数据结构与算法题解（5）： 剑指offer解题报告

剑指offer编程题java实现整理。

3. 二维数组中的查找（数组）

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

首先选取数组中右上角的数字，如果该数字等于我们要查找的数组，查找过程结束；如果该数字大于要查找的数组，剔除这个数字所在的列；如果该数字小于要查找的数组，剔除这个数字所在的行。也就是说如果要查找的数字不再数组的右上角，则每一次都在数组的查找范围中剔除一行或者一列，这样每一步都可以缩小查找的范围，直到找到要查找的数字，或者查找范围为空。

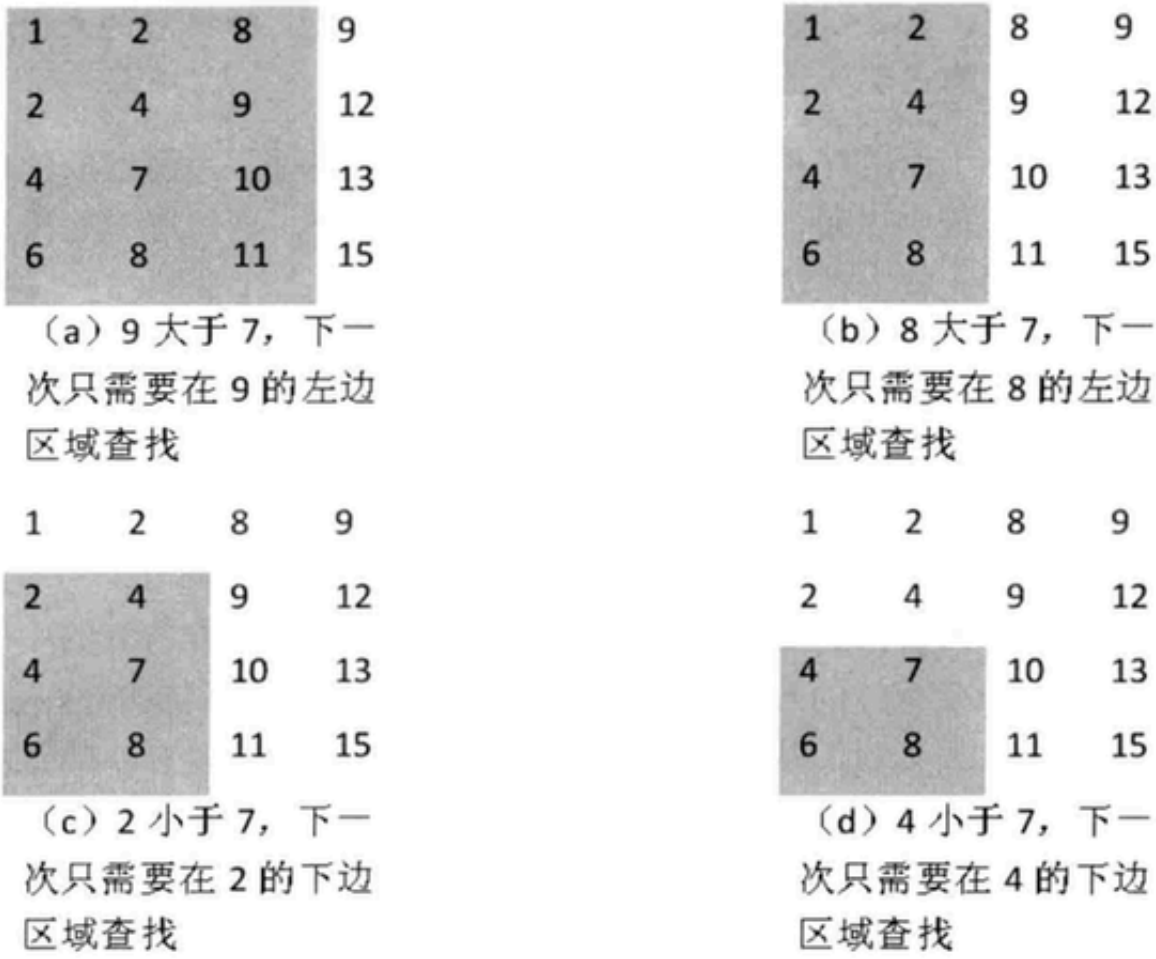


图 2.2　在二维数组中查找 7 的步骤

注：矩阵中加阴影背景的区域是下一步查找的范围。

java

```java
public class Solution {
    public boolean Find(int target, int [][] array) {
        int row = 0;
        int col = array[0].length - 1;
        while(row<=array.length-1&&col>=0){
            if(target == array[row][col]){
                return true;
            }
            else if(target>array[row][col]){
                row++;
            }
            else{
                col--;
            }
        }
        return false;
```

```
    }
}
```

> python

```
# -*- coding:utf-8 -*-
class Solution:
    # array 二维列表
    def Find(self, target, array):
        # write code here
        row = 0
        col = len(array[0])-1
        while row<=len(array)-1 and col>=0:
            if target==array[row][col]:
                return True
            elif target>array[row][col]:
                row+=1
            else:
                col-=1
        return False
```

也可以把每一行看做是一个递增的序列，利用二分查找。

> java

```
public class Solution {
    public boolean Find(int target, int [][] array) {
        for(int i=0;i<array.length;i++){
            int low =0;
            int high = array[i].length-1;
            while(low<=high){
                int mid = (low+high)/2;
                if(array[i][mid]==target)
                    return true;
                else if(array[i][mid]>target)
                    high =mid-1;
                else
                    low=mid+1;
            }

        }
        return false;
    }
}
```

4. 替换空格（字符串）

请实现一个函数，把字符串中的每个空格替换成"%20"。例如输入“We are happy.”，则输出“We%20are20happy”

网络编程中，要把特殊符号转换成服务器可识别的字符。转换的规则是在“%”后面跟上ASCII码的两位十六进制的表示。比如空格的ASCII码是32，即十六进制的0X20，因此空格被替换成“%20”。

问题1：替换字符串，是在原来的字符串上做替换，还是新开辟一个字符串做替换！
问题2：在当前字符串替换，怎么替换才更有效率（不考虑java里现有的replace方法）。从前往后替换，后面的字符要不断往后移动，要多次移动，所以效率低下；从后往前，先计算需要多少空间，然后从后往前移动，则每个字符只为移动一次，这样效率更高一点。

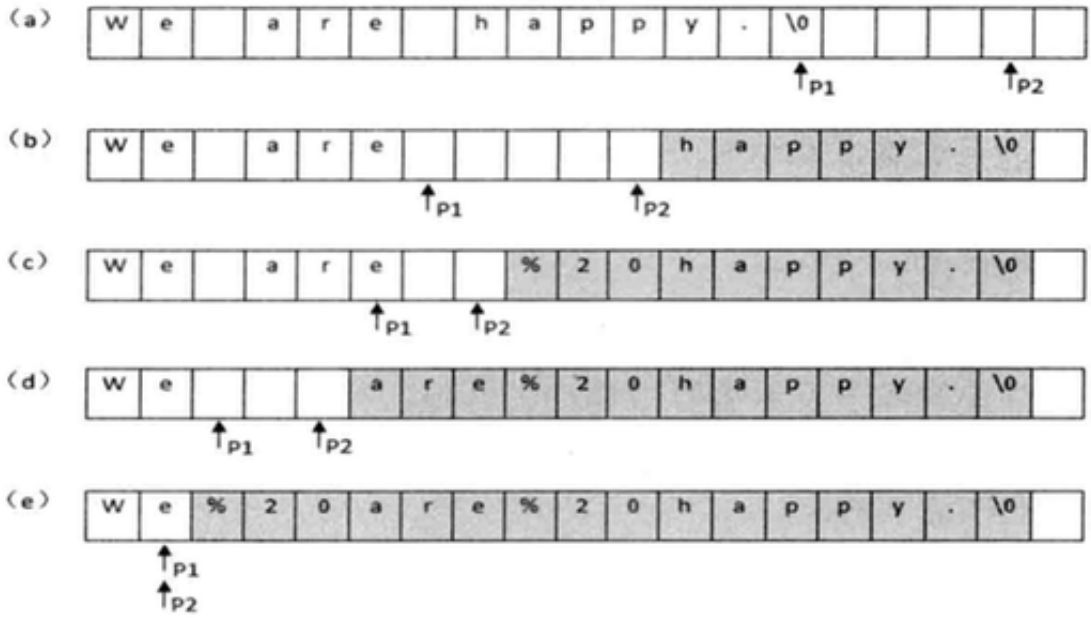


图 2.4　从后往前把字符串中的空格替换成“%20”的过程

```
public class Solution {
    public String replaceSpace(StringBuffer str) {
        int spacenum = 0;//spacenum为计算空格数
        for(int i=0;i<str.length();i++){
            if(str.charAt(i)==' ')
                spacenum++;
        }
        int indexold = str.length()-1;//indexold为为替换前的str下标
        int newlength = str.length()+2*spacenum;//计算空格转换成%20之后的str长度
        int indexnew = newlength-1;//indexold为为把空格替换为%20后的str下标
        str.setLength(newlength);//使str的长度扩大到转换成%20之后的长度,防止下标越界,setLength方法
        for(;indexold>=0&&indexold<newlength;--indexold){
```

```
            if(str.charAt(indexold)==' '){//charAt方法
                str.setCharAt(indexnew--,'0');
                str.setCharAt(indexnew--,'2');
                str.setCharAt(indexnew--,'%');
            }
            else{
                str.setCharAt(indexnew--,str.charAt(indexold));
            }
        }
        return str.toString();
    }
}
```

5. 从尾到头打印链表（链表）

输入一个链表的头结点，从尾到头反过来打印每个结点的值（注意不能改变链表的结构）。

解决这个问题肯定要遍历链表。遍历的顺序是从头到尾的顺序，可输出的顺序却是从尾到头。也就是说第一个遍历到的结点最后一个输出，而最后一个遍历到的结点第一个输出。这就是典型的“后进先出”，我们可以用栈实现这种顺序。没经过一个节点的时候，把该结点放到一个栈中。当遍历完整个链表后，再从栈顶开始逐个输出结点的值，此时输出的结点的顺序就翻转过来了。实现代码如下：

```
import java.util.Stack;
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
        Stack<Integer> stack = new Stack<>();
        while (listNode!=null){
            stack.push(listNode.val);
            listNode = listNode.next;
        }
        ArrayList<Integer> List = new ArrayList<>();
        while(!stack.isEmpty()){
            List.add(stack.pop());
        }
        return List;
    }
}
```

既然想到了用栈来实现这个函数，而递归在本质上就是一个栈结构，因此可用递归来实现。要实现反过来输出链表，我们每访问到一个节点的时候，先递归输出它后面的结点，再输出该结点自身，这样链表的输出结果就反过来了。实现代码如下：

```
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
        ArrayList<Integer> list=new ArrayList<Integer>();

        ListNode pNode=listNode;
        if(pNode!=null){
            if(pNode.next!=null){
                list=printListFromTailToHead(pNode.next);
            }
            list.add(pNode.val);
        }

        return list;
    }
}
```

6. 重建二叉树（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。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{4,7,2,1,5,3,8,6}，则重建二叉树并返回。

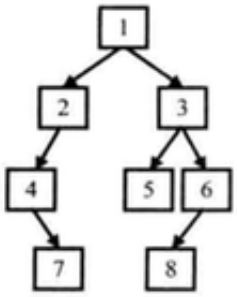


图 2.6　根据前序遍历序列{1, 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}重建的二叉树

1. 根据前序遍历的特点，我们知道根结点为1
2. 观察中序遍历。其中root节点G左侧的472必然是root的左子树，G右侧的5386必然是root的右子树。
3. 观察左子树472，左子树的中的根节点必然是大树的root的leftchild。在前序遍历中，大树的root的leftchild位于root之后，所以左子树的根节点为2。
4. 同样的道理，root的右子树节点5386中的根节点也可以通过前序遍历求得。在前序遍历中，一定是先把root和root的所有左子树节点遍历完之后才会遍历右子树，并且遍历的左子树的

第一个节点就是左子树的根节点。同理，遍历的右子树的第一个节点就是右子树的根节点。

5. 观察发现，上面的过程是递归的。先找到当前树的根节点，然后划分为左子树，右子树，然后进入左子树重复上面的过程，然后进入右子树重复上面的过程。最后就可以还原一棵树了。

该步递归的过程可以简洁表达如下：

1. 确定根,确定左子树，确定右子树。
2. 在左子树中递归。
3. 在右子树中递归。
4. 打印当前根。

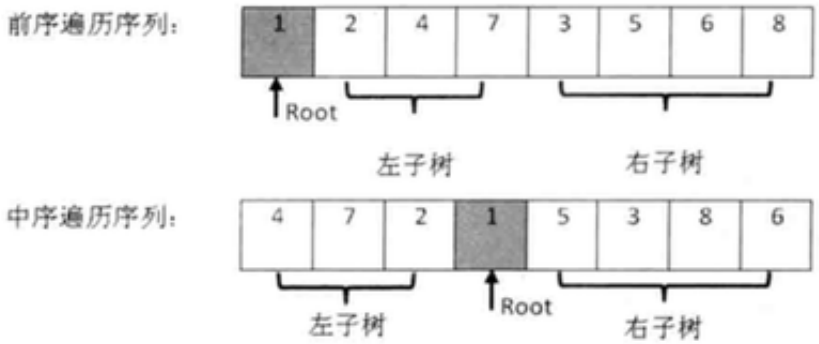


图 2.7　在二叉树的前序遍历和中序遍历的序列中确定根结点的值、左子树结点的值和右子树结点的值

递归代码如下：

```
public class Solution {
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        return reConBTree(pre,0,pre.length-1,in,0,in.length-1);
    }
    public TreeNode reConBTree(int [] pre,int preleft,int preright,int [] in,int in
left,int inright){
        if(preleft > preright || inleft> inright)//当到达边界条件时候返回null
            return null;
        //新建一个TreeNode
        TreeNode root = new TreeNode(pre[preleft]);
        //对中序数组进行输入边界的遍历
        for(int i = inleft; i<= inright; i++){
            if(pre[preleft] == in[i]){
                //重构左子树，注意边界条件
                root.left = reConBTree(pre,preleft+1,preleft+i-inleft,in,inleft,i-1
);
                //重构右子树，注意边界条件
                root.right = reConBTree(pre,preleft+i+1-inleft,preright,in,i+1,inri
```

```
ght);
                }
            }
            return root;
        }
}
```

7. 用两个栈实现队列（栈与队列）

栈是一个非常常见的数据结构，它在计算机领域中被广泛应用，比如操作系统会给每个线程创建一个栈来存储函数调用时各个函数的参数、返回地址及临时变量等。栈的特点是后进先出，即最后被压入（push）栈的元素会第一个被弹出（pop）。

队列是另外一种很重要的数据结构。和栈不同的是，队列的特点是先进先出，即第一个进入队列的元素将会第一个出来。

栈和队列虽然是针锋相对的两个数据结构，但有意思的是他们却相互联系。

通过一个具体的例子来分析往队列插入和删除元素的过程。首先插入一个元素a，不妨先把它插入到stack1，此时stack1中的元素有{a}，stack2为空，再向stack1压入b和c，此时stack1中的元素有{a,b,c}，其中c处于栈顶，而stack2仍然是空的。

因为a是最先进的，最先被删除的元素应该是a，但a位于栈低。我们可以把stack1中的元素逐个弹出并压入stack2，元素在stack2的顺序正好和原来在stack1的顺序相反因此经过三次弹出stack1和压入stack2操作之后，stack1为空，而stack2的元素是{c,b,a}，这时就可以弹出stack2的栈顶a了，随后弹出stack2中的b和c，而这个过程中stack1始终为空.

