# 树

定义：n个节点的有限集，在任意一棵非空树中:

（1）有且仅有一个特定的称为根的节点；

（2）当n>1时，其余节点可以分为m个互不相交的有限集，其中每个集合本身又是一棵

树并且称为根的子树。

# 二叉树

1.定义：每个结点至多有两棵子树的树，并且子树存在左右之分，次序不可颠倒

2.性质：

1）在二叉树的第i层上至多有2的i-1次方个结点

2）深度为k的二叉树至多有2的k次方-1个结点（当相等时为完全二叉树）

3）二叉树叶子结点数为n0，度为2的结点数为n1,则n0=n2+1

4)n个结点的完全二叉树，对任一结点i

    i=1，i是二叉树的根

   2i>n，则结点i无左孩子

   2i+1>n,则结点i无右孩子

   完全二叉树叶子结点数n0=[n/2](n为奇数，n=n+1)

## 二叉树的遍历

## 1.前序遍历（先序遍历）

若二叉树为空，则空操作；

否则，先访问根节点；先序遍历左子树，先序遍历右子树。

public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

### 递归算法：

**public** **class** Preorderdigui {

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

**public** ArrayList<Integer> preOrder(TreeNode root){

//ArrayList<Integer> list = new ArrayList<Integer>();//犯了无数次的错误

**if**(root == **null**)

**return** list;

list.add(root.val);

//System.out.println(root.val);

**if**(root.left != **null**)

preOrder(root.left);

**if**(root.right != **null**)

preOrder(root.right);

**return** list;

}

**public** **static** **void** main(String[] args) {

Preorderdigui p = **new** Preorderdigui();

TreeNode a = **new** TreeNode(1);

TreeNode b = **new** TreeNode(2);

TreeNode c = **new** TreeNode(3);

TreeNode d = **new** TreeNode(4);

TreeNode e = **new** TreeNode(6);

TreeNode f = **new** TreeNode(5);

TreeNode g = **new** TreeNode(7);

a.left = b;

a.right = c;

b.left = d;

b.right = f;

d.right = e;

f.left = g;

ArrayList<Integer> result = p.preOrder(a);

System.***out***.println(result);

}

### }

### 非递归算法：

**public** **class** Preorder {

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

**public** ArrayList<Integer> preoder(TreeNode root){

Stack<TreeNode> sta = **new** Stack<TreeNode>();

**if**(root == **null**)

**return** list;

**while**(root != **null** || !sta.isEmpty()){

**while**(root != **null**){

list.add(root.val);//在压栈之前将左孩子访问

sta.push(root);//将所有的左孩子压栈

root = root.left;

}

**if**(!sta.isEmpty()){

root = sta.pop();

root = root.right;

}

}

**return** list;

}

**public** **static** **void** main(String[] args) {

……

}

}

## 2.中序遍历

### 递归算法：

**public** **class** Inoderdigui {

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

**public** ArrayList<Integer> inOrder(TreeNode root){

**if**(root == **null**)

**return** list;

**if**(root.left != **null**)

inOrder(root.left);

list.add(root.val);

//System.out.println(root.val);

**if**(root.right != **null**)

inOrder(root.right);

**return** list;

}

**public** **static** **void** main(String[] args) {

……

}

}

### 非递归算法：

**public** **class** Inoder {

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

**public** ArrayList<Integer> inorder(TreeNode root){

**if**(root == **null**)

**return** list;

Stack<TreeNode> sta = **new** Stack<TreeNode>();

**while**(root != **null** || !sta.isEmpty()){

**while**(root != **null**){

sta.push(root);

root = root.left;

}

**if**(!sta.isEmpty()){

root = sta.pop();

list.add(root.val);

root = root.right;

}

}

**return** list;

}

**public** **static** **void** main(String[] args) {

……

}

}

## 3.后序遍历

### 递归算法：

**public** **class** Postorder {

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

**public** ArrayList<Integer> postorder(TreeNode root){

**if**(root == **null**)

**return** list;

**if**(root.left != **null**)

postorder(root.left);

