可以认为大小王是0。

#### 解析

先将数组排序（5个元素排序时间复杂O(1）），然后遍历数组统计王的数量和相邻非王牌之间的缺口数（需要用几个王来填）。还有一点值得注意：如果发现两种相同的非王牌，则不可能组成五张不同的顺子。

public boolean isContinuous(int [] arr) {  
 if(arr == null || arr.length != 5){  
 return false;  
 }  
 //5 numbers -> O(1)  
 Arrays.sort(arr);  
 int zeroCount = 0, slots = 0;  
 for(int i = 0 ; i < arr.length ; i++){  
 //如果遇到两张相同的非王牌则不可能组成顺子，这点很容易忽略！！！  
 if(i > 0 && arr[i - 1] != 0){  
 if(arr[i] == arr[i - 1]){  
 return false;  
 }else{  
 slots += arr[i] - arr[i - 1] - 1;  
 }  
  
 }  
 zeroCount = (arr[i] == 0) ? ++zeroCount : zeroCount;  
 }  
  
 return zeroCount >= slots;  
}

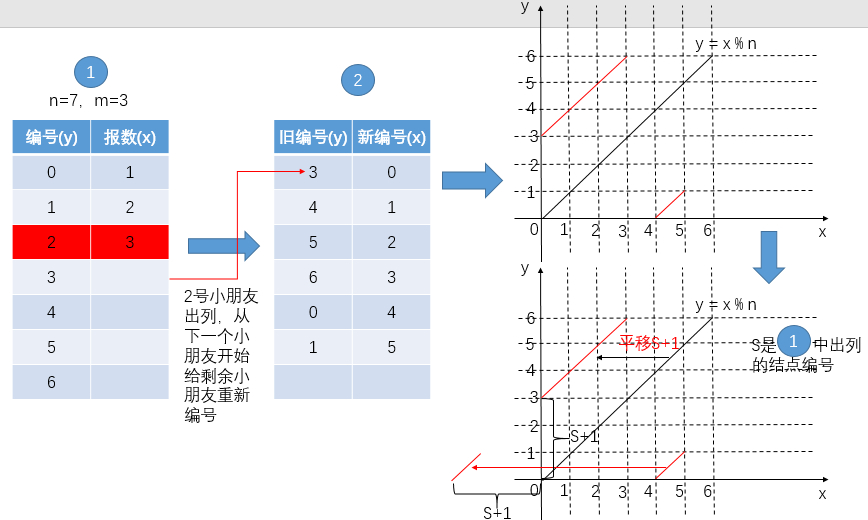
### 孩子们的游戏（圆圈中剩下的数）

#### 题目描述

每年六一儿童节,牛客都会准备一些小礼物去看望孤儿院的小朋友,今年亦是如此。HF作为牛客的资深元老,自然也准备了一些小游戏。其中,有个游戏是这样的:首先,让小朋友们围成一个大圈。然后,他随机指定一个数m,让编号为0的小朋友开始报数。每次喊到m-1的那个小朋友要出列唱首歌,然后可以在礼品箱中任意的挑选礼物,并且不再回到圈中,从他的下一个小朋友开始,继续0...m-1报数....这样下去....直到剩下最后一个小朋友,可以不用表演,并且拿到牛客名贵的“名侦探柯南”典藏版(名额有限哦!!^\_^)。请你试着想下,哪个小朋友会得到这份礼品呢？(注：小朋友的编号是从0到n-1)

#### 解析

1. 报数时，在报到m-1之前，可通过报数求得报数的结点编号：
2. 在某个结点（小朋友）出列后的重新编号过程中，可通过新编号求结点的就编号

* 
* 因此在某轮重新编号时，我们能在已知新编号x的情况下通过公式y = (x + S + 1) % n求得结点重新标号之前的旧编号，上述两步分析的公式整理如下：
  1. 某一轮报数出列前：编号 = （报数 - 1）% 出列前结点个数
  2. 某一轮报数出列后：旧编号 = （新编号 + 出列编号 + 1）% 出列前结点个数，因为出列结点是因为报数m才出列的，所以有：出列编号 = （m - 1）% 出列前结点个数
  3. 由2可推出：旧编号 = （新编号 + （m - 1）% 出列前结点个数 + 1）% 出列前结点个数 ，若用n表示**出列后**结点个数：y = (x + (m - 1) % n + 1) % n = (x + m - 1) % n + 1

经过上面3步的复杂分析之后，我们得出这么一个通式：旧编号 = （新编号 + m - 1 ）% 出列前结点个数 + 1，于是我们就可以自下而上（用链表模拟出列过程是自上而下），求出**最后一轮重新编号为1**的小朋友（只剩他一个了）在倒数第二轮重新编号时的旧编号，自下而上可倒推出这个小朋友在第一轮编号时（这时还没有任何一个小朋友出列过）的原始编号，即目标答案。