从上面的分析我们可以总结出删除一个元素的步骤：**当stack2中不为空时，在stack2的栈顶元素是最先进入队列的元素，可以弹出。如果stack2为空时，我们把stack1中的元素逐个弹出并压入stack2。由于先进入队列的元素被压到stack1的底端，经过弹出和压入之后就处于stack2的顶端了，又可以直接弹出。**

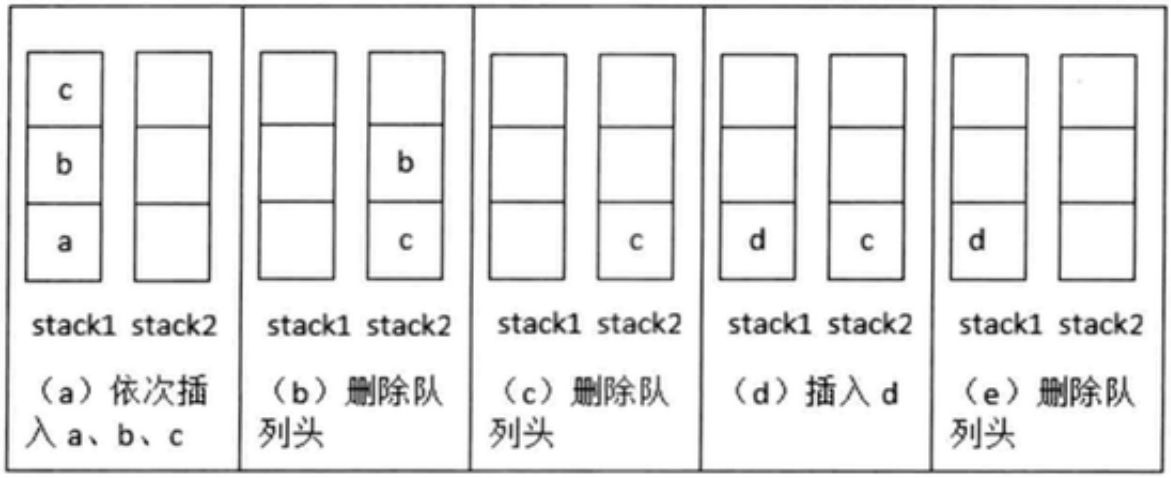


图 2.8　用两个栈模拟一个队列的操作

```
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() {
        while(!stack2.isEmpty()){
            return stack2.pop();
        }
        while(!stack1.isEmpty()){
            stack2.push(stack1.pop());
        }
        return stack2.pop();

    }
}
```

8. 旋转数组的最小数字(数组)

在准备面试的时候，我们应该重点掌握二分查找、归并排序和快速排序，做到能随时正确、完整地写出它们的代码。

若面试题是要求在排序的数组（或部分排序的数组）中查找一个数字或者统计某个数字出现的次数，我们都可以尝试用二分查找算法。

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。 输入一个非递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。

可以采用二分法解答这个问题， mid = low + (high - low)/2 ，需要考虑三种情况：

1. array[mid] > array[high]: 出现这种情况的array类似[3,4,5,6,0,1,2]，此时最小数字一定在mid的右边。 low = mid + 1
2. array[mid] == array[high]: 出现这种情况的array类似 [1,0,1,1,1] 或者[1,1,1,0,1]，此时最小数字不好判断在mid左边，还是右边,这时只好一个一个试，low = low + 1 或者 high = high - 1
3. array[mid] < array[high]: 出现这种情况的array类似[2,2,3,4,5,6,6],此时最小数字一定就是array[mid]或者在mid的左边。因为右边必然都是递增的。 high = mid。注意这里有个坑：如果待查询的范围最后只剩两个数，那么mid一定会指向下标靠前的数字，比如 array = [4,6], array[low] = 4 ;array[mid] = 4 ; array[high] = 6 ; 如果high = mid - 1，就会产生错误， 因此high = mid，但情形(1)中low = mid + 1就不会错误。

代码如下：

> java

```java
import java.util.ArrayList;
public class Solution {
    public int minNumberInRotateArray(int [] array) {
        int low = 0;
        int high = array.length-1;
        while(low<high){
            int mid = low+(high-low)/2;
            if(array[mid]>array[high]){
                low=mid+1;
            }
            else if(array[mid]==array[high]){
                high=high-1;
            }
            else{
                high = mid;
            }

        }
        return array[low];

    }
}
```

9. 斐波那契数列(数组)

9.1 斐波那契数列

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。n<=39。这个题可以说是迭代（Iteration） VS 递归（Recursion），f(n) = f(n-1) + f(n-2)，第一眼看就是递归啊，简直完美的递归环境，递归肯定很爽，这样想着关键代码两三行就搞定了，注意这题的n是从0开始的：

```
if(n<=1) return n;
else return Fibonacci(n-1)+Fibonacci(n-2);
```

然而并没有什么用，测试用例里肯定准备着一个超大的n来让Stack Overflow，为什么会溢出？因为重复计算，而且重复的情况还很严重，举个小点的例子，n=4，看看程序怎么跑的：

```
Fibonacci(4) = Fibonacci(3) + Fibonacci(2);
                    = Fibonacci(2) + Fibonacci(1) + Fibonacci(1) + Fibonacci(0);
                    = Fibonacci(1) + Fibonacci(0) + Fibonacci(1) + Fibonacci(1) + F
ibonacci(0);
```

由于我们的代码并没有记录Fibonacci(1)和Fibonacci(0)的结果，对于程序来说它每次递归都是未知的，因此光是n=4时f(1)就重复计算了3次之多。

更简单的办法是从下往上计算，首先根据f(0)和f(1)算出f(2)，再根据f(1)和f(2)算出f(3).....依此类推就可以算出第n项了。很容易理解，这种思路的时间复杂度是O(n)。实现代码如下：

> java

```
public class Solution {
    public int Fibonacci(int n) {
        if(n==0)
            return 0;
        if(n==1)
            return 1;
        int num1 = 0;
        int num2 = 1;
        int fibN=0;
        for(int i=2;i<=n;++i){
            fibN=num1+num2;
            num1=num2;
            num2=fibN;
```

```
        }
        return fibN;
    }
}
```

9.2 跳台阶

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级台阶总共有多少种跳法。

我们把n级台阶的跳法看成是n的函数，记为f(n)。当n>2时，第一次跳的时候就有两种不同的选择：一是第一次只跳1级，此时跳法数目等于后面剩下的n-1级台阶的跳法数目，即为f(n-1)；另外一种选择是第一次跳2级，此时跳法数目等于后面剩下的n-2级台阶的跳法数目，即为f(n-2)，因此n级台阶的不同跳法的总数f(n)=f(n-1)+f(n-2)。分析到这里，我们不难看出这实际上是斐波那契数列了。
代码如下：

> java

```
public class Solution {
    public int JumpFloor(int target) {
        if(target == 0)
            return 0;
        if(target == 1)
            return 1;
        if(target == 2)
            return 2;
        int num1 = 0;
        int num2 = 1;
        int jump = 0;
        for(int i=0;i<target;i++){
            jump = num1+num2;
            num1=num2;
            num2=jump;
        }
        return jump;

    }
}
```

9.3 变态跳台阶

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。

因为n级台阶，第一步有n种跳法：跳1级、跳2级、到跳n级。跳1级，剩下n-1级，则剩下跳法是f(n-1)，跳2级，剩下n-2级，则剩下跳法是f(n-2)。所以f(n)=f(n-1)+f(n-2)+...+f(1)，因为f(n-1)=f(n-2)+f(n-3)+...+f(1)，所以f(n)=2*f(n-1)

> java

```
public class Solution {
    public int JumpFloorII(int target) {
        if(target==0)
            return 0;
        if(target==1)
            return 1;
        else{
            return 2*JumpFloorII(target-1);
        }
    }
}
```

9.4 矩形覆盖

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?

把$2*8$的覆盖方法记为f(8)。用一个$1*2$小矩形去覆盖大矩形的最左边有两个选择。竖着放或者横着放。当竖着放时，右边剩下$2*7$的区域，记为f(7)。横着放时，当$1*2$的小矩阵横着放在左上角的时候，左下角必须横着放一个$1*2$的小矩阵，剩下$2*6$，记为f(6)，因此f(8)=f(7)+f(6)。此时可以看出，仍然是斐波那契数列。

代码如下:

> java

```
public class Solution {
    public int RectCover(int target) {
        if(target==0)
            return 0;
        if(target==1)
            return 1;
        int num1=0;
        int num2=1;
        int cover =0;
```

```
        for(int i=0;i<target;i++){
            cover = num1+num2;
            num1=num2;
            num2=cover;
        }
        return cover;
    }
}
```

10. 二进制中1的个数(位运算)

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。

如果一个整数不为0，那么这个整数至少有一位是1。如果我们把这个整数减1，那么原来处在整数最右边的1就会变为0，原来在1后面的所有的0都会变成1(如果最右边的1后面还有0的话)。其余所有位将不会受到影响。
举个例子：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。

> java

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

11. 数值的整数次方（位运算）

```java
public class Solution {
    public double Power(double base, int n) {
```

```
        double res = 1,curr = base;
        int exponent;
        if(n>0){
            exponent = n;
        }else if(n<0){
            if(base==0)
                throw new RuntimeException("分母不能为0");
            exponent = -n;
        }else{// n==0
            return 1;// 0的0次方
        }
        while(exponent!=0){
            if((exponent&1)==1)
                res*=curr;
            curr*=curr;// 翻倍
            exponent>>=1;// 右移一位
        }
        return n>=0?res:(1/res);
    }
}
```

12. 打印1到最大的n位数（null）

13. 在O(1)时间删除链表结点（链表）

给定单向链表的头指针和一个结点指针，定义一个函数在O(1)时间删除该结点。

我们要删除结点i，先把i的下一个结点i.next的内容复制到i，然后在把i的指针指向i.next结点的下一个结点即i.next.next，它的效果刚好是把结点i给删除了。

此外还要考虑删除的结点是头尾结点、链表中只有一个结点、链表为空这几种情况。

> java

```
public class DeleteNode {
    /**
     * 链表结点
     */
    public static class ListNode {
        int value; // 保存链表的值
        ListNode next; // 下一个结点
    }
    /**
```

```
     * 给定单向链表的头指针和一个结点指针，定义一个函数在O(1)时间删除该结点，
     * 【注意1：这个方法和文本上的不一样，书上的没有返回值，这个因为JAVA引用传递的原因，
     * 如果删除的结点是头结点，如果不采用返回值的方式，那么头结点永远删除不了】
     * 【注意2：输入的待删除结点必须是待链表中的结点，否则会引起错误，这个条件由用户进行保
证】
     *
     * @param head         链表表的头
     * @param toBeDeleted 待删除的结点
     * @return 删除后的头结点
     */
    public static ListNode deleteNode(ListNode head, ListNode toBeDeleted) {
        // 如果输入参数有空值就返回表头结点
        if (head == null || toBeDeleted == null) {
            return head;
        }
        // 如果删除的是头结点，直接返回头结点的下一个结点
        if (head == toBeDeleted) {
            return head.next;
        }
          // 下面的情况链表至少有两个结点
        // 在多个节点的情况下，如果删除的是最后一个元素
        if (toBeDeleted.next == null) {
            // 找待删除元素的前驱
            ListNode tmp = head;
            while (tmp.next != toBeDeleted) {
                tmp = tmp.next;
            }
            // 删除待结点
            tmp.next = null;
        }
        // 在多个节点的情况下，如果删除的是某个中间结点
        else {
            // 将下一个结点的值输入当前待删除的结点
            toBeDeleted.value = toBeDeleted.next.value;
            // 待删除的结点的下一个指向原先待删除引号的下下个结点，即将待删除的下一个结点
删除
            toBeDeleted.next = toBeDeleted.next.next;
        }
        // 返回删除节点后的链表头结点
        return head;
    }
```

14. 调整数组顺序使奇数位于偶数前面（排序）

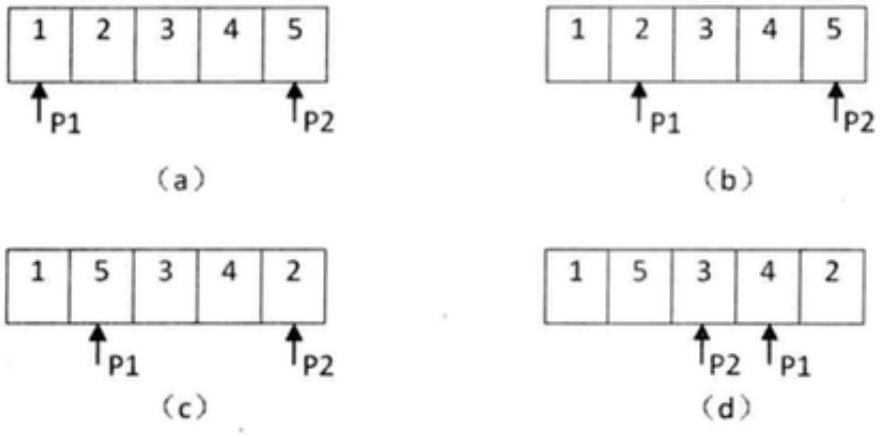


图 3.4　调整数组{1, 2, 3, 4, 5} 使得奇数位于偶数前面的过程

书上的方法类似于快排，但快排是不稳定的，即其相对位置会发生变化。

> java

```java
public class Solution {
    public void reOrderArray(int [] array) {
        int length = array.length;
        if(array==null||length==0)
            return;
        int left = 0;
        int right = length-1;
        while(left<right){
            while(left<right&&array[left]%2==1){
                left++;
            }
            while(left<right&&array[right]%2==0){
                right--;
            }
            int temp =array[right];
            array[right]=array[left];
            array[left]=temp;
        }
    }
}
```