**if**(root.right != **null**)

postorder(root.right);

list.add(root.val);

**return** list;

}

**public** **static** **void** main(String[] args) {

……

}

}

### 非递归算法：

**public** **class** Postorderdigui {

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

**public** ArrayList<Integer> postorderdigui(TreeNode root){

Stack<TreeNode> sta = **new** Stack<TreeNode>();

**if**(root == **null**)

**return** list;

TreeNode pre = root;

**while**(root != **null** || !sta.isEmpty()){

**while**(root != **null**){

sta.push(root);

root = root.left;

}

**if**(!sta.isEmpty()){

TreeNode temp = sta.peek().right;

**if**(temp == **null** || temp == pre){

root = sta.pop();

list.add(root.val);

pre = root;

root = **null**;

}

**else**{

root = temp;

}

}

}

**return** list;

}

**public** **static** **void** main(String[] args) {

……

}

}

## 4.层次遍历

### 求二叉树的深度：

**public** **class** Depth {

**public** **static** **int** getDepth(TreeNode root){

Queue<TreeNode> que = **new** LinkedList<TreeNode>();

**if**(root == **null**)

**return** 0;

que.add(root);

**int** layer = 0;

**int** count = 0;

**int** nodenum = 1;

**while**(!que.isEmpty()){

root = que.poll();

count++;

**if**(root.left != **null**)

que.add(root.left);

**if**(root.right != **null**)

que.add(root.right);

**if**(count == nodenum){

layer++;

count = 0;

nodenum = que.size();

}

}

**return** layer;

}

**public** **static** **void** main(String[] args) {

……

}

}

### 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** **class** Solution26 {

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

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

**if**(root == **null**)

**return** list;

Queue<TreeNode> que = **new** LinkedList<TreeNode>();

que.add(root);

**while**(!que.isEmpty()){

TreeNode node = que.poll();

**if**(node.left != **null**)

que.add(node.left);

**if**(node.right != **null**)

que.add(node.right);

list.add(node.val);

}

**return** list;

}

}

### 从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行

**public** **class** Solution27 {

**public** ArrayList<ArrayList<Integer>> Print(TreeNode root){

ArrayList<ArrayList<Integer>> result = **new** ArrayList<ArrayList<Integer>>();//result中的数据类型是list型

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

Queue<TreeNode> que = **new** LinkedList<TreeNode>();

**if**(root == **null**)

**return** result;

que.add(root);

**int** count = 0;

**int** nodenum = 1;

**while**(!que.isEmpty()){

TreeNode node = que.poll();

count++;

**if**(node.left != **null**)

que.add(node.left);

**if**(node.right != **null**)

que.add(node.right);

list.add(node.val);

**if**(count == nodenum){

count = 0;

nodenum = que.size();

result.add(list);

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

}

}

**return** result;

}

}

### 按Z字型顺序打印二叉树

**public** **class** Solution31 {

**public** ArrayList<ArrayList<Integer>> Print(TreeNode pRoot){

ArrayList<ArrayList<Integer>> result = **new** ArrayList<ArrayList<Integer>>();

Stack<TreeNode> s1 = **new** Stack<TreeNode>();//s1用于存放奇数层的结点

Stack<TreeNode> s2 = **new** Stack<TreeNode>();//s2用于存放偶数层的结点

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

s1.add(pRoot);

**int** layer = 1;

**while**(!s1.isEmpty() || !s2.isEmpty()){

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

**if**(layer % 2 != 0){//奇数层

**while**(!s1.isEmpty()){

TreeNode pNode = s1.pop();

**if**(pNode != **null**){

list.add(pNode.val);

**if**(pNode.left != **null**)

s2.add(pNode.left);

**if**(pNode.right != **null**)

s2.add(pNode.right);

}

}

**if**(!list.isEmpty()){

result.add(list);

layer++;