注意：式子y = (x + m - 1) % n + 1的计算结果不可能为0，因此我们可以按小朋友从1开始编号，将最后的计算结果应题目的要求（小朋友从0开始编号）减一个1即可。

public int LastRemaining\_Solution(int n, int m) {  
 if(n <= 0){  
 //throw new IllegalArgumentException();  
 return -1;  
 }  
 //最后一次重新编号：最后一个结点编号为1，出列前结点数为2  
 return orginalNumber(2, 0, n, m);  
}  
  
//根据出列后的重新编号（newNumber）推导出列前的旧编号（返回值）  
//n：出列前有多少小朋友，N：总共有多少个小朋友  
public int orginalNumber(int n, int newNumber, int N, int m){  
 int lastNumber = (newNumber + m - 1) % n + 1;  
 if(n == N){  
 return lastNumber;  
 }  
 return orginalNumber(n + 1, lastNumber, N, m);  
}

### 求1+2+3+…+n

#### 题目描述

求1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

public int Sum\_Solution(int n) {  
   
}

#### 解析

##### 递归轻松解决

既然不允许遍历求和，不如将计算分解，如果知道了f(n - 1)，f(n)则可以通过f(n - 1) + n算出：

public int Sum\_Solution(int n) {  
 if(n == 1){  
 return 1;  
 }  
 return n + Sum\_Solution(n - 1);  
}

### 不用加减乘除做加法

#### 题目描述

写一个函数，求两个整数之和，要求在函数体内不得使用+、-、\*、/四则运算符号。

#### 解析

不要忘了加减乘除是人类熟悉的运算方法，而计算机只知道位运算哦！

我们可以将两数的二进制表示写出来，然后按位与得出进位信息、按位或得出非进位信息，如果进位信息不为0，则循环计算直到进位信息为0，此时异或信息就是两数之和：

public int Add(int num1,int num2) {  
 if(num1 == 0 || num2 == 0){  
 return num1 == 0 ? num2 : num1;  
 }  
 int and = 0, xor = 0;  
 do{  
 and = num1 & num2;  
 xor = num1 ^ num2;  
 num1 = and << 1;  
 num2 = xor;  
 }while(and != 0);  
  
 return xor;  
}

### 把字符串转换成整数

#### 题目描述

将一个字符串转换成一个整数(实现Integer.valueOf(string)的功能，但是string不符合数字要求时返回0)，要求不能使用字符串转换整数的库函数。 数值为0或者字符串不是一个合法的数值则返回0。

public int StrToInt(String str) {  
   
}

#### 输入描述

输入一个字符串,包括数字字母符号,可以为空

#### 输出描述

如果是合法的数值表达则返回该数字，否则返回0

#### 示例

输入：+2147483647，输出：2147483647  
输入：1a33，输出0

#### 解析

1. 只有第一个位置上的字符可以是+或-或数字，其他位置上的字符必须是数字
2. 如果第一个字符是-，返回结果必须是负数
3. 如果字符串只有一个字符，且为+或-，这情况很容易被忽略
4. 在对字符串解析转换时，如果发现溢出（包括正数向负数溢出，负数向正数溢出），必须有所处理（此时可以和面试官交涉），但不能视而不见

public int StrToInt(String str) {  
 if(str == null || str.length() == 0){  
 return 0;  
 }  
 boolean minus = false;  
 int index = 0;  
 if(str.charAt(0) == '-'){  
 minus = true;  
 index = 1;  
 }else if(str.charAt(0) == '+'){  
 index = 1;  
 }  
 //如果只有一个正负号  
 if(index == str.length()){  
 return 0;  
 }  
  
 if(checkInteger(str, index, str.length() - 1)){  
 return transform(str, index, str.length() - 1, minus);  
 }  
  
 return 0;  
}  
  
public boolean checkInteger(String str, int start, int end){  
 if(str == null || str.length() == 0 || start < 0 || end > str.length() - 1 || start > end){  
 return false;  
 }  
 for(int i = start ; i <= end ; i++){  
 if(str.charAt(i) < '0' || str.charAt(i) > '9'){  
 return false;  
 }  
 }  
 return true;  
}  
  
public int transform(String str, int start, int end, boolean minus){  
 if(str == null || str.length() == 0 || start < 0 || end > str.length() - 1 || start > end){  
 throw new IllegalArgumentException();  
 }  
 int res = 0;  
 for(int i = start ; i <= end ; i++){  
 int num = str.charAt(i) - '0';  
 res = minus ? (res \* 10 - num) : (res \* 10 + num);  
 if((minus && res > 0) || (!minus && res < 0)){  
 throw new ArithmeticException("the str is overflow int");  
 }  
 }  
 return res;  
}