这里要保证奇数和奇数，偶数和偶数之间的相对位置不变。可以使用插入排序的思想

> java

```java
public class Solution {
    public void reOrderArray(int [] array) {
        int length = array.length;
```

```
        if(array==null||length==0)
            return;
        for(int i=1;i<length;i++){
            if(array[i]%2==1){
                int curr = array[i];
                int j=i-1;
                while(j>=0&&array[j]%2==0){
                    array[j+1]=array[j];
                    j--;
                }
                array[j+1]=curr;
            }
        }
    }
}
```

15. 链表中倒数第K个结点（链表）

输入一个链表，输出该链表中倒数第k个结点。为了符合大多数人的习惯，本题从1 开始计数，即链表的尾结点是倒数第1个结点。例如一个链表有6个结点，从头结点开始它们的值依次是1、2、3、4、5、6。这个链表的倒数第3个结点的值为4的结点。

很自然的想法是先走到链表尾端，再从尾端回溯k步。可是我们从链表结点的定义可以看出本题中的链表是单向链表，单向链表的结点只有从前向后的指针而没有从后往前的指针，这种思路行不通。

既然不能从尾结点开始遍历链表，我们还是把思路回到头结点上来。假设整个链表有n个结点，那么倒数第k个结点就是从头结点开始往后走n-k+1步就可以了。如何得到结点树n？只需要从头开始遍历链表，每经过一个结点，计数器加1就行了。

也就是说我们需要遍历链表两次，第一次统计出链表中的结点的个数，第二次就能找到倒数第k个结点。但是面试官期待的解法是只需要遍历链表一次。

为了实现只遍历链表一次就能找到倒数第k个结点，我们可以定义两个指针。第一个指针从链表的头指针开始遍历向前走k-1步，第二个指针保持不动；从第k步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在k-1，当第一个（走在前面的）指针到达链表的尾结点时，第二个指针（走在后边的）指针正好是倒数第k个结点。

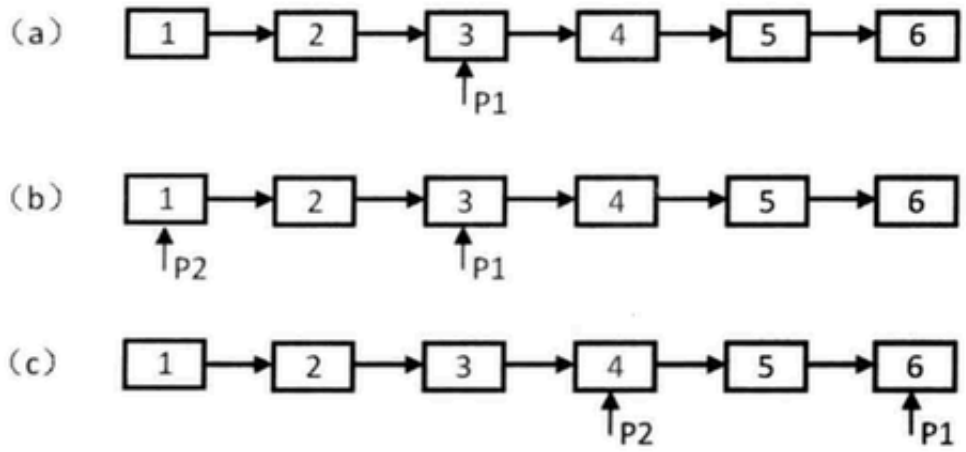


图 3.5　在有 6 个结点的链表上找倒数第 3 个结点的过程

注：（a）第一个指针在链表上走两步。（b）把第二个指针指向链表的头结点。（c）两个指针一同沿着链表向前走。当第一个指针指向链表的尾结点时，第二个指针指向倒数第 3 个结点。

但是这样写出来的代码不够鲁棒，面试官可以找出三种办法让这段代码崩溃：

- 输入的ListHead为空指针。由于代码会试图访问空指针指向的内存，程序崩溃。
- 2. 输入的以ListHead为头结点的链表的结点总数少于k。由于在for循环中会在链表上向前走k-1步，仍然会由于空指针造成的程序奔溃。
- 3. 输入的参数k为0.由于k是一个无符号整数，那么在for循环中k-1得到的将不是-1，而是4294967295（无符号的0xFFFFFFFF），因此for循环执行的次数远远超过我们的预计，同样也会造成程序崩溃。

面试过程中写代码特别要注意鲁棒性，若写出的代码存在多处崩溃的风险，那我们很可能和offer失之交臂。针对前面三个问题，分别处理。若输入的链表头指针为null，那么整个链表为空，此时查找倒数第k个结点自然应该返回null。若输入的k为0，也就是试图查找倒数第0个结点，由于我们计数是从1开始的，因此输入0是没有实际意义，也可以返回null。若链表的结点数少于k，在for循环中遍历链表可能会出现指向null的next，因此我们在for循环中应该加一个if循环。

代码如下：

> java版本

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
```

```
    }
}*/
public class Solution {
    public ListNode FindKthToTail(ListNode head,int k) {
        if(head==null||k <=0){return null;}
        ListNode pAhead = head;
        ListNode pBehind = head;

        for(int i=1;i<k;i++){
            if(pAhead.next != null)
                {pAhead = pAhead.next;}
            else
                {return null;}
        }


        while(pAhead.next!=null)
            {
            pAhead = pAhead.next;
            pBehind = pBehind.next;
        }
        return pBehind;
    }
}
```

> python版本

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    def FindKthToTail(self, head, k):
        # write code here
        if not head or k == 0:
            return None
        pAhead = head
        pBehind = None
        for i in xrange(0,k-1):
            if pAhead.next != None:
                pAhead = pAhead.next
            else:
                return None
        pBehind = head
        while pAhead.next != None:
            pAhead = pAhead.next
```

```
            pBehind = pBehind.next
        return pBehind
```

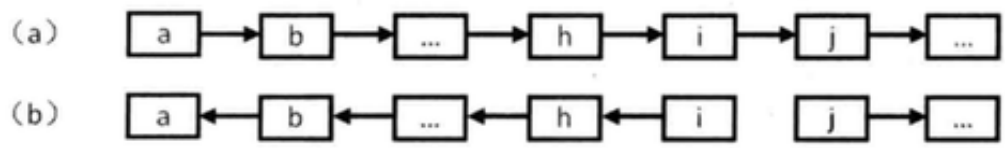
16. 反转链表（链表）

定义一个函数，输入一个链表的头结点，反转该链表并输出反转后的头结点。链表结点定义如下：

```
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}
```

解决与链表相关的问题总是有大量的指针操作，而指针操作的代码总是容易出错的。

为了正确地反转一个链表，需要调整链表中指针的方向。为了将调整指针这个复杂的过程分析清楚，可以借助图形来直观分析。在下图所示的链表中，h、i、j是3个相邻的结点。假设经过若干操作，我们已经把结点h之前的指针调整完毕，这些结点的next指向h，此时链表的结果如下所示：



其中（a）为一个链表，（b）把i之前的所有结点的next都指向前一个结点，导致链表在结点i、j之间断裂。

不难注意到，由于结点i的next指向了它的前一个结点，导致我们无法再链表中遍历到结点j。为了避免链表在结点i处断开，我们需要在调整结点i的next之前把结点j保存下来。

也就是说我们在调整结点i的next指针时，除了需要知道结点i本身之外，还需要前一个结点h，因为我们需要把结点i的next指向结点h。同时，我们还事先需要保存i的一个结点j，以防止链表断开。因此相应地我们需要定义3个指针，分别指向当前遍历到的结点、它的前一个结点及后一个结点。

最后我们试着找到反转后链表的头结点。不难分析出反转后链表的头结点是原始链表的尾结点。

什么结点是尾结点？自然是next为null的结点。

$$pre \rightarrow head \rightarrow next$$

先保存next，即$next = head.next$再反转head的指针$head.next = pre$，链表结构变成

$$pre \leftarrow head \quad next$$

接着向后移动结点$pre = head, head = next$

实现代码如下：

> java版本

```
public class Solution {
    public ListNode ReverseList(ListNode head) {

        if(head==null)
            return null;
        //head为当前节点，如果当前节点为空的话，那就什么也不做，直接返回null；
        ListNode pre = null;
        ListNode next = null;
        //当前节点是head，pre为当前节点的前一节点，next为当前节点的下一节点
        //需要pre和next的目的是让当前节点从pre->head->next1->next2变成pre<-head next1->next2
        //即pre让节点可以反转所指方向，但反转之后如果不用next节点保存next1节点的话，此单链表就此断开了
        //所以需要用到pre和next两个节点
        //1->2->3->4->5
        //1<-2<-3 4->5
        while(head!=null){
            //做循环，如果当前节点不为空的话，始终执行此循环，此循环的目的就是让当前节点从指向next到指向pre
            //如此就可以做到反转链表的效果
            //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
            next = head.next;
            //保存完next，就可以让head从指向next变成指向pre了，代码如下
            head.next = pre;
            //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
            pre = head;
            head = next;
        }
        //如果head为null的时候，pre就为最后一个节点了，但是链表已经反转完毕，pre就是反转后链表的第一个节点
```

```
        //直接输出pre就是我们想要得到的反转后的链表
        return pre;
    }
}
```

> python版本

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    # 返回ListNode
    def ReverseList(self, pHead):
        # write code here
        if not pHead or not pHead.next:
            return pHead
        pre = None
        while pHead:
            next1 = pHead.next
            pHead.next = pre
            pre = pHead
            pHead = next1
        return pre
```

17. 合并两个排序的链表（链表）

输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按照递增排序的。例如下图中的链表1和链表2，则合并之后的升序链表3如下所示：

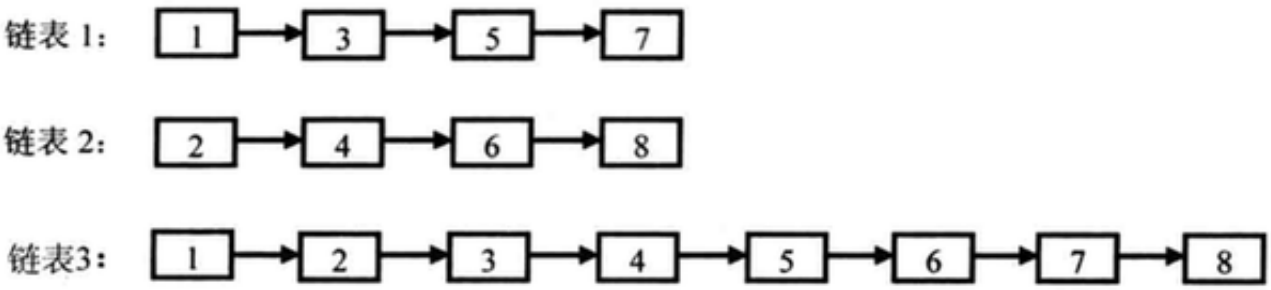


图 3.7　合并两个排序链表的过程

注：链表 1 和链表 2 是两个递增排序的链表，合并这两个链表得到升序链表为链表 3。

这是一个经常被各公司采用的面试题。在面试过程中，最容易犯两种错误：一是在写代码之前没有对合并的过程想清楚，最终合并出来的链表要么中间断开了，要么并没有做到递增排序；二是代码在鲁棒性方面存在问题，程序一旦有特殊的输入（如空链表）就会奔溃。
首先分析合并两个链表的过程。从合并两个链表的头结点开始。链表1的头结点的值小于链表2的头结点的值，因此链表1的头结点将是合并后链表的头结点。

继续合并剩余的结点。在两个链表中剩下的结点依然是排序的，因此合并这两个链表的步骤和前面的步骤是一样的。依旧比较两个头结点的值。此时链表2的头结点值小于链表1的头结点的值，因此链表2的头结点的值将是合并剩余结点得到的链表的头结点。把这个结点和前面合并链表时得到的链表的尾结点链接起来。

当我们得到两个链表中值较小的头结点并把它链接到已经合并的链表之后，两个链表剩余的结点依然是排序的，因此合并的步骤和之前的步骤是一样的。这是典型的递归过程，我们可以定义递归函数完成这一合并过程。（解决这个问题需要大量的指针操作，如没有透彻地分析问题形成清晰的思路，很难写出正确的代码）

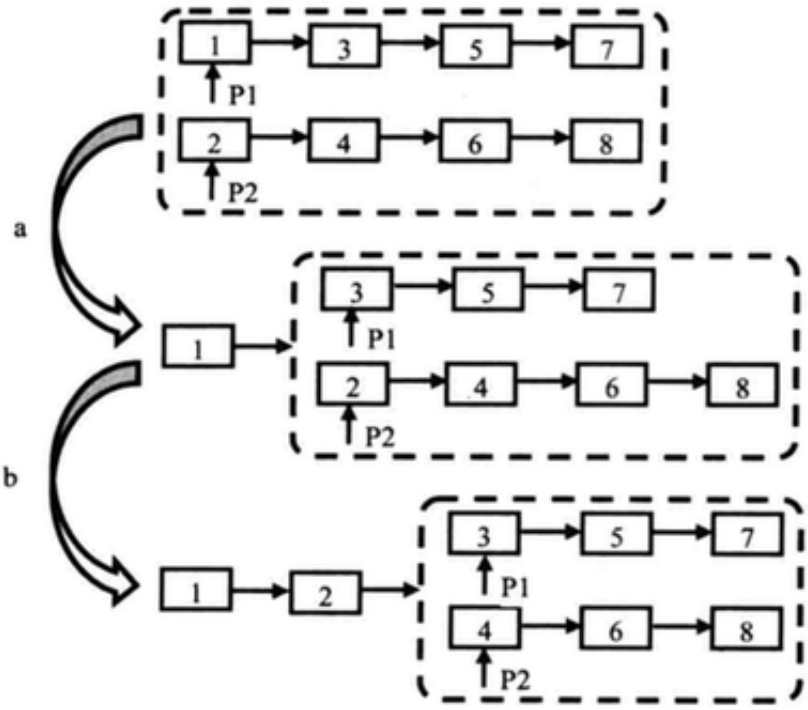


图 3.8　合并两个递增链表的过程

接下来解决鲁棒性问题，每当代码试图访问空指针指向的内存时程序就会奔溃，从而导致鲁棒性问题。本题中一旦输入空的链表就会引入空的指针，因此我们要对空链表单独处理。当第一个链表是空链表，也就是它的头结点是一个空指针时，和第二个链表合并的结果就是第二个链表。同样，当输入的第二个链表的头结点是空指针的时候，和第一个链表合并得到的结果就是第一个链表。如果两个链表都为空，合并得到的是一个空链表。（由于有大量的指针操作，如果稍有不慎就会在代码中遗留很多与鲁棒性相关的隐患。建议应聘者在写代码之前全面分析哪些情况会引入空指针，并考虑清楚怎么处理这些空指针。）

代码如下：

> java版本

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/
public class Solution {
    public ListNode Merge(ListNode list1,ListNode list2) {
        if (list1==null) return list2;
        else if (list2==null) return list1;
        ListNode MergeHead = null;

        if (listval<=list2.val){
            MergeHead = list1;
            MergeHead.next = Merge(listnext,list2);
        }
        else
            {MergeHead = list2;
             MergeHead.next = Merge(list1,list2.next);
             }
        return MergeHead;

    }
}
```