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

}

}

**else**{

**while**(!s2.isEmpty()){

TreeNode pNode = s2.pop();

**if**(pNode != **null**){

list.add(pNode.val);

**if**(pNode.right != **null**)

s1.add(pNode.right);

**if**(pNode.left != **null**)

s1.add(pNode.left);

}

}

**if**(!list.isEmpty()){

result.add(list);

layer++;

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

}

}

}

**return** result;

}

}

## 重建二叉树

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。

public class Solution {

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

if(pre.length != in.length)

return null;

TreeNode root = reConstructBinaryTree(pre, 0, pre.length-1, in, 0, in.length-1);

return root;

}

private TreeNode reConstructBinaryTree(int[] pre, int prestart, int preend, int[] in,int instart, int inend) {

// TODO Auto-generated method stub

if(prestart > preend || instart > inend )

return null;

TreeNode root = new TreeNode(pre[prestart]);

for(int i = instart; i<= inend; i++){

if(in[i] == pre[prestart]){

root.left = reConstructBinaryTree(pre, prestart+1, **prestart+i-instart,** in, instart, i-1);

root.right = reConstructBinaryTree(pre,prestart+i-instart+1, preend, in, i+1, inend);

}

}

return root;

}

}

## 普通二叉树，找一下两个节点的LCA （最小公共祖先）

## 树的子结构

## 二叉树的镜像

操作给定的二叉树，将其变换为源二叉树的镜像。

递归：

        if(pRoot==NULL)

           return;

        TreeNode \*ptemp=pRoot->left;

        pRoot->left=pRoot->right;

        pRoot->right=ptemp;

        if(pRoot->left)

            Mirror(pRoot->left);

        if(pRoot->right)

            Mirror(pRoot->right);

非递归:

public class Solution {

public void Mirror(TreeNode root) {

if(root == null)

return;

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

return;

Stack<TreeNode> sta = new Stack<TreeNode>();

sta.push(root);

while(!sta.isEmpty()){

TreeNode node = sta.pop();

if(node.left != null || node.right != null){

TreeNode temp = node.left;

node.left = node.right;

node.right= temp;

if(node.left != null)

sta.push(node.left);

if(node.right != null)

sta.push(node.right);

}

}

}

}

## 判断二叉树是否对称

递归：

public class Solution {

boolean isSymmetrical(TreeNode pRoot)

{

if(pRoot == null)

return true;

return isSymmetrical(pRoot.left, pRoot.right);

}

private Boolean isSymmetrical(TreeNode left, TreeNode right) {

// TODO Auto-generated method stub

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

return true;

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

return false;

if(left.val != right.val)

return false;

return isSymmetrical(left.left, right.right) && isSymmetrical(left.right, right.left);

}

}

非递归：

先求出该二叉树的镜像，然后判断二者是否相等。

## 二叉树中和为某一值的路径

import java.util.ArrayList;

/\*

\* 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。

\*

\* 递归

\*/

public class Solution30 {

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

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

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

if(root == null)

return paths;

path.add(root.val);

target = target - root.val;

if(root.left == null && root.right == null && target == 0){

paths.add(path);

path = new ArrayList<Integer>(path);//不能把path刷新，要保存path的路径

}

//如果不是叶子节点，则遍历他的子节点

if(root.left != null){

FindPath(root.left, target);

}

if(root.right != null){

FindPath(root.right, target);

}

//在返回父节点之前，在路径上删除当前结点

path.remove(path.size()-1);

return paths;

}

}

## 二叉树的下一个结点

## 平衡二叉树

## 二叉树的下一个结点

## 序列化二叉树

## 二叉树任意两个节点之间路径的最大长度

## 排序树，添加节点并计数

已知有一颗二叉排序树，向树里面插入节点，如果该节点已存在（节点值相等），将节点中的count字段加一；如果不存在，将节点插入树中，并将节点的count值置为1。自行设计数据结构，插入算法并且分析算法的复杂度.

