1.调整数组顺序使奇数位于偶数前面

题目：输入一个整数数组，调整数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

解题思路：维护两个指针，第一个指针初始化为数组的第一个数字，它只向后移动；第二个指针初始化为数组的最后一个数字，它只向前移动。在两个指针相遇之前，第一个指针总是位于第二个指针的前面。如果第一个指针指向的数字是偶数而第二个指针指向的数字是奇数，我们就交换这两个数字。

public class Test{

public static void main(String[] args){

int[] arr = {2,1,6,5,8,9,3,6,5,7};

Reorder(arr);

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

System.out.print(arr[i]+",");

}

}

//判断n是否为偶数，偶数返回true，奇数返回false

public static boolean isEven(int n ){

if(n%2==0){

return true;

}

else

return false;

}

public static void Reorder(int[] arr){

if(arr == null){

return ;

}

int small = 0;

int big = arr.length-1;

while(small<=big){

if(!isEven(arr[small]))small++;

if(isEven(arr[big])) big--;

if(isEven(arr[small])&&!isEven(arr[big])){

int temp = arr[small];

arr[small] = arr[big];

arr[big] = temp;

}

}

}

}

2.最小生成树

Kruskal算法

T是最小生成树的边的集合，初始为空；E是带全无向图的变的集合

开始从E中选一条具有最小权重的边(v,w)，从E中删除(v,w)，如果(v,w)加入T之后不产生回路，则将(v,w)加入T，否则放弃T；直到T中有n-1条边且E为空

Prim算法

Em是最小生成树的边集合，Vm是其顶点集合；从E中选一条边(u,v),u属于Vm且v属于V，且具有最小的权值(选出一条连接两个顶点集合且权值最小的边)，直到Vm中有n个顶点

3．二叉搜索树

它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值

查找算法：

若树为空，则搜索失败；否则：如果目标等于根节点，查找成功；如果小于根节点，递归搜索左子树；否则递归搜索右子树

插入算法：如果树为空，则将目标作为根节点插入；如果目标等于树的根节点，返回；如果目标大于根节点值，则插入右节点，否则插入左节点(递归)

4.删除单链表中的节点

将要删除的节点的值与next修改为要删除节点的next

public class Solution {

public void deleteNode(ListNode node) {

node.val = node.next.val;

node.next = node.next.next;

}

}

5.找到重复数字

有一个数组有n+1个整数，所有的数都在[1, n]区域内。假设在数组中只有一个数字是重复的，找到这个数字。

考虑用二分搜索法，我们在区间[1, n]中搜索，首先求出中点mid，然后遍历整个数组，统计所有小于等于mid的数的个数，如果个数大于mid，则说明重复值在[mid+1, n]之间，反之，重复值应在[1, mid-1]之间，然后依次类推，知道搜索完成，此时的low就是我们要求的重复值。

public class Solution {

public int findDuplicate(int[] nums) {

int low =1;

int high = nums.length;

while (low <= high) {

int mid = low + (high - low)/2;

int cnt = 0;

for (int a : nums) {

if (a <= mid){

++cnt;

}

}

if (cnt <= mid) {

low = mid + 1;

}

else{

high = mid - 1;

}

}

return low;

}

}

6.反转单链表

7.判断一棵树是否是二叉搜索树

public class Solution {

public boolean isValidBST(TreeNode root) {

return isValidBST(root, Double.NEGATIVE\_

INFINITY, Double.POSITIVE\_INFINITY);

}

public boolean isValidBST(TreeNode p, double min, double max){

if(p==null) {

return true;

}

if(p.val <= min || p.val >= max){

return false;

}

return isValidBST(p.left, min, p.val) && isValidBST(p.right,

p.val, max);

}

}

8.二叉树非递归遍历

9.移动0:将一个数组中的0移动到数组的后面，其余数字的相对顺序不改变。

//使用了两个索引，一个指向0，一个指向非0，二者分别向后遍历，当出现一对（0，非0）时，交换二者位置，这样当遍历到结束时，此时0也就被移动到了最后

public void moveZeroes(int[] nums) {

int zeroIndex = 0;

int nonZeroIndex = 0;

int size = nums.length;

while(zeroIndex<size && nonZeroIndex<size){

while(zeroIndex<size && nums[zeroIndex]!=0){

zeroIndex++;

}

while(nonZeroIndex < size && (nums[nonZeroIndex]==0 || nonZeroIndex<zeroIndex)) {

nonZeroIndex++;

}

//swap the value

if(zeroIndex<size && nonZeroIndex<size){

nums[zeroIndex++] = nums[nonZeroIndex];

nums[nonZeroIndex++] = 0;

}

}

}

10.一个数组中存有整数，除一个外其他都出现了两次，找出这个数

public int singleNumber(int[] nums) {

int x = 0;

for (int a : nums) {

x = x ^ a;

}

return x;