> python版本

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    # 返回合并后列表
    def Merge(self, pHead1, pHead2):
        # write code here
        if pHead1== None:
            return pHead2
        if pHead2== None:
```

```
            return pHead1
        MergeHead = None
        if pHeadval < pHead2.val:
            MergeHead = pHead1
            MergeHead.next = self.Merge(pHeadnext,pHead2)
        else:
            MergeHead = pHead2
            MergeHead.next = self.Merge(pHead1,pHead2.next)
        return MergeHead
```

18. 树的子结构（二叉树）

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）

要查找树A中是否存在和树B结构一样的子树，我们可以分成两步：第一步在树A中找到和B的根结点的值一样的结点R，第二步再判断树A以R为根结点的子树是不是包含和树B一样的结构。

第一步在树A中查找与根结点的值一样的结点，实际上就是树的遍历。对二叉树这种数据结构熟悉的读者自然知道可以用递归的方法去遍历，也可以用循环的方法去遍历。由于递归的代码实现比较简洁，面试时如果没有特别要求，通常会采用递归的方式。参考代码如下：

> java第一步

```
public boolean HasSubtree(TreeNode root1,TreeNode root2) {
        boolean result = false;
        //一定要注意边界条件的检查，即检查空指针。否则程序容易奔溃，面试时尤其要注意。这里当Tree1和Tree2都不为零的时候，才进行比较。否则直接返回false
        if(root1!=null&&root2!=null){
        ////如果找到了对应Tree2的根节点的点
            if(root1.val==root2.val){
            //以这个根节点为为起点判断是否包含Tree2
                result = DoesTree1HaveTree2(root1,root2);
            }
            //如果找不到，那么就再去root的左儿子当作起点，去判断是否包含Tree2
            if(!result){
                result=HasSubtree(root1.left,root2);
            }
            //如果还找不到，那么就再去root的右儿子当作起点，去判断是否包含Tree2
            if(!result){
                result=HasSubtree(root1.right,root2);
            }
        }
        return result;
```

```
    }
```

第二步是判断树A中以R为根结点的子树是不是和树B具有相同的结构。同样，我们也可以用递归的思路来考虑：如果结点R的值和树B的根结点不同，则以R为根结点的子树和树B一定不具有相同的结点；如果他们的值相同，则递归地判断它们各自的左右结点的值是不是相同。递归的终止条件是我们达到了树A或者树B的叶结点。

代码如下：

> java

```
public boolean DoesTree1HaveTree2(TreeNode root1,TreeNode root2){
       //如果Tree2已经遍历完了都能对应的上，返回true
        if(root2==null){
            return true;
        }
        //如果Tree2还没有遍历完，Tree1却遍历完了。返回false
        if(root1==null){
            return false;
        }
         //如果其中有一个点没有对应上，返回false
        if(root1.val!=root2.val){
            return false;
        }
        //如果根节点对应的上，那么就分别去左右子节点里面匹配
        return DoesTree1HaveTree2(root1.left,root2.left)&&DoesTree1HaveTree2(root1.
right,root2.right);
    }
```

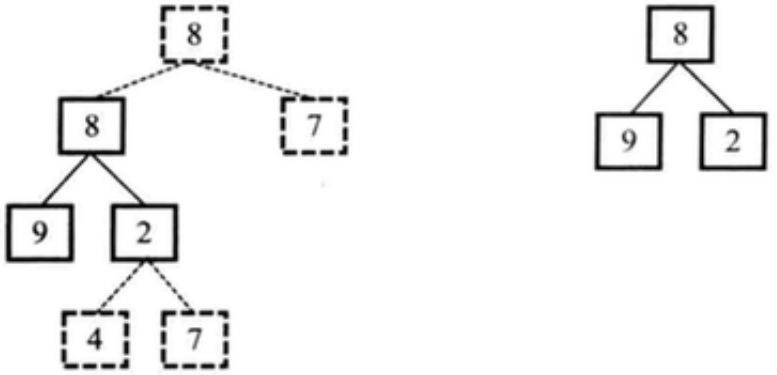


图 3.11　在树 A 中找到第二个值为 8 的结点，该结点下面（实线部分）的结构和 B 的结构一致

二叉树相关的代码有大量的指针操作，每一次使用指针的时候，我们都要问自己这个指针有没有

可能是NULL，如果是NULL该怎么处理。

19. 二叉树的镜像（二叉树）

操作给定的二叉树，将其变换为源二叉树的镜像。

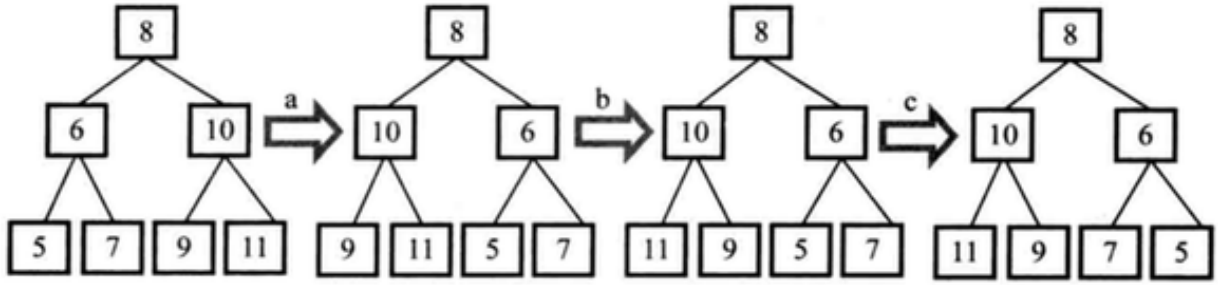


图 4.2　求二叉树镜像的过程

注：（a）交换根结点的左右子树；（b）交换值为 10 的结点的左右子结点；（c）交换值为 6 的结点的左右子结点。

```
public class Solution {
    public void Mirror(TreeNode root) {
        //边界
        if(root==null)
            return;
        if(root.left==null&&root.right==null)
            return;
        //交换左右子树
        TreeNode temp = root.left;
        root.left=root.right;
        root.right=temp;
        //递归
        if(root.left!=null){
            Mirror(root.left);
        }
        if(root.right!=null){
            Mirror(root.right);
        }

    }
}
```

20. 顺时针打印矩阵（数组）

输入一个矩阵，按照从外向里以顺时针依次打印出每一个数字。例如，输入如下矩阵：

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。

> java

```
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> printMatrix(int [][] matrix) {
        int row = matrix.length;
        int col = matrix[0].length;
        ArrayList<Integer> result = new ArrayList<Integer> ();
        // 输入的二维数组非法，返回空的数组
        if(row==0&&col==0)return result;
        // 定义四个关键变量，表示左上和右下的打印范围
        int left =0,top=0,right=col-1,bottom=row-1;
        while(left<=right&&top<=bottom){
             // left to right
            for(int i=left;i<=right;i++){result.add(matrix[top][i]);}
            // top to bottom
            for(int i=top+1;i<=bottom;i++){result.add(matrix[i][right]);}
            // right to left
            if(top!=bottom){
                for(int i=right-1;i>=left;i--){result.add(matrix[bottom][i]);}}
            // bottom to top
            if(left!=right){
                for(int i=bottom-1;i>=top+1;i--){result.add(matrix[i][left]);}}
            left++;right--;top++;bottom--;
        }
        return result;

    }

}
```

21.包含min函数的栈（栈）

定义栈的数据结构，请在该类型中实现一个能够得到栈最小元素的min函数。在该栈中，调用min、push及pop的时间复杂度都是O(1)。

可以利用一个辅助栈来存放最小值

表 4.1　栈内压入 3、4、2、1 之后接连两次弹出栈顶数字再压入 0 时，数据栈、辅助栈和最小值的状态

步骤	操作	数据栈	辅助栈	最小值
1	压入 3	3	3	3
2	压入 4	3, 4	3, 3	3
3	压入 2	3, 4, 2	3, 3, 2	2
4	压入 1	3, 4, 2, 1	3, 3, 2, 1	1
5	弹出	3, 4, 2	3, 3, 2	2
6	弹出	3, 4	3, 3	3
7	压入 0	3, 4, 0	3, 3, 0	0

每入栈一次，就与辅助栈顶比较大小，如果小就入栈，如果大就入栈当前的辅助栈顶 。
当出栈时，辅助栈也要出栈
这种做法可以保证辅助栈顶一定都是最小元素。

```
import java.util.Stack;

public class Solution {
    Stack<Integer> data = new Stack<Integer>();
    Stack<Integer> min = new Stack<Integer>();

    public void push(int node) {
        data.push(node);
        if(min.empty()){min.push(data.peek());}
        else if(data.peek()<min.peek()){min.push(data.peek());}
        else min.push(min.peek());
        }
    public void pop() {
        data.pop();
        min.pop();
    }

    public int top() {
```

```
        return data.peek();

    }
    public int min() {
        return min.peek();
    }
}
```

22. 栈的压入、弹出序列（栈）

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。（注意：这两个序列的长度是相等的）

借用一个辅助的栈，遍历压栈顺序，先讲第一个放入栈中，这里是1，然后判断栈顶元素是不是出栈顺序的第一个元素，这里是4，很显然1≠4，所以我们继续压栈，直到相等以后开始出栈，出栈一个元素，则将出栈顺序向后移动一位，直到不相等，这样循环等压栈顺序遍历完成，如果辅助栈还不为空，说明弹出序列不是该栈的弹出顺序。
举例：
入栈1,2,3,4,5
出栈4,5,3,2,1
首先1入辅助栈，此时栈顶1≠4，继续入栈2
此时栈顶2≠4，继续入栈3
此时栈顶3≠4，继续入栈4
此时栈顶4＝4，出栈4，弹出序列向后一位，此时为5，,辅助栈里面是1,2,3
此时栈顶3≠5，继续入栈5
此时栈顶5=5，出栈5,弹出序列向后一位，此时为3，,辅助栈里面是1,2,3
....
依次执行，最后辅助栈为空。如果不为空说明弹出序列不是该栈的弹出顺序。

> java

```
import java.util.ArrayList;
import java.util.Stack;
public class Solution {
    public boolean IsPopOrder(int [] pushA,int [] popA) {
        if(pushA.length==0||popA.length==0)return false;
        Stack<Integer> S=new Stack<Integer>();
        int popIndex = 0;
        for(int i=0;i<pushA.length;i++){
```

```
            S.push(pushA[i]);
            while(!S.empty()&&popA[popIndex]==S.peek()){
                S.pop();
                popIndex++;
            }
        }
        return S.empty();
    }
}
```

23. 从上往下打印二叉树（二叉树）

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。

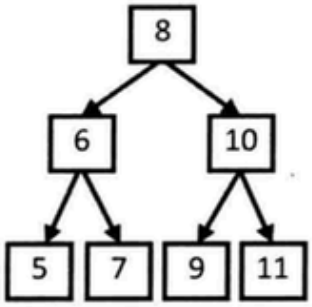


图 4.5　一棵二叉树，从上往下按层打印的顺序为 8、6、10、5、7、9、11

表 4.4　按层打印图 4.5 中的二叉树的过程

步骤	操作	队列
1	打印结点 8	结点 6、结点 10
2	打印结点 6	结点 10、结点 5、结点 7
3	打印结点 10	结点 5、结点 7、结点 9、结点 11
4	打印结点 5	结点 7、结点 9、结点 11
5	打印结点 7	结点 9、结点 11
6	打印结点 9	结点 11
7	打印结点 11	

每次打印一个结点时，如果该结点有子结点，则把该结点的子结点放到队列的末尾。接下来到队列的头部取出最早进入队列的结点，重复前面的打印操作。

> java

```
import java.util.ArrayList;
```

```
import java.util.LinkedList;
public class Solution {
    public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
        ArrayList<Integer> List=new ArrayList<Integer>();
        if(root==null){return List;}

        LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
        queue.add(root);//先把根结点加入队列q

        while(!queue.isEmpty()){//队列非空时
            TreeNode treenode=queue.remove();//取出队列头结点
            if(treenode.left!=null){queue.add(treenode.left);}//向队列加入左孩子（若有）
            if(treenode.right!=null){queue.add(treenode.right);}//向队列加入右孩子（若有）
            List.add(treenode.val);//加到打印列表中
        }
        return List;
    }
}
```

24. 二叉搜索树的后序遍历序列（二叉树）

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。

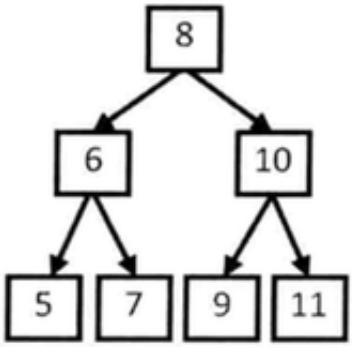


图 4.6　后序遍历序列 5、7、6、9、11、10、8 对应的二叉搜索树

在后序遍历得到的序列中，最后一个数字是树的根结点的值。数组中前面的数字可以分成两部分：第一部分是左子树结点的值，它们都比根结点小；第二部分是右子树结点的值，它们都比根结点大。

> java

```
import java.util.Arrays;
```

```
public class Solution {
    public boolean VerifySquenceOfBST(int [] sequence) {
        int length = sequence.length;
        if(sequence==null||length==0){return false;}
        int root = sequence[length-1];//根结点
        int i=0; //外部初始化
        //找到左子树的最后一个结点位置
        for(;i<length-1;i++){
            if(sequence[i]>root){
                break;
            }
        }
        //如果右子树的结点值小于根结点的值，则返回false
        for(int j=i;j<length-1;j++){
            if(sequence[j]<root){
                return false;
            }
        }
        //初始化
        boolean left=true;
        boolean right=true;
        //递归左右子树
        if(i>0){
            left = VerifySquenceOfBST(Arrays.copyOfRange(sequence,0,i));//Arrays的copyOfRange方法
        }
        if(i<length-1){
            right = VerifySquenceOfBST(Arrays.copyOfRange(sequence,i,length-1));
        }
        return left&&right;
    }
}
```

25. 二叉树中和为某一值的路径（二叉树）

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。

> java

```
public class Solution {
    private ArrayList<ArrayList<Integer>> listAll = new ArrayList<ArrayList<Integer>>();
    private ArrayList<Integer> list = new ArrayList<Integer>();
    public ArrayList<ArrayList<Integer>> FindPath(TreeNode root,int target) {
```