# 链表

定义链表的结构：

public class ListNode {

int val;

ListNode next = null;

ListNode(int val) {

this.val = val;

}

}

## 两条链表找第一个交叉结点

public class Solution {

public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {

int length1 = getLength(pHead1);

int length2 = getLength(pHead2);

int diff = length1 - length2;

ListNode plong = pHead1;

ListNode pshort = pHead2;

if(diff < 0){

plong = pHead2;

pshort = pHead1;

diff = length2 - length1;

}

//先在长表上先走几步，再同时在两个链表上遍历；

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

plong = plong.next;

}

while(plong != null && pshort != null && plong != pshort){

plong = plong.next;

pshort = pshort.next;

}

ListNode first = plong;

return first;

}

public static int getLength(ListNode pHead){

int length = 0;

ListNode p = pHead;

while(p != null){

length ++;

p = p.next;

}

return length;

}

}

## 从尾到头打印链表

import java.util.ArrayList;

import java.util.ArrayList;

import java.util.Stack;

public class Solution {

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

Stack<ListNode> sta = new Stack();

ListNode pNode = listNode;

ArrayList<Integer> result = new ArrayList();

ListNode p = null;

while(pNode != null){

sta.push(pNode);

pNode = pNode.next;

}

while(!sta.empty()){

p = sta.pop();

result.add(p.val);

}

return result;

}

}

## 如何判断一个链表有环

1.第一步，找环中相汇点。分别用p1，p2指向链表头部，p1每次走一步，p2每次走二步，直到p1==p2找到在环中的相汇点。  
2.找环的入口。

public ListNode EntryNodeOfLoop(ListNode pHead)

{

ListNode meetingNode = MeetingNode(pHead);

if(meetingNode == null)

return null;

//得到环中结点的数目

int nodeInLoop = 1;

ListNode p1 = meetingNode;

while(p1.next != meetingNode){

p1 = p1.next;

nodeInLoop++;

}

//求出环的长度

p1 = p1.next;

ListNode p3 = pHead;

while(p1 != p3){

p1 = p1.next;

p3 = p3.next;

}

return p1;

}

//判断是否有环，寻找相遇节点

public static ListNode MeetingNode(ListNode pHead){

if(pHead == null){

return null;

}

ListNode pslow = pHead.next;

if(pslow == null){

return null;

}

ListNode pfast = pslow.next;

while(pslow != null && pfast != null){

if(pfast == pslow){

return pfast;

}

pslow = pslow.next;

pfast = pfast.next.next;

}

return null;

}

}

## 如何取到链表倒数第k个数

public class Solution {

public ListNode FindKthToTail(ListNode head,int k) {

if (head == null || k == 0){

return null;

}

ListNode right = head;

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

if(right.next != null){

right = right.next;

}

else{

return null;//链表的节点可能不够k个

}

}

ListNode left = head;

while(right.next != null){

right = right.next;

left = left.next;

}

return left;

}

}

## 手撕代码，单链表反转

public class Solution {

public ListNode ReverseList(ListNode head) {

if (head == null)

return null;

ListNode pre = null;

ListNode next = null;

while(head != null){

//先用next保存head的下一个节点的信息，保证单链表不会因为失去head节点的原next节点而就此断裂

//保存完next，就可以让head从指向next变成指向pre了

//head指向pre后，就继续依次反转下一个节点

//让pre，head，next依次向后移动一个节点，继续下一次的指针反转

next = head.next;

head.next = pre;

pre = head;

head = next;

}

return pre;

}

}

## 合并两个排序的链表

public Class Solution {

   public static ListNode Merge(ListNode pHead1, ListNode pHead2)

    {

        ListNode result = NULL;

        ListNode current = NULL;

        if(pHead1 == NULL)

            return pHead2;

        if(pHead2 == NULL)

            return pHead1;

        /\*递归

        if(pHead1->val <= pHead2->val){

            result = pHead1;

            result->next = Merge(pHead1->next, pHead2);

        } else {

            result = pHead2;

            result->next = Merge(pHead1, pHead2->next);