### 数组中重复的数字

#### 题目描述

在一个长度为n的数组里的所有数字都在0到n-1的范围内。 数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中任意一个重复的数字。 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是第一个重复的数字2。

// Parameters:  
// numbers: an array of integers  
// length: the length of array numbers  
// duplication: (Output) the duplicated number in the array number,length of duplication array is 1,so using duplication[0] = ? in implementation;  
// Here duplication like pointor in C/C++, duplication[0] equal \*duplication in C/C++  
// 这里要特别注意~返回任意重复的一个，赋值duplication[0]  
// Return value: true if the input is valid, and there are some duplications in the array number  
// otherwise false  
public boolean duplicate(int numbers[],int length,int [] duplication) {  
   
}

#### 解析

认真审题发现输入数据是有特征的，即数组长度为n，数组中的元素都在0~n-1范围内，如果数组中没有重复的元素，那么排序后每个元素和其索引值相同，这就意味着数组中如果有重复的元素，那么数组排序后肯定有元素和它对应的索引是不等的。

顺着这个思路，我们可以将每个元素放到与它相等的索引上，如果某次放之前发现对应的索引上已有了和索引相同的元素，那么说明这个元素是重复的，由于每个元素最多会被调整两次，因此时间复杂O(n)

public boolean duplicate(int arr[],int length,int [] duplication) {  
 if(arr == null || arr.length == 0){  
 return false;  
 }  
 int index = 0;  
 while(index < arr.length){  
 if(arr[index] == arr[arr[index]]){  
 if(index != arr[index]){  
 duplication[0] = arr[index];  
 return true;  
 }else{  
 index++;  
 }  
 }else{  
 int tmp = arr[index];  
 arr[index] = arr[tmp];  
 arr[tmp] = tmp;  
 }  
 }  
  
 return false;  
}

### 构建乘积数组

#### 题目描述

给定一个数组A[0,1,...,n-1],请构建一个数组B[0,1,...,n-1],其中B中的元素B[i]=A[0]*A[1]*...*A[i-1]*A[i+1]*...*A[n-1]。不能使用除法。

public int[] multiply(int[] arr) {  
   
}

#### 分析

规律题：

public int[] multiply(int[] arr) {  
 if(arr == null || arr.length == 0){  
 return arr;  
 }  
 int len = arr.length;  
 int[] arr1 = new int[len], arr2 = new int[len];  
 arr1[0] = 1;  
 arr2[len - 1] = 1;  
 for(int i = 1 ; i < len ; i++){  
 arr1[i] = arr1[i - 1] \* arr[i - 1];  
 arr2[len - 1 - i] = arr2[len - i] \* arr[len - i];  
 }  
 int[] res = new int[len];  
 for(int i = 0 ; i < len ; i++){  
 res[i] = arr1[i] \* arr2[i];  
 }  
  
 return res;  
}

### 正则表达式匹配

#### 题目描述

请实现一个函数用来匹配包括'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符，而'*'表示它前面的字符可以出现任意次（包含0次）。 在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但是与"aa.a"和"ab\*a"均不匹配

public boolean match(char[] str, char[] pattern){  
   
}

#### 解析

使用p1指向str中下一个要匹配的字符，使用p2指向pattern中剩下的模式串的首字符

1. 如果p2 >= pattern.length，表示模式串消耗完了，这时如果p1仍有字符要匹配那么返回false否则返回true
2. 如果p1 >= str.length，表示要匹配的字符都匹配完了，但模式串还没消耗完，这时剩下的模式串必须符合a\*b\*c\*这样的范式以能够作为空串处理，否则返回false
3. p1和p2都未越界，按照p2后面是否是\*来讨论
   1. p2后面如果是\*，又可按照pattern[p2]是否能够匹配str[p1]分析：
      1. pattern[p2] == ‘.’ || pattern[p2] == str[p1]，这时可以选择匹配一个str[p1]并继续向后匹配（不用跳过p2和其后面的\*），也可以选择将pattern[p2]和其后面的\*作为匹配空串处理，这时要跳过p2和 其后面的\*
      2. pattern[p2] != str[p1]，只能作为匹配空串处理，跳过p2
   2. p2后面如果不是\*：
      1. pattern[p2] == str[p1] || pattern[p2] == ‘.’，p1,p2同时后移一个继续匹配
      2. pattern[p2] == str[p1]，直接返回false

public boolean match(char[] str, char[] pattern){  
 if(str == null || pattern == null){  
 return false;  
 }  
 if(str.length == 0 && pattern.length == 0){  
 return true;  
 }  
 return matchCore(str, 0, pattern, 0);  
}  
  
public boolean matchCore(char[] str, int p1, char[] pattern, int p2){  
 //模式串用完了  
 if(p2 >= pattern.length){  
 return p1 >= str.length;  
 }  
 if(p1 >= str.length){  
 if(p2 + 1 < pattern.length && pattern[p2 + 1] == '\*'){  
 return matchCore(str, p1, pattern, p2 + 2);  
 }else{  
 return false;  
 }  
 }  
  