```
        if(root == null) return listAll;
        list.add(root.val);
        target -= root.val;//每次减去结点的值
        //如果target等于0，则说明这条路径和为target，添加到listAll中
        if(target == 0 && root.left == null && root.right == null)
            listAll.add(new ArrayList<Integer>(list));//因为add添加的是引用，如果不new一个的话，后面的操作会更改listAll中list的值
        //向左孩子递归
        if(root.left!=null)FindPath(root.left, target);
        //向右孩子递归
        if(root.right!=null)FindPath(root.right, target);
        //如果不满足条件，则回到父节点；
        list.remove(list.size()-1);
        return listAll;
    }
}
```

26. 复杂链表的复制（链表）

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）

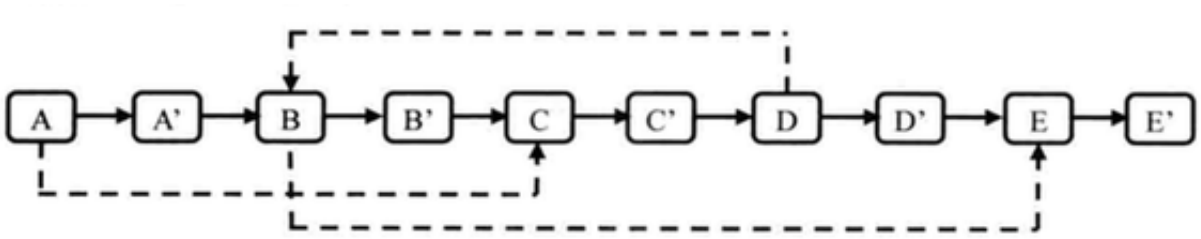


图 4.9　复制复杂链表的第一步

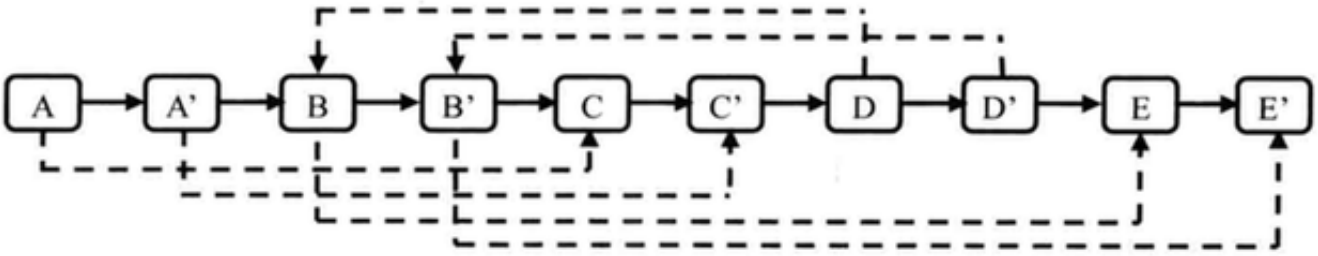


图 4.10　复制复杂链表的第二步

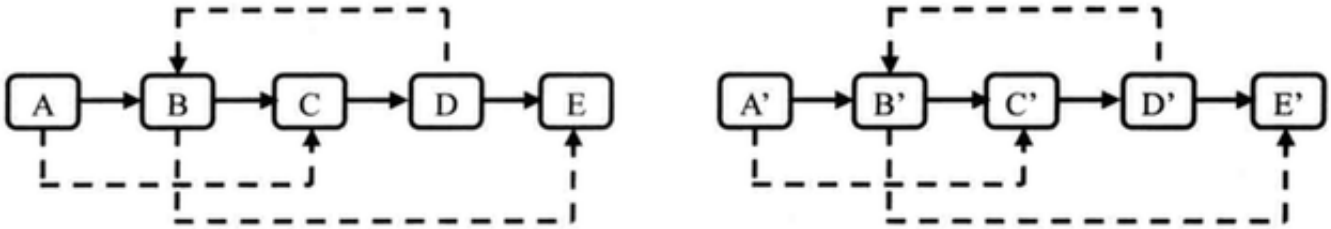


图 4.11　复制复杂链表的第三步

java

```
public class Solution {
    public RandomListNode Clone(RandomListNode pHead)
    {
        if (pHead == null) return null;
         //复制next 如原来是A->B->C 变成A->A'->B->B'->C->C'
        RandomListNode pCur = pHead;
        while (pCur != null)
        {
            RandomListNode node = new RandomListNode(pCur.label);
            node.next = pCur.next;
            pCur.next = node;
            pCur = node.next;
        }
         //复制random pCur是原来链表的结点 pCur.next是复制pCur的结点
        pCur = pHead;
        while (pCur!=null)
        {
            if (pCur.random!=null)
                pCur.next.random = pCur.random.next;
            pCur = pCur.next.next;
        }
         //拆分链表
        RandomListNode head = pHead.next;
        RandomListNode tmp = head;
        pCur = pHead;
        while(pCur.next!=null)
        {
            tmp = pCur.next;
            pCur.next = tmp.next;
            pCur = tmp;
        }
        return head;
    }
}
```

27. 二叉搜索树与双向链表（二叉树）

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。

> java

```
public class Solution {
    TreeNode head = null;
    TreeNode realHead = null;
    public TreeNode Convert(TreeNode pRootOfTree) {
        ConvertSub(pRootOfTree);
        return realHead//realHead是每个子树排序后的第一个结点，head是排序后的最后一个结点;
    }

    private void ConvertSub(TreeNode pRootOfTree) {
        //递归中序遍历
        if(pRootOfTree==null) return;
        ConvertSub(pRootOfTree.left);
        if (head == null) {
            //初始处
            head = pRootOfTree;
            realHead = pRootOfTree;
        } else {
            //前两句实现双向，第三句跳到下一个节点。
            head.right = pRootOfTree;
            pRootOfTree.left = head;
            head = pRootOfTree;
        }
        ConvertSub(pRootOfTree.right);
    }
}
```

28. 字符串的排列（字符串）

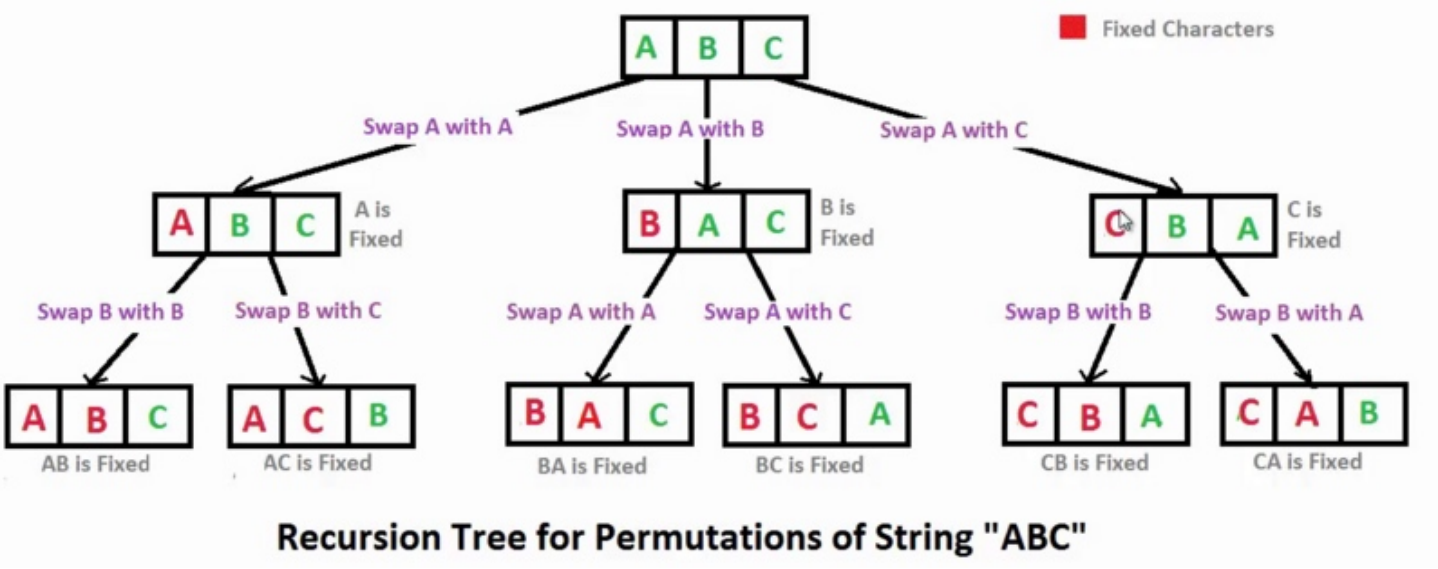
输入一个字符串,按字典序打印出该字符串中字符的所有排列。例如输入字符串abc,则打印出由字符a,b,c所能排列出来的所有字符串abc,acb,bac,bca,cab和cba。

首先我要打印abc的全排列，就是第一步把a 和bc交换（得到bac,cab），这需要一个for循环，循环里面有一个swap，交换之后就相当于不管第一步了，进入下一步递归，所以跟一个递归函数， 完成递归之后把交换的换回来，变成原来的字串

abc 为例子：
1. 固定a，求后面**bc的全排列：** abc，acb。 求完后，a 和 **b交换；** 得到**bac,开始第二轮**
2. 固定**b，**求后面ac的全排列： **bac，bca。** 求完后，**b** 和 c交换； 得到cab,开始第三轮
3. 固定c，求后面**ba的全排列：** cab，cba

即递归树：

```
        str:     a            b            c
               ab ac        ba bc         ca cb
      result:     abc acb      bac bca         cab cba
```



> java

```java
import java.util.ArrayList;
import java.util.*;
public class Solution {
    public ArrayList<String> Permutation(String str) {
        ArrayList<String> list = new ArrayList<>();
        if(str.length()==0)
            return list;
        char[] array = str.toCharArray();
        permutation(array,0,list);
        Collections.sort(list);
        return list;
    }
    public void permutation(char[] array,int begin,ArrayList<String> list) {
        if(begin == array.length-1) {
            list.add(String.valueOf(array));
        }else {
            for(int i=begin;i<array.length;++i) {
                if(i==begin || array[i]!=array[begin]) {
                    swap(array,begin,i);
                    permutation(array,begin+1,list);
```

```
                    swap(array,begin,i);
                }
            }
        }
    }
    public void swap(char[] array,int i,int j) {
        char temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

29. 数组中出现次数超过一半的数字（数组）

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。

数组中有一个数字出现的次数超过数组长度的一半，也就是说它出现的次数比其他所有数字出现的次数的和还要多。因此我们可以考虑在遍历数组的时候保存两个值：一个是数组的一个数字，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；如果不同，则次数减1；如果次数为0，则保存下一个数字，并把次数设为1。

还要判断这个数字是否超过数组长度的一半，如果不存在输出0。

```
public class Solution {
    public int MoreThanHalfNum_Solution(int [] array) {
        if(array==null||array.length==0){
            return 0;
        }

        int result=array[0];
        int count=1;
        for(int i=1;i<array.length;i++){
           if(result==array[i]){
               count++; }
           else if(result!=array[i]){
               count--; }
           if(count==0){
               result=array[i];
               count=1;
           }
        }
        int times=0;
```

```
        for(int i=0;i<array.length;i++){
            if(array[i]==result){
                times++;
            }
        }
        if(times*2<=array.length){
            System.out.println(times);
            return 0;
        }
        else
            return result;

    }
}
```

30. 最小的K个数（数组）

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,。

第一种方法，借用partition函数

> java

```
import java.util.ArrayList;
public class Solution {
    public ArrayList<Integer> GetLeastNumbers_Solution(int[] input, int k) {
        ArrayList<Integer> output = new ArrayList<Integer>();
        int length = input.length;
        if (input == null || length <= 0 || length < k || k<= 0) {
            return output;
        }
        int left = 0;
        int right = length - 1;
        int index = partition(input,left,right);
        while(index != k -1) {
            if(index < k - 1) {
                left = index + 1; //不够的话往右边走走
                index = partition(input,left,right);
            }
            else {
                right = index - 1; //太多的话往左边走走
                index = partition(input,left,right);
            }
        }
```

```
        for (int i = 0;i < k;i++) {
            output.add(input[i]);
        }
        return (ArrayList<Integer>) output;
    }
   //基准左右分区
    private int partition(int[] input,int left,int right) {
        int pivot = input[left];
        while(left < right) {
            while(input[right] >= pivot && left < right)
            {
                right--;
            }
            input[left] = input[right];
            while(input[left] <= pivot && left <right) {
                left++;
            }
            input[right] = input[left];
        }
        input[left] = pivot;
        return left;
    }
}
```

第二种方法

用最大堆保存这k个数，每次只和堆顶比，如果比堆顶小，删除堆顶，新数入堆。

> java

```
import java.util.ArrayList;
import java.util.PriorityQueue;
import java.util.Comparator;
public class Solution {
   public ArrayList<Integer> GetLeastNumbers_Solution(int[] input, int k) {
       ArrayList<Integer> result = new ArrayList<Integer>();
       int length = input.length;
       if(k > length || k == 0){
           return result;
       }
        PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(k, new Comparat
or<Integer>() {

            @Override//PriorityQueue默认是小顶堆，实现大顶堆，需要反转默认排序器
            public int compare(Integer o1, Integer o2) {
                return o2.compareTo(o1);
            }
```

```
        });
        for (int i = 0; i < length; i++) {
            //如果最大堆中已有的数字少于k个，直接读入
            if (maxHeap.size() != k) {
                maxHeap.offer(input[i]);
            }
            //如果最大堆中已有k个数字了，即容器已满，且大顶堆顶大于待插入数字，将待插入数字替换进大顶堆
            else if (maxHeap.peek() > input[i]) {
                Integer temp = maxHeap.poll();
                temp = null;
                maxHeap.offer(input[i]);
            }
        }
       //输出大顶堆中的数
        for (Integer integer : maxHeap) {
            result.add(integer);
        }
        return result;
    }
}
```

31. 连续子数组的最大和（数组）

输入一个整型数组，数组中有正数也有负数。数组中一个或连续的多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为O(n)

第一种方法

> java

```
public class Solution {
    public int FindGreatestSumOfSubArray(int[] array) {
        if(array.length==0||array==null)
            return 0;
        int cSum = 0;
        int result = array[0];// result存储最大和，不能初始为0，存在负数
        for(int i=0;i<array.length;i++){
            if(cSum<0){
                cSum=array[i];// 当前和<0，抛弃不要
            }else{
                cSum += array[i];//否则累加上去
            }
            if(cSum>result){
                result = cSum;// 存储当前的最大结果
```

```
            }
        }
        return result;
    }
}
```