        }

        \*/

        while(pHead1 != NULL && pHead2 != NULL){

            if(pHead1->val <= pHead2->val){

                if(result == NULL){

                    current = result = pHead1;

                } else {

                    current->next = pHead1;

                    current = current->next;

                }

                pHead1 = pHead1->next;

            } else {

                if(result == NULL){

                    current = result = pHead2;

                } else {

                    current->next = pHead2;

                    current = current->next;

                }

                pHead2 = pHead2->next;

            }

        }

        if(pHead1 == NULL){

            current->next = pHead2;

        }

        if(pHead2 == NULL){

            current->next = pHead1;

        }

        return result;

    }

}

## 删除链表中的重复结点

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 class Solution {

public ListNode deleteDuplication(ListNode pHead)

{

if(pHead == null)

return null;

ListNode prenode = null;

ListNode node = pHead;

while(node != null){

if(node.next != null && node.val == node.next.val){

int value = node.val;

while(node.next != null && value == node.next.val){

node = node.next;

}

if (prenode == null){

pHead = node.next;

}

else{

prenode.next = node.next;

}

}

else{

prenode = node;

}

node = node.next;

}

return pHead;

}

}

## 双向链表实现队列

链表尾部插入，链表头部删除

public class DlinkQue {

private Node first;//最左边的头节点

private Node last;//最右边的尾节点

private int N;//记录当前链表长度

private class Node{

private Node pre;

private Node next;

private int val;

}

//返回存贮节点个数

public int size(){

注意写size函数

return N;

}

//是否为空

private boolean isEmpty(){

和empty函数

return N==0;

}

//右进

public void pushRight(int item){

if(isEmpty()){

last=new Node();

last.val =item;

first=last;

}else{

Node oldLast=last;

last=new Node();

last.val=item;

last.pre=oldLast;

oldLast.next=last;

}

N++;

}

//左出

public int popLeft() throws Exception{

//当为空时报错

if(isEmpty()){

throw new Exception("链表为空");

}

int item=first.val;

first=first.next;

if(N==1){

first=last=null;//当只有一个节点时也准备出的时候，last和first同时置null

}

else{

first.pre=null;//丢弃前一个节点

}

N--;

return item;

}

}

## 复杂链表的复制

## 一个链表奇数位升序偶数位降序 让链表变成升序的

首先根据奇数位和偶数位拆分成两个链表。

然后对偶数链表进行反转。

最后将两个有序链表进行合并。

# 线程

## 四个线程实现买票

public class Maipiao4 {

public static void main(String [] args){

Ticket t = new Ticket();

Thread t1 = new Thread(t);//创建了一个线程；

Thread t2 = new Thread(t);//创建了一个线程；

Thread t3 = new Thread(t);//创建了一个线程；

Thread t4 = new Thread(t);//创建了一个线程；

t1.start();

t2.start();

t3.start();

t4.start();

}

}

class Ticket implements Runnable//extends Thread

{

private int tick = 100;

public void run()

{

try {

while(true)

{

synchronized(**this**){

if(tick>0)

{

System.out.println(Thread.currentThread().getName()+"....sale : "+ tick--);

}

else {

break;

}

Thread.sleep(5000);

}

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

## 交替打印ABAB

一个lock,用wait（）-notify

public class ABAXiancheng {

private static Object lock= new Object();

private static int num = 0;//用于计数

public static void main(String[] args){

Thread thead1 = new Thread(){

public void run(){

while(num<=10){

synchronized(lock){

if(num%2==0){

System.out.print("A");

num++;

}

else{

//唤醒

lock.notifyAll();

try{

lock.wait(1000);//等待

}catch(Exception e){

e.printStackTrace();

}

}

}

}

}

};

Thread thead2 = new Thread(){

public void run(){

while(num<=10){

synchronized(lock){

if(num%2!=0){

System.out.print("B");

num++;

}

else{

lock.notifyAll();

try{

lock.wait(1000);