 //如果p2的后面是“\*”  
 if(p2 + 1 < pattern.length && pattern[p2 + 1] == '\*'){  
 if(pattern[p2] == '.' || pattern[p2] == str[p1]){  
 //匹配一个字符，接着还可以向后匹配；或者将当前字符和后面的星合起来做空串  
 return matchCore(str, p1 + 1, pattern, p2) || matchCore(str, p1, pattern, p2 + 2);  
 }else{  
 return matchCore(str, p1, pattern, p2 + 2);  
 }  
 }  
 //如果p2的后面不是\*  
 else{  
 if(pattern[p2] == '.' || pattern[p2] == str[p1]){  
 return matchCore(str, p1 + 1, pattern, p2 + 1);  
 }else{  
 return false;  
 }  
 }  
}

### 表示数值的字符串

#### 题目描述

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100","5e2","-123","3.1416"和"-1E-16"都表示数值。 但是"12e","1a3.14","1.2.3","+-5"和"12e+4.3"都不是。

public boolean isNumeric(char[] str) {  
  
}

#### 解析

由题式可得出如下约束：

1. 正负号只能出现在第一个位置或者e/E后一个位置
2. e/E后面有且必须有整数
3. 字符串中只能包含数字、小数点、正负号、e/E，其它的都是非法字符
4. e/E的前面最多只能出现一次小数点，而e/E的后面不能出现小数点

public boolean isNumeric(char[] str) {  
 if(str == null || str.length == 0){  
 return false;  
 }  
  
 boolean signed = false; //标识是否以正负号开头  
 boolean decimal = false; //标识是否有小数点  
 boolean existE = false; //是否含有e/E  
 int start = -1; //一段连续数字的开头  
 int index = 0; //从0开始遍历字符  
  
 if(existSignAtIndex(str, 0)){  
 signed = true;  
 index++;  
 }  
  
 while(index < str.length){  
 //以下按照index上可能出现的字符进行分支判断  
 if(str[index] >= '0' && str[index] <= '9'){  
 start = (start == -1) ? index : start;  
 index++;  
  
 }else if(str[index] == '+' || str[index] == '-'){  
 //首字符的+-我们已经判断过了，因此+-只可能出现在e/E的后面  
 if(!existEAtIndex(str, index - 1)){  
 return false;  
 }  
 index++;  
  
 }else if(str[index] == '.'){  
 //小数点只可能出现在e/E前面，且只可能出现一次  
 //如果出现过小数点了，或者小数点前一段连续数字的前面是e/E  
 if(decimal || existEAtIndex(str, start - 1)  
 || existEAtIndex(str, start - 2) ){  
 return false;  
 }  
 decimal = true;//出现了小数点  
 index++;  
 //下一段连续数字的开始  
 start = index;  
  
 }else if(existEAtIndex(str, index)){  
 if(existE){  
 //如果已出现过e/E  
 return false;  
 }  
 existE = true;  
 index++;  
 //由于e/E后面可能是正负号也可能是数字，所以下一段连续数字的开始不确定  
 start = !existSignAtIndex(str, index) ? index : index + 1;  
  
 }else{  
 return false;  
 }  
 }  
  
 //如果最后一段连续数字的开始不存在 -> e/E后面没有数字  
 if(start >= str.length){  
 return false;  
 }  
  
 return true;  
}  
  
//在index上的字符是否是e或者E  
public boolean existEAtIndex(char[] str, int index){  
 if(str == null || str.length == 0 || index < 0 || index > str.length - 1){  
 return false;  
 }  
 return str[index] == 'e' || str[index] == 'E';  
}  
  
//在index上的字符是否是正负号  
public boolean existSignAtIndex(char[] str, int index){  
 if(str == null || str.length == 0 || index < 0 || index > str.length - 1){  
 return false;  
 }  
 return str[index] == '+' || str[index] == '-';  
}

### 字符流中第一个只出现一次的字符

#### 题目描述

请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符"go"时，第一个只出现一次的字符是"g"。当从该字符流中读出前六个字符“google"时，第一个只出现一次的字符是"l"。

#### 输出描述

如果当前字符流没有存在出现一次的字符，返回#字符。

#### 解析

首先要选取一个容器来保存字符，并且要记录字符进入容器的顺序。如果不考虑中文字符，那么可以使用一张大小为256（对应ASCII码值范围）的表来保存字符，用字符的ASCII**码值**作为索引，用字符进入容器的**次序**作为索引对应的记录，表内部维护了一个计数器position，每当有字符进入时以该计数器的值作为该字符的次序（初始时，每个字符对应的次序为-1），如果设置该字符的次序时发现之前已设置过（次序不为-1，而是大于等于0），那么将该字符的次序置为-2，表示以后从容器取第一个只出现一次的字符时不考虑该字符。