第二种方法：动态规划

```
F（i）：以array[i]为末尾元素的子数组的和的最大值，子数组的元素的相对位置不变
     F（i）=max（F（i-1）+array[i] ，  array[i]）
     res：所有子数组的和的最大值
     res=max（res，F（i））

     如数组[6, -3, -2, 7, -15, 1, 2, 2]
     初始状态：
         F（0）=6
         res=6
     i=1：
         F（1）=max（F（0）-3, -3）=max（6-3, 3）=3
         res=max（F（1），res）=max（3, 6）=6
     i=2：
         F（2）=max（F（1）-2, -2）=max（3-2, -2）=1
         res=max（F（2），res）=max（1, 6）=6
     i=3：
         F（3）=max（F（2）+7, 7）=max（1+7, 7）=8
         res=max（F（2），res）=max（8, 6）=8
     i=4：
         F（4）=max（F（3）-15, -15）=max（8-15, -15）=-7
         res=max（F（4），res）=max（-7, 8）=8
     以此类推
     最终res的值为8
```

> java

```
public class Solution {
    public int FindGreatestSumOfSubArray(int[] array) {
        int res = array[0];
        int max = array[0];
        for(int i=1;i<array.length;i++){
            max=Math.max(max+array[i],array[i]);
            res = Math.max(max,res);
        }
        return res;
    }
}
```

32.从1到n整数中1出现的次数（数组）

输入一个整数n，求1到n这n个整数的十进制表示中1出现的次数。例如输入12，从1到12这些整数中包含1的数字有1、10、11、12，1一共出现了5次。

一、1的数目

编程之美上给出的规律：

1. 如果第i位（自右至左，从1开始标号）上的数字为0，则第i位可能出现1的次数由更高位决定（若没有高位，视高位为0），等于更高位数字X当前位数的权重10^{i-1}。
2. 如果第i位上的数字为1，则第i位上可能出现1的次数不仅受更高位影响，还受低位影响（若没有低位，视低位为0），等于更高位数字X当前位数的权重10^{i-1}+（低位数字+1）。
3. 如果第i位上的数字大于1，则第i位上可能出现1的次数仅由更高位决定（若没有高位，视高位为0），等于（更高位数字+1）X当前位数的权重10^{i-1}。

二、X的数目
这里的 X∈[1,9]，因为 X=0 不符合下列规律，需要单独计算。
首先要知道以下的规律：

- 从 1 至 10，在它们的个位数中，任意的 X 都出现了 1 次。
- 从 1 至 100，在它们的十位数中，任意的 X 都出现了 10 次。
- 从 1 至 1000，在它们的百位数中，任意的 X 都出现了 100 次。

依此类推，从 1 至 10^i，在它们的左数第二位（右数第 i 位）中，任意的 X 都出现了 10^{i-1} 次。

这个规律很容易验证，这里不再多做说明。

接下来以 n=2593,X=5 为例来解释如何得到数学公式。从 1 至 2593 中，数字 5 总计出现了 813 次，其中有 259 次出现在个位，260 次出现在十位，294 次出现在百位，0 次出现在千位。

现在依次分析这些数据，首先是个位。从 1 至 2590 中，包含了 259 个 10，因此任意的 X 都出现了 259 次。最后剩余的三个数 2591, 2592 和 2593，因为它们最大的个位数字 3 < X，因此不会包含任何 5。（也可以这么看，3<X，则个位上可能出现的X的次数仅由更高位决定，等于更高位数字（259）$\times 10^{1-1} = 259$）。

然后是十位。从 1 至 2500 中，包含了 25 个 100，因此任意的 X 都出现了 25×10=250 次。剩下的数字是从 2501 至 2593，它们最大的十位数字9>X，因此会包含全部10个5。最后总计250 + 10 = 260。（也可以这么看，9>X，则十位上可能出现的X的次数仅由更高位决定，等于更高位数字（$25 + 1$）$\times 10^{2-1} = 260$）。

接下来是百位。从 1 至 2000 中，包含了 2 个 1000，因此任意的 X 都出现了 2×100=200 次。剩下的数字是从 2001 至 2593，它们最大的百位数字 5 == X，这时情况就略微复杂，它们的百位肯定是包含 5 的，但不会包含全部 100 个。如果把百位是 5 的数字列出来，是从 2500 至 2593，数字的个数与百位和十位数字相关，是 93+1 = 94。最后总计 200 + 94 = 294。（也可以这么看，5==X，则百位上可能出现X的次数不仅受更高位影响，还受低位影响，等于更高位数字 $（2）\times 10^{3-1}+（93+1）=294$）。

最后是千位。现在已经没有更高位，因此直接看最大的千位数字 2 < X，所以不会包含任何 5。（也可以这么看，2<X，则千位上可能出现的X的次数仅由更高位决定，等于更高位数字 $（0）\times 10^{4-1}=0$）。

到此为止，已经计算出全部数字 5 的出现次数。
总结一下以上的算法，可以看到，当计算右数第 i 位包含的 X 的个数时：

- 取第 i 位左边（高位）的数字，乘以10^{i-1} ，得到基础值a 。
- 取第 i 位数字，计算修正值：
  - 如果大于 X，则结果为 $a+10^{i-1}$ 。
  - 如果小于 X，则结果为 a 。
  - 如果等 X，则取第 i 位右边（低位）数字，设为 b ，最后结果为 a+b+1 。

相应的代码非常简单，效率也非常高，时间复杂度只有 $O(log_{10}n)$。

代码如下：

```
public int NumberOfXBetween1AndN_Solution(int n,int x) {
    if(n<0||x<1||x>9)
        return 0;
    int high,low,curr,tmp,i = 1;
    high = n;
    int total = 0;
    while(high!=0){
        high = n/(int)Math.pow(10, i);// 获取第i位的高位
        tmp = n%(int)Math.pow(10, i);
        curr = tmp/(int)Math.pow(10, i-1);// 获取第i位
        low = tmp%(int)Math.pow(10, i-1);// 获取第i位的低位
        if(curr==x){
            total+= high*(int)Math.pow(10, i-1)+low+1;
        }else if(curr<x){
            total+=high*(int)Math.pow(10, i-1);
        }else{
            total+=(high+1)*(int)Math.pow(10, i-1);
        }
        i++;
    }
```

```
    return total;
}
```

33. 把数组排成最小的数(数组)

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。

> java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
public class Solution {
    public String PrintMinNumber(int [] numbers) {
        int n;
        String s="";
        ArrayList<Integer> list=new ArrayList<Integer>();
        n=numbers.length;
        for(int i=0;i<n;i++){
            list.add(numbers[i]);//将数组放入arrayList中
        }
        //实现了Comparator接口的compare方法，将集合元素按照compare方法的规则进行排序
        Collections.sort(list,new Comparator<Integer>(){
            @Override
            public int compare(Integer str1, Integer str2) {
                // TODO Auto-generated method stub
                    String s1=str1+""+str2;
                    String s2=str2+""+str1;

                    return s1.compareTo(s2);
                }
            });

        for(int j:list){
            s+=j;
        }
        return s;
    }
}
```

34. 丑数（数组）

把只包含因子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。

> java

```
import java.util.*;
public class Solution {
    public int GetUglyNumber_Solution(int index) {
        if(index<7)return index;
        int[] res = new int[index];
        res[0] = 1;
        int t2 = 0, t3 = 0, t5 = 0, i;
        for(i=1;i<index;i++){
            res[i] = min(res[t2]*2,min(res[t3]*3,res[t5]*5));
            if(res[i] == res[t2]*2)t2++;
            if(res[i] == res[t3]*3)t3++;
            if(res[i] == res[t5]*5)t5++;
        }
        return res[index-1];
    }
    private int min(int a,int b){
        return (a>b)? b:a;
    }
}
```

35. 第一次只出现一次的字符（字符串）

在一个字符串(1<=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.

我们可以使用一个容器来存放每个字符的出现次数。在这个数据容器中可以根据字符来查找出现的次数，也就是这个容器的作用是把一个字符映射成一个数字。在常用的数据容器中，哈希表正是这个用途。

为了解决这个问题，我们可以定义哈希表的键值（Key）是字符，而值（Value）是该字符出现的次数。同时我们还需要从头开始扫描字符串两次。第一次扫面字符串时，每扫到一个字符就在哈希表的对应项把次数加1.接下来第二次扫描时，每扫描到一个字符就能在哈希表中得到该字符出现的次数，这样第一个只出现一次的字符就是符合要求的输出。

需要涉及到Java中HashMap工作原理及实现，资料链接

> java

```
import java.util.HashMap;
public class Solution {
    public int FirstNotRepeatingChar(String str) {
        HashMap<Character,Integer> map = new HashMap<Character,Integer>();
        for(int i=0;i<str.length();i++){
            char c = str.charAt(i);//charAt方法，获得位置i的串
                if(map.containsKey(c)){//HashMap的containKey方法；
                    int time = map.get(c);//HashMap的get方法，得到Key c的Value；
                    time++;
                    map.put(c,time);//HashMap的put方法，将Key c的Value置为time；
                }else{
                    map.put(c,1);
                }
        }
        for(int i=0;i<str.length();i++){
            char c = str.charAt(i);
            if(map.get(c)==1){
                return i;
            }
        }
        return -1;
    }
}
```

36. 数组中的逆序对（数组）

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。例如在数组{7，5，6，4}中，一共存在5个逆序对，分别是（7，6）、（7，5）、（7，4）、（5，4）和（6，4）。

可以按照归并排序的思路，先把数组分隔成子数组，先统计出子数组内部的逆序对的数目，然后再统计出两个相邻子数组之间的逆序对的数目。在统计逆序对的过程中，还需要对数组进行排序。

```
public class Solution {
    int cnt;

    public int InversePairs(int[] array) {
        cnt = 0;
        if (array != null)
            mergeSortUp2Down(array, 0, array.length - 1);
        return cnt;
    }
    /*
```

```
     * 归并排序(从上往下)
     */
    public void mergeSortUp2Down(int[] a, int start, int end) {
        if (start >= end)
            return;
        int mid = (start + end) >> 1;

        mergeSortUp2Down(a, start, mid);
        mergeSortUp2Down(a, mid + 1, end);

        merge(a, start, mid, end);
    }
    /*
     * 将一个数组中的两个相邻有序区间合并成一个
     */
    public void merge(int[] a, int start, int mid, int end) {
        int[] tmp = new int[end - start + 1];

        int i = start, j = mid + 1, k = 0;
        while (i <= mid && j <= end) {
            if (a[i] <= a[j])
                tmp[k++] = a[i++];
            else {
                tmp[k++] = a[j++];
                cnt += mid - i + 1;  //关键的一步，统计逆序对.....
                cnt%=1000000007;
            }
        }
        while (i <= mid)
            tmp[k++] = a[i++];
        while (j <= end)
            tmp[k++] = a[j++];
        for (k = 0; k < tmp.length; k++)
            a[start + k] = tmp[k];
    }
}
```

37. 两个链表的第一个公共结点（链表）

输入两个链表找出他们的第一个公共结点。

面试的时候碰到这道题，很多应聘者的第一个想法就是蛮力法：在第一个链表上顺序遍历每个结点，每遍历到一个结点的时候，在第二个链表上顺序遍历每个结点。若第二个链表上有一个结点和第一个链表上的结点一样，说明两个链表在这个结点上重合，于是就找到了它们的公共结点。如果第一个链表的长度为m，第二个链表的长度为n，显然该方法的时间复杂度是O(mn)。

通常蛮力法不会是最好的办法，我们接下来试着分析有公共结点的两个链表有哪些特点。从链表结构的定义看出，这两个链表是单向链表。如果他们有公共的结点，那么这两个链表从某一结点开始，他们的next指向同一个结点。但由于是单向链表的结点，每个结点只有一个next，因此从第一个公共结点开始，之后的结点都是重合的，不可能再出现分叉。所以两个有公共结点而部分重合的链表，拓扑形状看起来像一个Y，而不是X。

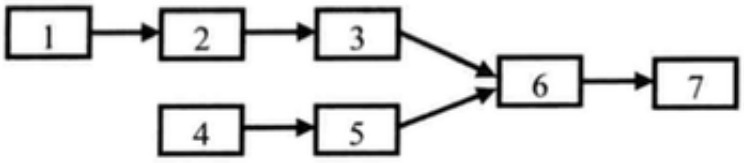


图 5.3　两个链表在值为 6 的结点处交汇

经过我们的分析发现，若两个链表有公共结点，那么公共结点出现在两个链表的尾部。如果我们从两个链表的尾部开始往前比较，最后一个相同的结点就是我们要找的结点。我们想到用栈的特点来解决这个问题：分别把两个链表的结点放入两个栈中，这样两个链表的尾结点就位于两个栈的栈顶，接下来比较两个栈顶的结点是否相同。若果相同，则把栈顶弹出接着比较下一个栈顶，直到找到最后一个相同的结点。

上面需要用到两个辅助栈。若链表的长度分别为m和n，那么空间复杂度是O(m+n)。这种思路的时间复杂度也是O(m+n)。和最开始的蛮力法相比，时间效率得到了提升，相当于是用空间换取时间效率。

之所以需要用到栈，是因为我们想同时遍历到达两个栈的尾结点。当两个链表的长度不相同时，如果我们从头开始遍历到达尾结点的时间就不一致。其实解决这个问题还有一个更简单的办法：首先遍历两个链表得到他们的长度，就能知道哪个链表比较长，以及长的链表比短的链表多几个结点。在第二次遍历的时候，在较长的链表上先走若干步，接着再同时在两个链表上遍历，找到的第一个相同的结点就是他们的第一个公共结点。

第三种思路和第二种思路相比，时间复杂度都是O(m+n)，但我们不再需要辅助的栈，因此提高了空间效率。实现代码如下：

> java版本

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/
public class Solution {
```

```
    public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {
        ListNode current1 = pHead1;//链表1
        ListNode current2 = pHead2;//链表2
        if(pHead1 ==null||pHead2==null){return null;}//

        int len1 = getlistlength(pHead1);//链表1的长度
        int len2 = getlistlength(pHead2);//链表2的长度

        //若链表1长度大于链表2
        if(len1>=len2){
            int len=len1-len2;
            //遍历链表1，遍历长度为两链表长度差
            while (len>0){
                current1 = currentnext;
                len--;

            }
        }
        //若链表2长度大于链表1
        else if(len1<len2){
            int len=len2-len1;
            //遍历链表2，遍历长度为两链表长度差
            while (len>0){
                current2=current2.next;
                len--;
            }
        }
        //开始齐头并进，直到找到第一个公共结点
        while(current1!=current2){
            current1 = currentnext;
            current2 = current2.next;
        }
        return current1;
    }
    //求指定链表的长度
    public static int getlistlength(ListNode pHead){
        int length = 0;
        ListNode current = pHead;
        while(current!=null){
            length++;
            current = current.next;
        }
        return length;
    }
}
```

> python版本

```python
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    def FindFirstCommonNode(self, pHead1, pHead2):
        # write code here
        current1=pHead1
        current2=pHead2
        len1 = self.getlistlength(current1)
        len2 = self.getlistlength(current2)
        if len1>=len2:
            length = len1-len2
            while length>0:
                current1 = currentnext
                length=length-1
        elif len1<len2:
            length = len2-len1
            while length>0:
                current2 = current2.next
                length=length-1
        while current1!=current2:
            current1=currentnext
            current2=current2.next
        return current1

    def getlistlength(self,pHead):
        length =0
        current =pHead
        while current!=None:
            length=length+1
            current = current.next
        return length
```