}catch(Exception e){

e.printStackTrace();

}

}

}

}

}

};

thead1.start();

thead2.start();

}

}

## 线程t1，t2执行完再执行t3

两个线程设计题。记得一个是：t1,t2,t3，让t1，t2执行完才执行t3，原生实现。

使用join或者使用线程池

## main10个线程执行完后再往下执行

**CountDownLatch**

public static void main(String[] args) throws Exception{

CountDownLatch c = new CountDownLatch(10);

for(int i= 0;i<10;i++){

new Thread(new Runnable(){

public void run() {

try{

System.out.println("子线程执行");

c.countDown();

}catch(Exception e){

e.printStackTrace();

}

}

}).start();

}

c.await();//主线程中阻塞，等待一组进程执行完

System.out.println("主线程执行");

}

**cyclicBarrier**

public static void main(String[] args) throws Exception{

Executor excutor = Executors.newFixedThreadPool(10);

CyclicBarrier c = new CyclicBarrier(11);//多取一个，主线程中也得等

for(int i = 0;i<10;i++){

excutor.execute(new Runnable(){

public void run(){

System.out.println(num++);

try{

c.await();

}catch(Exception e){

e.printStackTrace();

}

}

});

}

c.await();//主线程到达同步点

System.out.println("主线程执行");

}

# 查找

## 二分查找

## 1.有序数组查找第一个不小于给定值的数`

## 2.统计一个数字在排序数组中出现的次数。

## 3.查找和为k的两个数

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。

思路：两个指针，一个从第一个开始，一个从最后一个开始。

若两个指针的和<K，那么start++

若两个指针的和>K，那么end--

相等则为找到，继续查找

import java.util.ArrayList;

public class Solution {

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

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

if (array == null || array.length < 2) {

return list;

}

int i=0,j=array.length-1;

while(i<j){

if(array[i]+array[j]==sum){

list.add(array[i]);

list.add(array[j]);

return list;

}else if(array[i]+array[j]>sum){

j--;

}else{

i++;

}

}

return list;

}

}

# 未分类题目

## 两个非常大的自然数相加，结果可能会超出long的范围，问我可以怎么做到正确显示结果。

（把两个自然数转化成了String，然后分别对每一位相加再考虑进位，最后用一个StringBuilder对象储存再打印）

## 输出k对括号的匹配

输出k对括号的全部正确匹配方案，如k=2,输出()(),(())

## 从1到n整数中1出现的次数

# 数组

## 最大子数组求和问题

## 有序数组的合并

## 把数组排成最小的数

\* 先将整型数组转换成String数组，然后将String数组排序，最后将排好序的字符串数组拼接出来。关键就是制定排序规则

public String PrintMinNumber(int [] numbers) {

        if(numbers == null || numbers.length == 0) return "";

        int len = numbers.length;

        String[] str = new String[len];

        StringBuilder sb = new StringBuilder();

        for(int i = 0; i < len; i++){

            str[i] = String.valueOf(numbers[i]);

        }

        Arrays.sort(str,new Comparator<String>(){

            @Override

            public int compare(String s1, String s2) {

                String c1 = s1 + s2;

                String c2 = s2 + s1;

                return c1.compareTo(c2);

            }

        });

        for(int i = 0; i < len; i++){

            sb.append(str[i]);

        }

        return sb.toString();

    }

## 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。

首先：位运算中异或的性质：**两个相同数字异或=0**，**一个数和0异或还是它本身**。

当**只有一个数出现一次**时，我们把数组中所有的数，依次异或运算，最后剩下的就是落单的数，因为成对儿出现的都抵消了。

依照这个思路，我们来看两个数（我们假设是AB）出现一次的数组。我们首先还是先异或，剩下的数字肯定是A、B异或的结果，**这个结果的二进制中的1，表现的是A和B的不同的位**。我们就取第一个1所在的位数，假设是第3位，接着把原数组分成**两组**，分组标准是第3位是否为1。如此，**相同的数肯定在一个组**，因为相同数字所有位都相同，而不同的数，**肯定不在一组**。然后把这两个组按照最开始的思路，依次异或，剩余的两个结果就是这两个只出现一次的数字。

public class Solution {

    public void FindNumsAppearOnce(int[] array, int[] num1, int[] num2)    {

        int length = array.length;

        if(length == 2){

            num1[0] = array[0];

            num2[0] = array[1];

            return;