当从容器取第一个只出现一次的字符时，考虑次序大于等于0的字符，在这个前提下找出次序最小的字符并返回。

//不算中文，保存所有ascii码对应的字符只需256字节，记录ascii码为index的字符首次出现的位置  
int[] arr = new int[256];  
int position = 0;  
{  
 for(int i = 0 ; i < arr.length ; i++){  
 //初始时所有字符的首次出现的位置为-1  
 arr[i] = -1;  
 }  
}  
//Insert one char from stringstream  
public void Insert(char ch){  
 int ascii = (int)ch;  
 if(arr[ascii] == -1){  
 arr[ascii] = position++;  
 }else if(arr[ascii] >= 0){  
 arr[ascii] = -2;  
 }  
}  
//return the first appearence once char in current stringstream  
public char FirstAppearingOnce(){  
 int minPosi = Integer.MAX\_VALUE;  
 char res = '#';  
 for(int i = 0 ; i < arr.length ; i++){  
 if(arr[i] >= 0 && arr[i] < minPosi){  
 minPosi = arr[i];  
 res = (char)i;  
 }  
 }  
  
 return res;  
}

### 删除链表中重复的结点

#### 题目描述

在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。 例如，链表1->2->3->3->4->4->5 处理后为 1->2->5

public ListNode deleteDuplication(ListNode pHead){  
   
}

#### 解析

此题处理起来棘手的有两个地方：

1. 如果某个结点的后继结点与其重复，那么删除该结点的一串连续重复的结点之后如何删除该结点本身，这就要求我们需要保留当前遍历结点的前驱指针。

* 但是如果从头结点开始就出现一连串的重复呢？我们又如何删除删除头结点，因此我们需要新建一个辅助结点作为头结点的前驱结点。

1. 在遍历过程中如何区分当前结点是不重复的结点，还是在删除了它的若干后继结点之后最后也要删除它本身的重复结点？这就需要我们使用一个布尔变量记录是否开启了删除模式（deleteMode）

经过上述两步分析，我们终于可以安心遍历结点了：

public ListNode deleteDuplication(ListNode pHead){  
 if(pHead == null){  
 return null;  
 }  
 ListNode node = new ListNode(Integer.MIN\_VALUE);  
 node.next = pHead;  
 ListNode pre = node, p = pHead;  
 boolean deletedMode = false;  
 while(p != null){  
 if(p.next != null && p.next.val == p.val){  
 p.next = p.next.next;  
 deletedMode = true;  
 }else if(deletedMode){  
 pre.next = p.next;  
 p = pre.next;  
 deletedMode = false;  
 }else{  
 pre = p;  
 p = p.next;  
 }  
 }  
  
 return node.next;  
}

### 二叉树的下一个结点

#### 题目描述

给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。

#### 解析

由于中序遍历来到某个结点后，首先会接着遍历它的右子树，如果它没有右子树则会回到祖先结点中将它当做左子树上的结点的那一个，因此有如下分析：

1. 如果当前结点有右子树，那么其后继结点就是其右子树上的最左结点
2. 如果当前结点没有右子树，那么其后继结点就是其祖先结点中，将它当做左子树上的结点的那一个。

public TreeLinkNode GetNext(TreeLinkNode pNode){  
 if(pNode == null){  
 return null;  
 }  
 //如果有右子树，后继结点是右子树上最左的结点  
 if(pNode.right != null){  
 TreeLinkNode p = pNode.right;  
 while(p.left != null){  
 p = p.left;  
 }  
 return p;  
 }else{  
 //如果没有右子树，向上查找第一个当前结点是父结点的左孩子的结点  
 TreeLinkNode p = pNode.next;  
 while(p != null && pNode != p.left){  
 pNode = p;  
 p = p.next;  
 }  
  
 if(p != null && pNode == p.left){  
 return p;  
 }  
 return null;  
 }  
}

### 对称的二叉树

#### 题目描述

请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

boolean isSymmetrical(TreeNode pRoot){  
   
｝

#### 解析

判断一棵树是否是镜像二叉树，只需将经典的先序遍历序列和变种的**先根再右再左**的先序遍历序列比较，如果相同则为镜像二叉树。

boolean isSymmetrical(TreeNode pRoot){  
 if(pRoot == null){  
 return true;  
 }  
 StringBuffer str1 = new StringBuffer("");  
 StringBuffer str2 = new StringBuffer("");  
 preOrder(pRoot, str1);  
 preOrder2(pRoot, str2);  
 return str1.toString().equals(str2.toString());  
}  
  
public void preOrder(TreeNode root, StringBuffer str){  
 if(root == null){  
 str.append("#");  
 return;  
 }  
 str.append(String.valueOf(root.val));  
 preOrder(root.left, str);  
 preOrder(root.right, str);  
}  
  
public void preOrder2(TreeNode root, StringBuffer str){  
 if(root == null){  
 str.append("#");  
 return;  
 }  
 str.append(String.valueOf(root.val));  
 preOrder2(root.right, str);  
 preOrder2(root.left, str);  
}