38. 数字在排序数组中出现的次数（数组）

统计一个数字在排序数组中出现的次数。

利用二分查找直接找到第一个K和最后一个K。以下代码使用递归方法找到第一个K，使用循环方法最后一个K。

> java

```java
public class Solution {
    public int GetNumberOfK(int [] array , int k) {
        int length = array.length;
        if(length == 0){
            return 0;
        }
        int firstK = getFirstK(array, k, 0, length-1);
        int lastK = getLastK(array, k, 0, length-1);
        if(firstK != -1 && lastK != -1){
             return lastK - firstK + 1;
        }
        return 0;
    }
    //递归写法
    private int getFirstK(int [] array , int k, int start, int end){
        if(start > end){
            return -1;
        }
        int mid = (start + end) >> 1;
        if(array[mid] > k){
            return getFirstK(array, k, start, mid-1);
        }
        else if (array[mid] < k){
            return getFirstK(array, k, mid+1, end);
        }
        else if(mid-1 >=0 && array[mid-1] == k){
            return getFirstK(array, k, start, mid-1);
        }
        else{
            return mid;
        }
    }
    //循环写法
    private int getLastK(int [] array , int k, int start, int end){
        int length = array.length;
        int mid = (start + end) >> 1;
        while(start <= end){
            if(array[mid] > k){
                end = mid-1;
                }
            else if(array[mid] < k){
                start = mid+1;
                }
            else if(mid+1 <= length-1 && array[mid+1] == k){
                start = mid+1;
                }

            else{
```

```
                return mid;
                }
            mid = (start + end) >> 1;
            }
        return -1;
     }
}
```

39. 二叉树的深度（二叉树）

39.1 二叉树的深度

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。

Java 经典的求二叉树深度 递归写法

> java

```
public class Solution {
    public int TreeDepth(TreeNode root) {
        if(root==null)return 0;
        int nleft = TreeDepth(root.left);
        int nright = TreeDepth(root.right);
        return nleft>nright?(nleft+1):(nright+1);
    }
}
```

39.2 平衡二叉树

输入一棵二叉树，判断该二叉树是否是平衡二叉树。

有了求二叉树的深度的经验之后，我们就很容易想到一个思路：在遍历树的每个结点的时候，调用函数TreeDepth得到它的左右子树的深度。如果每个结点的左右子树的深度相差都不超过1，按照定义它就是一颗平衡的二叉树。

> java

```
public class Solution {
    public boolean IsBalanced_Solution(TreeNode root) {
        if(root==null)return true;
```

```
        int left = TreeDepth(root.left);
        int right = TreeDepth(root.right);
        int diff = left-right;
        if(diff>1||diff<-1)
            return false;
        return IsBalanced_Solution(root.left)&&IsBalanced_Solution(root.right);

    }
    public int TreeDepth(TreeNode root) {
        if(root==null)return 0;
        int nleft = TreeDepth(root.left);
        int nright = TreeDepth(root.right);
        return nleft>nright?(nleft+1):(nright+1);
    }
}
```

40. 数组中只出现一次的数字（数组）

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是$O(n)$，空间复杂度是$O(1)$。

首先我们考虑这个问题的一个简单版本：一个数组里除了一个数字之外，其他的数字都出现了两次。请写程序找出这个只出现一次的数字。

这个题目的突破口在哪里？题目为什么要强调有一个数字出现一次，其他的出现两次？我们想到了异或运算的性质：任何一个数字异或它自己都等于0 。也就是说，如果我们从头到尾依次异或数组中的每一个数字，那么最终的结果刚好是那个只出现一次的数字，因为那些出现两次的数字全部在异或中抵消掉了。

有了上面简单问题的解决方案之后，我们回到原始的问题。如果能够把原数组分为两个子数组。在每个子数组中，包含一个只出现一次的数字，而其它数字都出现两次。如果能够这样拆分原数组，按照前面的办法就是分别求出这两个只出现一次的数字了。
我们还是从头到尾依次异或数组中的每一个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果。因为其它数字都出现了两次，在异或中全部抵消掉了。由于这两个数字肯定不一样，那么这个异或结果肯定不为0 ，也就是说在这个结果数字的二进制表示中至少就有一位为1 。我们在结果数字中找到第一个为1 的位的位置，记为第N 位。现在我们以第N 位是不是1 为标准把原数组中的数字分成两个子数组，第一个子数组中每个数字的第N 位都为1 ，而第二个子数组的每个数字的第N 位都为0 。
现在我们已经把原数组分成了两个子数组，每个子数组都包含一个只出现一次的数字，而其它数字都出现了两次。因此到此为止，所有的问题我们都已经解决。

> java

```java
//num1,num2分别为长度为1的数组。传出参数
//将num1[0],num2[0]设置为返回结果
public class Solution {
    public void FindNumsAppearOnce(int [] array,int num1[] , int num2[]) {
        if(array==null ||array.length<2)
            return ;
        int temp = 0;
        for(int i=0;i<array.length;i++)
            temp ^= array[i];

        int indexOf1 = findFirstBitIs(temp);
        for(int i=0;i<array.length;i++){
            if(isBit(array[i], indexOf1))
                num1[0]^=array[i];
            else
                num2[0]^=array[i];
        }
    }
     //在正数num的二进制表示中找到最右边是1的位
    public int findFirstBitIs(int num){
        int indexBit = 0;
        while(((num & 1)==0) && (indexBit)<8*4){
            num = num >> 1;
            ++indexBit;
        }
        return indexBit;
    }
    //判断在num的二进制表示中从右边数起的indexBit位是不是1.
    public boolean isBit(int num,int indexBit){
        num = num >> indexBit;
        return (num & 1) == 1;
    }
}
```

41.和为S的两个数字VS和为s的连续正数序列（数组）

41.1 和为s的两个数字

一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

数列满足递增，设两个头尾两个指针i和j，

- 若ai + aj == sum，就是答案（相差越远乘积越小）

- 若ai + aj > sum，aj肯定不是答案之一（前面已得出 i 前面的数已是不可能），j -= 1
- 若ai + aj < sum，ai肯定不是答案之一（前面已得出 j 后面的数已是不可能），i += 1

时间复杂度为O(n)。

```
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]);
                break;
            }
            else if(array[i]+array[j]>sum){
                j--;
            }
            else
                i++;
        }
        return list;
    }
}
```

41.2 和为s的连续正数序列

输入一个正数s，打印出所有和为s的连续正数序列（至少含有两个数）。例如输入15，由于1+2+3+4+5=4+5+6=7+8=15，所以结果打印出三个连续序列1~5、4~6和7~8。

考虑用两个数small和big分别表示序列的最小值和最大值。首先把small初始化为1，big初始化为2，如果从small到big的序列和大于s，我们可以从序列中去掉较小的值，也就是增大small的值。如果从small到big的序列和小于s，我们可以增大big，让这个序列包含更多的数字。因为这个序列至少要有两个数字，我们一直增加small到（1+s）/2为止。

> java

```
import java.util.ArrayList;
/*
*初始化small=1, big=2;
```

```
*small到big序列和小于sum，big++;大于sum，small++;
*当small增加到(1+sum)/2是停止
*/
public class Solution {
    public ArrayList<ArrayList<Integer>> FindContinuousSequence(int sum) {
        ArrayList<ArrayList<Integer>> lists=new ArrayList<ArrayList<Integer>>();
        if(sum<=1){return lists;}
        int small=1;
        int big=2;
        while(small!=(1+sum)/2){          //当small==(1+sum)/2的时候停止
            int curSum=sumOfList(small,big);
            if(curSum==sum){
                ArrayList<Integer> list=new ArrayList<Integer>();
                for(int i=small;i<=big;i++){
                    list.add(i);
                }
                lists.add(list);
                small++;big++;
            }else if(curSum<sum){
                big++;
            }else{
                small++;
            }
        }
        return lists;
    }
    public int sumOfList(int head,int leap){        //计算当前序列的和
        int sum=head;
        for(int i=head+1;i<=leap;i++){
            sum+=i;
        }
        return sum;
    }
}
```

42. 翻转单词顺序VS左旋转字符串（字符串）

42.1 翻转单词顺序

输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。为简单起见，标点符号和普通字母一样处理。例如输入字符串“I am a student”，则输出“student. a am I”。
可以先翻转整个句子，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

> java

```
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-1;i++){
            if(chars[i]==' '){
                int nextblank = i;
                reverse(chars,blank+1,nextblank-1);
                blank = nextblank;
            }
        }
        reverse(chars,blank+1,chars.length-1);//单独翻转最后一个单词
        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--;
        }
    }
}
```

42.2 左旋转字符串

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”。

以“abcdefg”为例，我们可以把它分为两部分。由于想把它的前两个字符移到后面，我们就把钱两个字符分到第一部分，把后面的所有字符都分到第二部分。然后先翻转这两部分，于是就得到“bagfedc”。接下来在翻转整个字符串，得到的"cdefgab"刚好就是把原始字符串左旋转2位的结果。

```
public class Solution {
    public String LeftRotateString(String str,int n) {
        char[] chars = str.toCharArray();
        if(chars.length < n) return "";
        reverse(chars, 0, n-1);
        reverse(chars, n, chars.length-1);
```

```
        reverse(chars, 0, chars.length-1);
        return new String(chars);
    }

    public void reverse(char[] chars,int low,int high){
        char temp;
        while(low<high){
            temp = chars[low];
            chars[low] = chars[high];
            chars[high] = temp;
            low++;
            high--;
        }
    }
}
```

43. N个骰子的点数（null）

44. 扑克牌的顺子（数组）

从扑克牌中随机抽5张牌，判断是不是顺子，即这5张牌是不是连续的。2~10为数字本身，A为1，J为11，Q为12，K为13，而大小王可以看做是任意数字，这里定为0.

> java

```
import java.util.*;
public class Solution {
    public boolean isContinuous(int [] numbers) {
        int length = numbers.length;
        if(numbers==null||length==0)return false;//特殊情况
        Arrays.sort(numbers);//排序
        //统计数组中0的个数
        int numberOfZero = 0;
        for(int i =0;i<length&&numbers[i]==0;i++){
            ++numberOfZero;
        }

        int numberOfGap = 0;
        int small = numberOfZero;
        int big = small+1;
        while(big<length){
            //含有对子，不可能是顺子
            if(numbers[small]==numbers[big]){
```

```
                return false;
            }
            //统计数组中的间隔数目
            numberOfGap += numbers[big]-numbers[small]-1;
            small=big;
            big++;
        }
        //如果间隔数小于等于零的数量则可以组成顺子，否则不行。
        if(numberOfGap<=numberOfZero){
            return true;
        }else{return false;}
    }
}
```

45. 圆圈中最后剩下的数字（链表）

0、.....，n-1这n个数字排成一个圆圈，从数字0开始每次从这个圆圈里删除第m个数字。求出这个圆圈里剩下的最后一个数字。
约瑟夫环问题，用环形链表模拟圆圈的经典解法，

> java

```
public class Solution{
    public int LastRemaining_Solution(int n, int m){
        if(m<=0||n<=0)return -1;
        //先构造循环链表
        ListNode head= new ListNode(0);//头结点，值为0
        ListNode pre = head;
        ListNode temp = null;
        for(int i=1;i<n;i++){
            temp = new ListNode(i);
            pre.next = temp;
            pre = temp;
        }
        temp.next = head;//将第n-1个结点(也就是尾结点)指向头结点
        ListNode temp2 = null;

        while(n>=1){
            //每次都当前头结点找到第m个结点的前驱
            temp2=head;
            for(int i =1;i<m-1;i++){
                temp2 = temp2.next;
            }
            temp2.next = temp2.next.next;
            head = temp2.next;//设置当前头结点
```

```
            n--;
        }
        return head.val;
    }
}
```

java

```
public class Solution
{
    public int LastRemaining_Solution(int n, int m)
    {
        if(n==0||m==0)return -1;
        int last=0;
        for(int i=2;i<=n;i++)
        {
            last=(last+m)%i;
        }
       return last ;
    }
}
```

46. 求1+2+.....+n（逻辑）

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）。

1. 需利用逻辑与的短路特性实现递归终止。
2. 当n==0时，(n>0)&&((sum+=Sum_Solution(n-1))>0)只执行前面的判断，为false，然后直接返回0；
3. 当n>0时，执行sum+=Sum_Solution(n-1)，实现递归计算Sum_Solution(n)。

java

```
public class Solution {
    public int Sum_Solution(int n) {
        int sum=n;
        boolean ans = (n>0)&&((sum+=Sum_Solution(n-1))>0);
        return sum;

    }
}
```

47. 不用加减乘除做加法（位运算）

首先看十进制是如何做的： 5+7=12，三步走

1. 第一步：相加各位的值，不算进位，得到2。
2. 第二步：计算进位值，得到10. 如果这一步的进位值为0，那么第一步得到的值就是最终结果。
3. 第三步：重复上述两步，只是相加的值变成上述两步的得到的结果2和10，得到12。

同样我们可以用三步走的方式计算二进制值相加： 5-101，7-111

1. 第一步：相加各位的值，不算进位，得到010，二进制每位相加就相当于各位做异或操作，101^{111}。
2. 第二步：计算进位值，得到1010，相当于各位做与操作得到101，再向左移一位得到1010，(101&111)<<1。
3. 第三步重复上述两步， 各位相加 010$^{1010=1000}$，进位值为100=(010&1010)<<1。继续重复上述两步：1000^{100} = 1100，进位值为0，跳出循环，1100为最终结果。

```
public class Solution {
    public int Add(int num1,int num2) {
        while (num2!=0) {
            int temp = num1^num2;
            num2 = (num1&num2)<<1;
            num1 = temp;
        }
        return num1;
    }
}
```

49. 把字符串转换成整数（字符串）

将一个字符串转换成一个整数，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。
问题不难，但是要把很多特殊情况都考虑进去，却并不容易。需要考虑的特殊情况有以下几个：

1. 空指针null
2. 字符串为空
3. 正负号
4. 上下溢出 Integer.MAX_VALUE (2$^{31-1)}$ Integer.MIN_VALUE(-2$^{31)}$

java

```java
public class Solution {
    public int StrToInt(String str) {
        if(str==null||str.length()==0){return 0;}//空指针或空字符串
        char[] c = str.toCharArray();
        boolean minus=false;
        int i=0;
        //正负号
        if(c[i]=='+'){
            i++;
        }else if(c[i]=='-'){
            i++;
            minus=true;
        }
        int num=0;
        if(i<c.length){
            num = StrToIntCore(c,minus,i);
        }else{
            return num;
        }
        return num;

    }
    int StrToIntCore(char[] str,boolean minus,int i){
        int num=0;
        for(int j=i;j<str.length;j++){
            if(str[j]>='0'&&str[j]<='9'){
                int flag = minus?-1:1;
                num = num*10+flag*(str[j]-'0');
                if((!minus&&num>Integer.MAX_VALUE)||minus&&num<Integer.MIN_VALUE){//上下溢出
                    num=0;
                    break;
                }
            }else{//非法数值
                num=0;
                break;
            }

        }
        return num;
    }
}
```