        }

        int bitResult = 0;

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

            bitResult ^= array[i];

        }

        int index = **findFirst1**(bitResult);

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

            if(**isBit1**(array[i], index)){

                num1[0] ^= array[i];

            }else{

                num2[0] ^= array[i];

            }

        }

    }

    private int findFirst1(int bitResult){

        int index = 0;

        while(((bitResult & 1) == 0) && index < 32){

            bitResult >>= 1;

            index++;

        }

        return index;

    }

    private boolean isBit1(int target, int index){

        return ((target >> index) & 1) == 1;

    }

}

## 怎么用减法实现除法，怎么优化？

## 二进制中1的个数

思路：   
一个二进制数1100，从右边数起第三位是处于最右边的一个1。减去1后，第三位变成0，它后面的两位0变成了1，而前面的1保持不变，因此得到的结果是1011.我们发现减1的结果是把最右边的一个1开始的所有位都取反了。这个时候如果我们再把原来的整数和减去1之后的结果做与运算，从原来整数最右边一个1那一位开始所有位都会变成0。如1100&1011=1000.也就是说，把一个整数减去1，再和原整数做与运算，会把该整数最右边一个1变成0.那么一个整数的二进制有多少个1，就可以进行多少次这样的操作。

public class Solution {

    public int NumberOf1(int n) {

        int count = 0;

        while(n!= 0){

            count++;

            n = n & (n - 1);

         }

        return count;

    }

}

## 旋转有序数组，找到其中一个值

## 字符流中第一个不重复的字符

import java.util.LinkedHashMap;

import java.util.Map;

public class Solution {

private Map<Character, Integer> map = new LinkedHashMap<>();

//Insert one char from stringstream

public void Insert(char ch) {

if (map.containsKey(ch)) {

map.put(ch, map.get(ch) + 1);

} else {

map.put(ch, 1);

}

}

//return the first appearence once char in current stringstream

public char FirstAppearingOnce() {

for (Map.Entry<Character, Integer> set : map.entrySet()) {

if (set.getValue() == 1) {

return set.getKey();

}

}

return '#';

}

}

## 用两个栈实现一个队列

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(new Integer(node));

}

public int pop() {

Integer r = null;

if(!stack2.isEmpty()){

r = stack2.pop();

}

else{

while(!stack1.isEmpty()){

stack2.push(stack1.pop());

}

if(!stack2.isEmpty()){

r = stack2.pop();

}

}

return r;

}

}

## 将一个n位的数组右移k位

如：12345678 右移2位  78123456  
      思路如下：  
         1：首先逆序123456，那么数组变为65432178  
         2：再逆序78，那么数组变为65432187  
         3：最后逆序整个数组：65432187 ，那么数组变为78123456

public class Move {

public static void shift\_Right(int[] a, int k){

if(a.length < 0 || a.length == 0)

return;

int n = a.length-1;

shift(a,0,n-newk);

shift(a,n-newk+1,n);

shift(a,0,n);

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

System.out.println(a[j]);

}

}

private static void shift(int[] a, int begin, int end) {

// TODO Auto-generated method stub

while(begin <= end){

int temp = a[begin];

a[begin] = a[end];

a[end] = temp;

begin++;

end--;

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int[] a = new int[n];

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

a[i] = sc.nextInt();

}

int k = sc.nextInt();

shift\_Right(a,k);

}

}

## 翻转单词的顺序

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？

/\*

    算法思想：先翻转整个句子，然后，依次翻转每个单词。

    依据空格来确定单词的起始和终止位置

\*/

public class Solution {

    public String ReverseSentence(String str) {

        char[] chars = str.toCharArray();

        reverse(chars,0,chars.length - 1);

        int blank = -1;