### 按之字形打印二叉树

#### 题目描述

请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。

#### 解析

注意下述代码的第14行，笔者曾写为stack2 = stack1 == empty ? stack1 : stack2，你能发现错误在哪儿吗？

public ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {  
 ArrayList<ArrayList<Integer>> res = new ArrayList();  
 if(pRoot == null){  
 return res;  
 }  
  
 Stack<TreeNode> stack1 = new Stack();  
 Stack<TreeNode> stack2 = new Stack();  
 stack1.push(pRoot);  
 boolean flag = true;//先加左孩子，再加右孩子  
 while(!stack1.empty() || !stack2.empty()){  
 Stack<TreeNode> empty = stack1.empty() ? stack1 : stack2;  
 stack1 = stack1 == empty ? stack2 : stack1;  
 stack2 = empty;  
 ArrayList<Integer> row = new ArrayList();  
 while(!stack1.empty()){  
 TreeNode p = stack1.pop();  
 row.add(p.val);  
 if(flag){  
 if(p.left != null){  
 stack2.push(p.left);  
 }  
 if(p.right != null){  
 stack2.push(p.right);  
 }  
 }else{  
 if(p.right != null){  
 stack2.push(p.right);  
 }  
 if(p.left != null){  
 stack2.push(p.left);  
 }  
 }  
 }  
 res.add(row);  
 flag = !flag;  
 }  
  
 return res;  
}

### 序列化二叉树

#### 题目描述

请实现两个函数，分别用来序列化和反序列化二叉树

#### 解析

怎么序列化的，就怎么反序列化。这里deserialize反序列化时对于序列化到String[] arr的哪个结点值来了的变量index有两个坑（都是笔者亲自踩的）：

1. 将index声明为成员的int，Java中函数调用时不会改变基本类型参数的值的，因此不要企图使用int表示当前序列化哪个结点的值来了
2. 之后笔者想用Integer代替，但是Integer和String一样，都是不可变对象，所有的值更改操作在底层都是拆箱和装箱生成新的Integer，因此也不要使用Integer做序列化到哪一个结点数值来了的计数器
3. 最好使用数组或者自定义的类（在类中声明一个int变量）

String Serialize(TreeNode root) {  
 if(root == null){  
 return "#\_";  
 }  
 //处理头结点、左子树、右子树  
 String res = root.val + "\_";  
 res += Serialize(root.left);  
 res += Serialize(root.right);  
 return res;  
}  
  
TreeNode Deserialize(String str) {  
 if(str == null || str.length() == 0){  
 return null;  
 }  
 Integer index = 0;  
 return deserialize(str.split("\_"), new int[]{0});  
}  
  
//怎么序列化的，就怎么反序列化  
TreeNode deserialize(String[] arr, int[] index){  
 if("#".equals(arr[index[0]])){  
 index[0]++;  
 return null;  
 }  
 //头结点、左子树、右子树  
 TreeNode root = new TreeNode(Integer.parseInt(arr[index[0]]));  
 index[0]++;  
 root.left = deserialize(arr, index);  
 root.right = deserialize(arr, index);  
 return root;  
}

### 二叉搜索树的第k个结点

#### 题目描述

给定一棵二叉搜索树，请找出其中的第k小的结点。例如， （5，3，7，2，4，6，8） 中，按结点数值大小顺序第三小结点的值为4。

TreeNode KthNode(TreeNode pRoot, int k){  
   
}

#### 解析

二叉搜索树的特点是，它的中序序列是有序的，因此我们可以借助中序遍历在递归体中第二次来到当前结点时更新一下计数器，直到遇到第k个结点保存并返回即可。

值得注意的地方是：

1. 由于计数器在递归中传来传去，因此你需要保证每个递归引用的是同一个计数器，这里使用的是一个int[]的第一个元素来保存
2. 我们写中序遍历是不需要返回值的，可以在找到第k小的结点时将其保存在传入的数组中以返回给调用方

TreeNode KthNode(TreeNode pRoot, int k){  
 if(pRoot == null){  
 return null;  
 }  
 TreeNode[] res = new TreeNode[1];  
 inOrder(pRoot, new int[]{ k }, res);  
 return res[0];  
}  
  
public void inOrder(TreeNode root, int[] count, TreeNode[] res){  
 if(root == null){  
 return;  
 }  
 inOrder(root.left, count, res);  
 count[0]--;  
 if(count[0] == 0){  
 res[0] = root;  
 return;  
 }  
 inOrder(root.right, count, res);  
}