}

与0做异或，出现两次的做两次异或会变为0；最后的结果就是只出现了一次的数字

11.树的子结构：输入两棵二叉树A，B，判断B是不是A的子结构

public class Solution {

public boolean isSubtree(TreeNode pRootA, TreeNode pRootB) {

if (pRootB == null) {

return true;

}

if (pRootA == null) {

return false;

}

if (pRootB.val == pRootA.val) {

return isSubtree(pRootA.left, pRootB.left)

&& isSubtree(pRootA.right, pRootB.right);

} else return false;

}

public boolean HasSubtree(TreeNode pRootA, TreeNode pRootB) {

if (pRootA == null || pRootB == null){

return false;

}

// 判断A是否与B有相同的结构或者A的左右节点为根的子树是否与B有相同结构

return isSubtree(pRootA, pRootB) ||

HasSubtree(pRootA.left, pRootB) ||

HasSubtree(pRootA.right, pRootB);

}

}

12.顺时针打印矩形

按层打印方框

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

int row = matrix.length;

int col = matrix[0].length;

int startX = 0;

int startY = 0;

int endY = col - 1;

int endX = row - 1;

while ((startX <= endX) && (startY <= endY)) {

printCircle(matrix, startX, startY, endX, endY);

startX++;

startY++;

endX--;

endY--;

}

return list;

}

public void printCircle(int[][] matrix, int startX, int startY, int endX, int endY){

// only one column left

if (startY == endY) {

for (int i = startX; i <= endX; i++ ) {

list.add(matrix[i][endY]);

}

return;

}

// only one row left

if (startX == endX) {

for (int i = startY; i <= endY; i++ ) {

list.add(matrix[startX][i]);

}

return;

}

for (int i = startY; i < endY; i++ ) {

list.add(matrix[startX][i]);

}

for (int i = startX; i < endX; i++ ) {

list.add(matrix[i][endY]);

}

for (int i = endY; i > startY; i-- ) {

list.add(matrix[endX][i]);

}

for (int i = endX; i > startX; i-- ) {

list.add(matrix[i][startY]);

}

}

13.合并两个排序的链表：输入两个单调递增的链表，输出两个链表合成后的链表，要求合成后的链表满足单调不减规则

public ListNode Merge(ListNode list1,ListNode list2) {

if(list1==null){

return list2;

}

if(list2==null){

return list1;

}

ListNode mergeHead = null;

//递归

if(list1.val<list2.val){

mergeHead = list1;

mergeHead.next = Merge(list1.next,list2);

}else{

mergeHead = list2;

mergeHead.next= Merge(list1,list2.next);

}

return mergeHead;

}

14.栈的压入弹出序列

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

public boolean IsPopOrder(int[] pushA,int[] popA) {

//2333

Stack stack = new Stack();

if( pushA.length == 0 && popA.length == 0 ) return false;

for( int i=0,j=0; i < pushA.length; i++ ){

stack.push( pushA[i] );//

while( ( !stack.empty() )&& ( stack.peek() == popA[j]) ){

stack.pop();

j ++;

}

}

return stack.empty() == true;

}

15.二叉树的层次遍历

使用一个队列存储节点;开始将根节点入队列，访问一个节点之后，将其左右子节点如队列；直到队列为空

public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {

ArrayList<Integer> list = new ArrayList<Integer>();

if(root==null){

return list;

}

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

q.add(root);

while(!q.isEmpty()){

TreeNode node = q.remove();

list.add(node.val);

if(node.left!=null){

q.add(node.left);

}

if(node.right!=null){

q.add(node.right);

}

}

return list;

}

16.二叉树的前中后序遍历非递归算法

17.二叉搜索树的后序遍历序列

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

二叉搜索树的后序遍历序列，最后一个一定是根节点，所以除根节点外的序列前半部分是左子树的节点，一定比根节点小，后半部分一定比根节点大；先根据此条件判断，然后对左右子树分别递归调用该方法，全部返回true则是一个后序遍历序列

public boolean VerifySquenceOfBST(int [] sequence) {

if(sequence==null||sequence.length==0){

return false;

}

int root=sequence[sequence.length-1];

int i=0;

for(;i<sequence.length-1;i++){

if(sequence[i]>root){

break;

}

}

int j=i;

for(;j<sequence.length-1;j++){

if(sequence[j]<root)

return false;

}

boolean left=true;

boolean right=true;

if(i>0){

left=VerifySquenceOfBST(Arrays.copyOfRange(sequence, 0, i));

}

if(i<sequence.length-1){

right=VerifySquenceOfBST(Arrays.copyOfRange(sequence, i, sequence.length-1));

}

return (left&&right);

}

18.求树的高度

递归,树的高度等于1+左右子树的最大高度

public int TreeDepth(TreeNode root) {

if(root!=null){

return Math.max(1+TreeDepth(root.left), 1+TreeDepth

(root.right));

}else{

return 0;

}

}

19.找到和为N的两个数字

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，是的他们的和正好是S

初始化两个指针，I,j分别指向数组的两端，计算两个指针指向的数字的和。如果和大于S，则j--;如果小于S，则i++；

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

ArrayList<Integer>list = new ArrayList<Integer>();

if(array.length==0){

return list;