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

            if(chars[i] == ' '){

                int nextBlank = i;

                reverse(chars,blank + 1,nextBlank - 1);

blank = nextBlank;

            }

        }

        reverse(chars,blank + 1,chars.length - 1);//最后一个单词单独进行反转

        return new String(chars);

    }

    public void reverse(char[] chars,int low,int high){

        while(low < high){

            char temp = chars[low];

            chars[low] = chars[high];

            chars[high] = temp;

            low++;

            high--;

        }

    }

}

## 用Java求出n！中有多少个0

解题思路：n！=1\*2\*3\*...\*n.那么会出现0的情况就是有5的时候，比如9！=1\*2\*3\*4\*5\*6\*7\*8\*9会出现9/5=1个0

10！=1\*2\*3\*4\*5\*6\*7\*8\*9\*10会出现10/5=2个0。

那么25！呢，因为25是5\*5所以算完1个5后还有一个5即25！=1\*2\*3\*...\*25=25/5+25/25=6个0.

而125是5\*5\*5，因此125！=1\*2\*3\*...\*125=125/5+125/25+125/125=31个0.找到这个规律，我们写下面的算法：

package N\_zero;



 public class Main {



 public static void main(String[] args){

 Integer result;

 result=n\_zero(100);

 System.out.println(result);

 }



 public static Integer n\_zero(Integer n){

 Integer result=0;

 while(n>0){

 n=n/5;

 result=result+n;

 }

 return result;

 }

 }

## 对角线打印矩阵

思路：  
将整个输出以最长的斜对角线分为两部分：右上部分和左下部分。  
右上部分：对角线的起点在第一行，列数递减，对角线上相邻元素之间横坐标和纵坐标均相差1；  
左下部分：对角线的起点在第一列上，行数递减，对角线上相邻元素之间横坐标和纵坐标均相差1；

public class Main {

public static void main(String[] args) {

Main main = new Main();

int[][] a = {{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};

main.print(a,4);

}

private void print(int a[][], int n) {

int row;

int col;

//输出右上角代码，包括对角线上的元素

for (int i = n - 1; i >= 0; i--) {//每次都是从第0行开始，所以需要row = 0,然后row col 同时自增

row = 0;

col = i;

while (row >= 0 && row < n && col >= 0 && col < n) {

System.out.println(a[row][col]);

row++;

col++;

}

}

//for输出左下角代码，对角线上的元素已经打印完了，所以在这里从1开始

for (int i = 1; i <= n - 1; i++) {//每次都是从第0列开始，所以需要col = 0,然后row col 同时自增

row = i;

col = 0;

while (row >= 0 && row < n && col >= 0 && col < n) {

System.out.println(a[row][col]);

row++;

col++;

}

}

}//print

## 顺时针打印矩阵

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.

import java.util.ArrayList;

public class Solution {

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

int left = 0;

int right = matrix[0].length - 1;

int top = 0;

int boom = matrix.length - 1;

//输出列表

ArrayList<Integer> list = new ArrayList();

while(left < right && boom > top){

//从左到右

for(int i = left; i <= right; i++){

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

}

//从上到下

for(int i = top + 1; i <= boom; i++){

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

}

//从右到左

for(int i = right - 1; i >= left; i--){

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

}

//从下到上

for(int i = boom - 1; i > top ; i--){

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

}

left++;

right--;

top++;

boom--;

}

//单独剩下一行的情况

if(top == boom && left < right){

for(int i = left; i <= right; i++){

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

}

}

//单独剩下一列的情况

if(left == right && top < boom){

for(int i = top; i <= boom; i++){

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

}

}

//单独剩下一个元素的情况

if(left == right && top == boom){

list.add(matrix[top][left]);

}

return list;

}

}

## 10亿个不重复未排序的整数，给定一个int，判断是否出现过？