如果可以利用我们熟知的算法，比如本题中的中序遍历。管它三七二十一先将熟知方法写出来，然后再按具体的业务需求对其进行改造（包括返回值、参数列表，但一般不会更改遍历算法的返回值）

### 数据流的中位数

#### 题目描述

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。我们使用Insert()方法读取数据流，使用GetMedian()方法获取当前读取数据的中位数。

public void Insert(Integer num) {  
   
}  
  
public Double GetMedian() {  
   
}

#### 解析

由于中位数只与排序后位于数组中间的一个数或两个数相关，而与数组两边的其它数无关，因此我们可以用一个大根堆保存数组左半边的数的最大值，用一个小根堆保存数组右半边的最小值，插入元素O(logn)，取中位数O(1)。

public class Solution {  
  
 //小根堆、大根堆  
 PriorityQueue<Integer> minHeap = new PriorityQueue(new MinRootHeadComparator());  
 PriorityQueue<Integer> maxHeap = new PriorityQueue(new MaxRootHeadComparator());  
 int count = 0;  
  
 class MaxRootHeadComparator implements Comparator<Integer>{  
 //返回值大于0则认为逻辑上i2大于i1（无关对象包装的数值）  
 public int compare(Integer i1, Integer i2){  
 return i2.intValue() - i1.intValue();  
 }  
 }  
  
 class MinRootHeadComparator implements Comparator<Integer>{  
 public int compare(Integer i1, Integer i2){  
 return i1.intValue() - i2.intValue();  
 }  
 }  
  
 public void Insert(Integer num) {  
 count++;//当前这个数是第几个进来的  
 //编号是奇数就放入小根堆（右半边），否则放入大根堆  
 if(count % 2 != 0){  
 //如果要放入右半边的数比左半边的最大值要小则需调整左半边的最大值放入右半边并将当前这个数放入左半边，这样才能保证右半边的数都比左半边的大  
 if(maxHeap.size() > 0 && num < maxHeap.peek()){  
 maxHeap.add(num);  
 num = maxHeap.poll();  
 }  
 minHeap.add(num);  
 }else{  
 if(minHeap.size() > 0 && num > minHeap.peek()){  
 minHeap.add(num);  
 num = minHeap.poll();  
 }  
 maxHeap.add(num);  
 }  
 }  
  
 public Double GetMedian() {  
 if(count == 0){  
 return 0.0;  
 }  
 if(count % 2 != 0){  
 return minHeap.peek().doubleValue();  
 }else{  
 return (minHeap.peek().doubleValue() + maxHeap.peek().doubleValue()) / 2;  
 }  
 }  
}

### 滑动窗口的最大值

#### 题目描述

给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}； 针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下6个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。

public ArrayList<Integer> maxInWindows(int [] num, int size){  
   
}

#### 解析

使用一个单调非增队列，队头保存当前窗口的最大值，后面保存在窗口移动过程中导致队头失效（出窗口）后的从而晋升为窗口最大值的候选值。

public ArrayList<Integer> maxInWindows(int [] num, int size){  
 ArrayList<Integer> res = new ArrayList();  
 if(num == null || num.length == 0 || size <= 0 || size > num.length){  
 return res;  
 }  
  
 //用队头元素保存窗口最大值，队列中元素只能是单调递减的，窗口移动可能导致队头元素失效  
 LinkedList<Integer> queue = new LinkedList();  
 int start = 0, end = size - 1;  
 for(int i = start ; i <= end ; i++){  
 addLast(queue, num[i]);  
 }  
 res.add(queue.getFirst());  
 //移动窗口  
 while(end < num.length - 1){  
 addLast(queue, num[++end]);  
 if(queue.getFirst() == num[start]){  
 queue.pollFirst();  
 }  
 start++;  
 res.add(queue.getFirst());  
 }  
  
 return res;  
}  
  
public void addLast(LinkedList<Integer> queue, int num){  
 if(queue == null){  
 return;  
 }  
 //加元素之前要确保该元素小于等于队尾元素  
 while(queue.size() != 0 && num > queue.getLast()){  
 queue.pollLast();  
 }  
 queue.addLast(num);  
}

### 矩形中的路径

#### 题目描述

请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则之后不能再次进入这个格子。 例如 a b c e s f c s a d e e 这样的3 X 4 矩阵中包含一条字符串"bcced"的路径，但是矩阵中不包含"abcb"路径，因为字符串的第一个字符b占据了矩阵中的第一行第二个格子之后，路径不能再次进入该格子。

#### 解析

定义一个黑盒hasPathCorechar(matrix, rows, cols, int i, int j, str, index)，表示从rows行cols列的矩阵matrix中的(i,j)位置开始走是否能走出一条与str的子串index ~ str.length-1相同的路径。那么对于当前位置(i,j)，需要关心的只有一下三点：