}

int len = array.length;

int small=0;

int big = len-1;

while(small<big){

int tsum = array[small]+array[big];

if(tsum==sum){

list.add(array[small]);

list.add(array[big]);

return list;

}else if(tsum>sum){

big--;

}else if(tsum<sum){

small++;

}

}

return list;

}

19.根据前序遍历序列和中序序列重建二叉树

前序遍历的第一个元素就是根节点的val值;根据根节点能得到左右子树的前序和中序序列,左右子节点分别由递归调用得到

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

if (pre == null || in == null) {

return null;

}

int len = pre.length;

if (len == 0) {

return null;

}

int val = pre[0];//二叉树的根节点

int rootPositionInLDR = -1;

TreeNode root = new TreeNode(val);

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

if (in[i] == val) {

rootPositionInLDR = i;//寻找中序遍历序列根节点的位置,分割左右子树

}

}

if (rootPositionInLDR == -1) {

return null;

}

// rebulid left child tree

int leftChildTreeNum = rootPositionInLDR;//左子树长度

int[] leftPre = new int[leftChildTreeNum];//左子树的前序序列;前序序列的第二个元素到第个元素

System.arraycopy(pre,1,leftPre,0,leftChildTreeNum);

int[] leftIn = new int[leftChildTreeNum];//左子树的中序序列;中序序列的前n个

System.arraycopy(in,0,leftIn,0,leftChildTreeNum);

root.left = reConstructBinaryTree(leftPre, leftIn);

// rebulid right child tree

int rightChildTreeNum = len - rootPositionInLDR - 1;//右子树长度

int[] rightPre = new int[rightChildTreeNum];

System.arraycopy(pre,1+leftChildTreeNum,rightPre,0,rightChildTreeNum);

int[] rightIn = new int[rightChildTreeNum];

System.arraycopy(in,1+leftChildTreeNum,rightIn,0,rightChildTreeNum);

root.right = reConstructBinaryTree(rightPre, rightIn);

return root;

}

20. 二叉树中和为某一值的路径

输入一颗二叉树和一个整数，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

ArrayList<ArrayList<Integer>> pathList=new ArrayList

<ArrayList<Integer>>();

Stack<Integer> path=new Stack<Integer>();

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

if (root == null) {

return pathList;

}

if (root.left == null && root.right == null) {//如果遍历到了叶节点

if (root.val == target) {

ArrayList<Integer> list = new ArrayList<Integer>();

for (int i : path) {//将栈中的元素全部取出放入list,作为一条路径

list.add(new Integer(i));

}

list.add(new Integer(root.val));

pathList.add(list);

}

} else {//尚未到叶节点,从左右子树中寻找和为target-root.val的路径

path.push(new Integer(root.val));

FindPath(root.left, target - root.val);

FindPath(root.right, target - root.val);

path.pop();

}

return pathList;

}

21.寻找数组中出现次数超过一半的数字

遍历数组的时候保存两个值：一个是数组中的一个数字，一个是次数。当我们遍历到下一个数字的时候，如果下一个数字和我们之前保存的数字相同，则次数加1；如果下一个数字和我们之前保存的数字不同，则次数减1。  
如果次数为零，我们需要保存下一个数字，并把次数设为1。由于我们要找的数字出现的次数比其他所有数字出现的次数之和还要多，那么要找的数字肯定是最后一次把次数设为1时对应的数字。

public int MoreThanHalfNum\_Solution(int [] array) {

if(array==null||array.length==0){

return 0;

}

int more = array[0];

int num = 0;

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

if(array[i]==more){

num++;

}else{

num--;

if(num==0){

more = array[i+1];

}

}

}

int times = 0;

for(int i:array){

if(i==more){

times++;

}

}

if(times<=array.length/2){

return 0;

}

return more;

}

22.给定一棵二叉树，判断其是否是平衡二叉树

计算左右子树的高度；如果相差大于一，则不是平衡二叉树；否则判断其左右子树是否是平衡二叉树

public boolean IsBalanced\_Solution(TreeNode root) {

if(root==null){

return true;

}else{

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

return false;

}else{

return IsBalanced\_Solution(root.left)&&IsBalanced\_Solution(root.right);

}

}

}

public int deep(TreeNode root){

if(root==null){

return 0;

}else{

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

}

}

23.求连续子数组的最大和

非分治算法;找到和不为0的起始元素与和大的结尾元素

public int FindGreatestSumOfSubArray(int[] array) {

int sum = 0;

int maxSum = Integer.MIN\_VALUE;

int start=0,end=0;

int length = array.length;

if(length <= 0 || array == null){

return 0;

}

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

sum += array[i];

if(sum >= maxSum){

maxSum = sum;

end = i;

}

if(sum < 0){

sum = 0;

if(i<=end){

start = i;

}

}

}

return maxSum;

}