1. (i,j)是否越界了
2. (i,j)上的字符是否和str[index]匹配
3. (i,j)是否已在之前走过的路径上

如果通过了上面三点检查，那么认为(i,j)这个位置是可以走的，剩下的就是(i,j)上下左右四个方向能否走出str的index+1 ~ str.length-1，这个交给黑盒就好了。

还有一点要注意，如果确定了可以走当前位置(i,j)，那么需要将该位置的visited标记为true，表示该位置在已走过的路径上，而退出(i,j)的时候（对应下面第32行）又要将他的visited重置为false。

public boolean hasPath(char[] matrix, int rows, int cols, char[] str){  
 if(matrix == null || matrix.length != rows \* cols || str == null){  
 return false;  
 }  
 boolean[] visited = new boolean[matrix.length];  
 for(int i = 0 ; i < rows ; i++){  
 for(int j = 0 ; j < cols ; j++){  
 //以矩阵中的每个点作为起点尝试走出str对应的路径  
 if(hasPathCore(matrix, rows, cols, i, j, str, 0, visited)){  
 return true;  
 }  
 }  
 }  
 return false;  
}  
  
//当前在矩阵的(i,j)位置上  
//index -> 匹配到了str中的第几个字符  
private boolean hasPathCore(char[] matrix, int rows, int cols, int i, int j,   
 char[] str, int index, boolean[] visited){  
 if(index == str.length){  
 return true;  
 }  
 //越界或字符不匹配或该位置已在路径上  
 if(!match(matrix, rows, cols, i, j, str[index]) || visited[i \* cols + j] == true){  
 return false;  
 }  
 visited[i \* cols + j] = true;  
 boolean res = hasPathCore(matrix, rows, cols, i + 1, j, str, index + 1, visited) ||  
 hasPathCore(matrix, rows, cols, i - 1, j, str, index + 1, visited) ||  
 hasPathCore(matrix, rows, cols, i, j + 1, str, index + 1, visited) ||  
 hasPathCore(matrix, rows, cols, i, j - 1, str, index + 1, visited);  
 visited[i \* cols + j] = false;  
 return res;  
}  
  
//矩阵matrix中的(i,j)位置上是否是c字符  
private boolean match(char[] matrix, int rows, int cols, int i, int j, char c){  
 if(i < 0 || i > rows - 1 || j < 0 || j > cols - 1){  
 return false;  
 }  
 return matrix[i \* cols + j] == c;  
}

### 机器人的运动范围

#### 题目描述

地上有一个m行和n列的方格。一个机器人从坐标0,0的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于k的格子。 例如，当k为18时，机器人能够进入方格（35,37），因为3+5+3+7 = 18。但是，它不能进入方格（35,38），因为3+5+3+8 = 19。请问该机器人能够达到多少个格子？

#### 解析

定义一个黑盒walk(int threshold, int rows, int cols, int i, int j, boolean[] visited)，它表示统计从rows行cols列的矩阵中的(i,j)开始所能到达的格子并返回，对于当前位置(i,j)有如下判断：

1. (i,j)是否越界矩阵了
2. (i,j)是否已被统计过了
3. (i,j)的行坐标和列坐标的数位之和是否大于k

如果通过了上面三重检查，则认为(i,j)是可以到达的（res=1），并标记(i,j)的visited为true表示已被统计过了，然后对(i,j)的上下左右的格子调用黑盒进行统计。

这里要注意的是，与上一题不同，visited不会在递归计算完子状态后被重置回false，因为每个格子只能被统计一次。visited的含义不一样

public int movingCount(int threshold, int rows, int cols){  
 if(threshold < 0 || rows < 0 || cols < 0){  
 return 0;  
 }  
 boolean[] visited = new boolean[rows \* cols];  
 return walk(threshold, rows, cols, 0, 0, visited);  
}  
  
private int walk(int threshold, int rows, int cols, int i, int j, boolean[] visited){  
 if(!isLegalPosition(rows, cols, i, j) || visited[i \* cols + j] == true  
 || bitSum(i) + bitSum(j) > threshold){  
 return 0;  
 }  
 int res = 1;  
 visited[i \* cols + j] = true;  
 res += walk(threshold, rows, cols, i + 1, j, visited) +  
 walk(threshold, rows, cols, i - 1, j, visited) +  
 walk(threshold, rows, cols, i, j + 1, visited) +  
 walk(threshold, rows, cols, i, j - 1, visited);  
 return res;  
}  
  
private boolean isLegalPosition(int rows, int cols, int i, int j){  
 if(i < 0 || j < 0 || i > rows - 1 || j > cols - 1){  
 return false;  
 }  
 return true;  
}  
  
public int bitSum(int num){  
 int res = num % 10;  
 while((num /= 10) != 0){  
 res += num % 10;  
 }  
 return res;  
}