50.树中两个结点的最低公共祖先（二叉树）

50.1 二叉搜索树的最低公共祖先

二叉搜索树是经过排序的，位于左子树的节点都比父节点小，位于右子树的节点都比父节点大。既然要找最低的公共祖先节点，我们可以从根节点开始进行比较。若当前节点的值比两个节点的值都大，那么最低的祖先节点一定在当前节点的左子树中，则遍历当前节点的左子节点；反之，若当前节点的值比两个节点的值都小，那么最低的祖先节点一定在当前节点的右子树中，则遍历当前节点的右子节点；这样，直到找到一个节点，位于两个节点值的中间，则找到了最低的公共祖先节点。

> java

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null||root==q||root==p)return root;
        if(root.val>p.val&&root.val>q.val){
            return lowestCommonAncestor(root.left,p,q);
        }else if(root.val<p.val&&root.val<q.val){
            return lowestCommonAncestor(root.right,p,q);
        }else return root;

    }
}
```

50.2 普通二叉树的最低公共祖先

一种简单的方法是DFS分别寻找到两个节点p和q的路径，然后对比路径，查看他们的第一个分岔口，则为LCA。
这个思路比较简单，代码写起来不如下面这种方法优雅：

我们仍然可以用递归来解决，递归寻找两个带查询LCA的节点p和q，当找到后，返回给它们的父亲。如果某个节点的左右子树分别包括这两个节点，那么这个节点必然是所求的解，返回该节点。否则，返回左或者右子树（哪个包含p或者q的就返回哪个）。复杂度O(n)

> java

```
public class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null||root==p||root==q){return root;}
        TreeNode left = lowestCommonAncestor(root.left,p,q);
        TreeNode right = lowestCommonAncestor(root.right,p,q);
        if(left!=null&&right!=null)return root;
```

```
        return left!=null? left:right;

    }
}
```

51. 数组中重复的数字（数组）

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}，那么对应的输出是重复的数字2或者3。

> java

```
public class Solution {
    public boolean duplicate(int numbers[],int length,int [] duplication) {
        if(numbers==null||length==0){return false;}//空指针或空数组
        // 判断数组是否合法,即每个数都在0~n-1之间
        for(int i=0;i<length;i++){
            if(numbers[i]>length-1||numbers[i]<0){
                return false;
            }
        }
        //若数值与下标不同，则调换位置；
        //比较位置下标为数值(numbers[i])的数值(numbers[numbers[i]])与该数值(numbers[i])是否一致，若一致，则说明有重复数字
        for(int i=0;i<length;i++){
            while(numbers[i]!=i){
                if(numbers[i]==numbers[numbers[i]]){
                    duplication[0] = numbers[i];
                    return true;
                }
                int temp=numbers[i];
                numbers[i]=numbers[temp];
                numbers[temp]=temp;
            }
        }
        return false;
    }
}
```

52. 构建乘积数组（数组）

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。

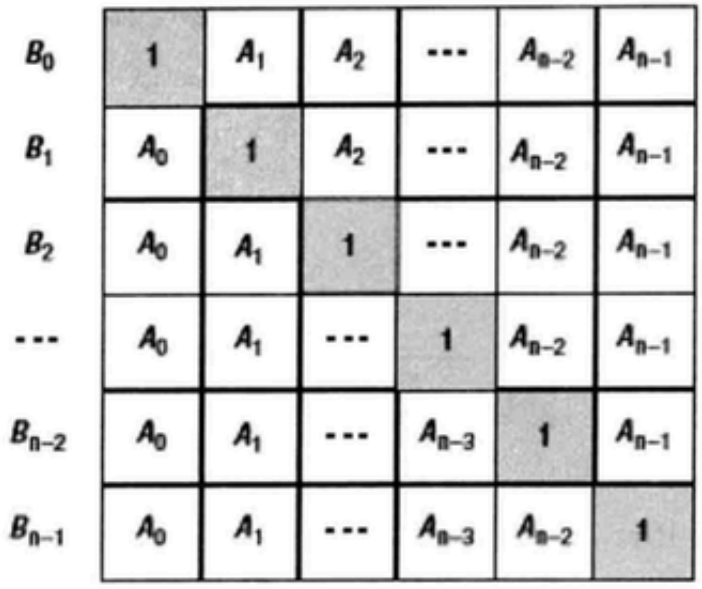


图 8.1　把矩阵 B 看成由一个矩形来创建

不妨定义 C[i]=A[0]×A[1]×...×A[i-1]，D[i]=A[i+1]×...×A[n-2]×A[n-1]。C[i]可以用自上而下的顺序计算出来，即 C[i]=C[i-1] ×A[i-1]。类似的，D[i]可以用自下而上的顺序计算出来，即 D[i]=D[i+1] ×A[i+1]。

java

```
import java.util.ArrayList;
public class Solution {
    public int[] multiply(int[] A) {
        int length = A.length;
        int[] B = new int[length];
        if(length!=0){
            B[0]=1;
            for(int i=1;i<length;i++){
                B[i]=B[i-1]*A[i-1];
            }
            int temp=1;
            for(int j=length-2;j>=0;j--){
                temp = temp*A[j+1];
                B[j]=temp*B[j];
            }
        }
        return B;
    }
}
```

53. 正则表达式匹配（字符串）

当模式中的第二个字符不是“*”时：

1. 如果字符串第一个字符和模式中的第一个字符相匹配，那么字符串和模式都后移一个字符，然后匹配剩余的。
2. 如果字符串第一个字符和模式中的第一个字符相不匹配，直接返回false。

而当模式中的第二个字符是“*”时：

如果字符串第一个字符跟模式第一个字符不匹配，则模式后移2个字符，继续匹配。如果字符串第一个字符跟模式第一个字符匹配，可以有3种匹配方式：

1. 模式后移2字符，相当于$x*$被忽略；
2. 字符串后移1字符，模式后移2字符；
3. 字符串后移1字符，模式不变，即继续匹配字符下一位，因为*可以匹配多位；

> java

```
public class Solution {
    public boolean match(char[] str, char[] pattern) {
    if (str == null || pattern == null) {
        return false;
    }
    int strIndex = 0;
    int patternIndex = 0;
    return matchCore(str, strIndex, pattern, patternIndex);
}

public boolean matchCore(char[] str, int strIndex, char[] pattern, int patternIndex
) {
    //有效性检验：str到尾，pattern到尾，匹配成功
    if (strIndex == str.length && patternIndex == pattern.length) {
        return true;
    }
    //pattern先到尾，匹配失败
    if (strIndex != str.length && patternIndex == pattern.length) {
        return false;
    }
    //模式第2个是*，且字符串第1个跟模式第1个匹配,分3种匹配模式；如不匹配，模式后移2位
    if (patternIndex + 1 < pattern.length && pattern[patternIndex + 1] == '*') {
        if ((strIndex != str.length && pattern[patternIndex] == str[strIndex]) || (
pattern[patternIndex] == '.' && strIndex != str.length)) {
            return matchCore(str, strIndex, pattern, patternIndex + 2)//模式后移2,
视为x*匹配0个字符
```

```
                    || matchCore(str, strIndex + 1, pattern, patternIndex + 2)//视为模式匹配1个字符
                    || matchCore(str, strIndex + 1, pattern, patternIndex);//*匹配1个，再匹配str中的下一个
        } else {
            return matchCore(str, strIndex, pattern, patternIndex + 2);
        }
    }
    //模式第2个不是*，且字符串第1个跟模式第1个匹配，则都后移1位，否则直接返回false
    if ((strIndex != str.length && pattern[patternIndex] == str[strIndex]) || (pattern[patternIndex] == '.' && strIndex != str.length)) {
        return matchCore(str, strIndex + 1, pattern, patternIndex + 1);
    }
    return false;
    }
}
```

54. 表示数值的字符串（字符串）

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。

> java

```
public class Solution {
    boolean isNumeric(char[] s) {
        if(s.length==0) return false;
        if((s.length==1)&&(s[0]<'0'||s[0]>'9')) return false;
        if(s[0]=='+'||s[0]=='-'){
            if(s.length==2&&(s[1]=='.')) return false;
        }else if((s[0]<'0'||s[0]>'9')&&s[0]!='.') return false;//首位既不是符号也不是数字还不是小数点，当然是false
        int i = 1;
        while((i<s.length)&&(s[i]>='0'&&s[i]<='9')) i++;
        if(i<s.length&&s[i]=='.'){
            i++;
            //if(i>=s.length) return false;
            while((i<s.length)&&(s[i]>='0'&&s[i]<='9')) i++;
        }
        if(i<s.length&&(s[i]=='e'||s[i]=='E')){
            i++;
            if((i<s.length)&&(s[i]=='+'||s[i]=='-')){
                i++;
                if(i<s.length) while((i<s.length)&&(s[i]>='0'&&s[i]<='9')) i++;
```

```
                else return false;
            }else if(i<s.length){
                while((i<s.length)&&(s[i]>='0'&&s[i]<='9')) i++;
            }else return false;
        }
        if(i<s.length) return false;
        return true;
    }
}
```

55. 字符流中第一个不重复的数组（字符串）

使用一个HashMap来统计字符出现的次数，同时用一个ArrayList来记录输入流，每次返回第一个出现一次的字符都是在这个ArrayList（输入流）中的字符作为key去map中查找。

> java

```
import java.util.*;
public class Solution {
    //HashMap来统计字符出现的次数
    HashMap<Character, Integer> map=new HashMap();
    //ArrayList来记录输入流
    ArrayList<Character> list=new ArrayList<Character>();
    //Insert one char from stringstream
    public void Insert(char ch)
    {
        if(map.containsKey(ch)){
            int time = map.get(ch);
            time++;
            map.put(ch,time);
        }else{
            map.put(ch,1);
        }

        list.add(ch);
    }

    //return the first appearence once char in current stringstream
    public char FirstAppearingOnce()
    {   char ch='#';
        for(char k : list){//list迭代
            if(map.get(k)==1){
                ch=k;
                break;//得到第一个结果即可break
            }
```

```
        }

        return ch;
    }
}
```

56. 链表中环的入口结点（链表）

一个链表中包含环，如何找到环的入口结点？例如在下图的链表中，环的入口结点是结点3。

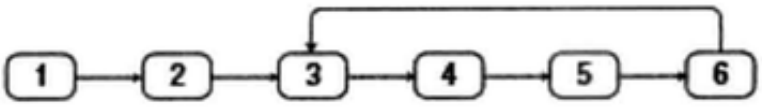


图 8.3　结点 3 是链表中环的入口结点

以3为例分析两个指针的移动规律。指针P_1和P_2在初始化时都指向链表的头结点。由于环中有4个结点，指针P_1先在链表上向前移动4步。接下来两个指针以相同的速度在链表上向前移动，直到它们相遇。它们相遇的结点正好是还的入口结点。

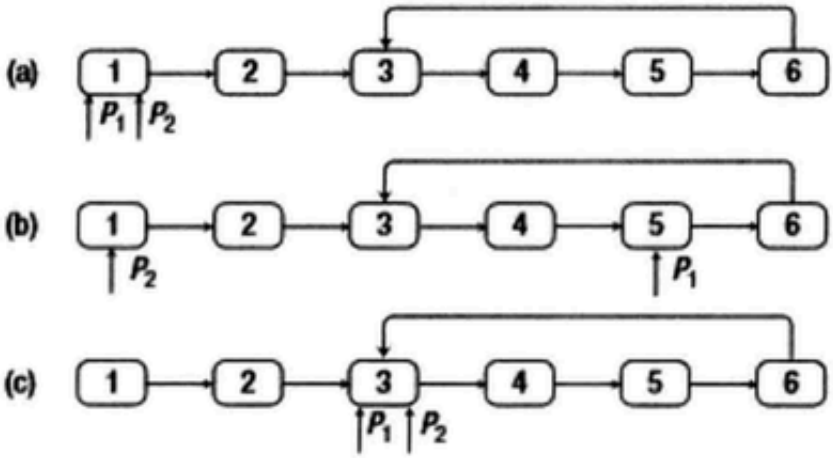


图 8.4　在有环的链表中找到环的入口结点的步骤。（1）指针 P_1 和 P_2 在初始化时都指向链表的头结点；（2）由于环中有 4 个结点，指针 P_1 先在链表上向前移动 4 步；（3）指针 P_1 和 P_2 以相同的速度在链表上向前移动直到它们相遇。它们相遇的结点就是环的入口结点

剩下的问题就是如何得到环中结点的数目。我们可以使用一快一慢两个指针。若两个指针相遇，说明链表中有环。两个指针相遇的结点一定是在环中的。可以从这个结点出发，一边继续向前移动一边计数，当再次回到这个结点时，就可以得到环中结点数了
实现代码如下：

> java版本

```java
/*
 public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}
*/
public class Solution {
    //找到一快一满指针相遇处的节点，相遇的节点一定是在环中
    public static ListNode meetingNode(ListNode pHead)
    {
        if(pHead == null){return null;}//空链表处理
        ListNode pslow = pHead.next;

        if(pslow == null){return null;}//无环链表处理

        ListNode pfast = pslow.next;
        while(pfast!=null && pslow!=null){
            if(pslow==pfast){return pfast;}

            pslow = pslow.next;//慢指针

            pfast = pfast.next;
            if(pfast!=null){
                pfast = pfast.next;
            }//块指针

        }
        return null;
    }

    public ListNode EntryNodeOfLoop(ListNode pHead){
        ListNode meetingNode=meetingNode(pHead);//相遇结点
        //环的结点个数
        if(meetingNode==null){return null;}//是否有环
        int nodesInLoop = 1;
        ListNode p1=meetingNode;
        while(pnext!=meetingNode){
            p1=pnext;
            ++nodesInLoop;
        }
        //p1慢指针,先往前走
        p1=pHead;
        for(int i=0;i<nodesInLoop;i++){
            p1=pnext;
```

```
        }
        //p1,p2同步走，相遇的地方即为环入口
        ListNode p2=pHead;
        while(p1!=p2){
            p1=pnext;
            p2=p2.next;
        }
        return p1;
    }
}
```

> python 版本

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    def meetingNode(self,pHead):
        if not pHead:
            return None
        pslow =pHead.next
        if not pslow:
            return None
        pfast = pslow.next
        while pfast and pslow:
            if pslow==pfast:
                return pfast
            pslow = pslow.next

            pfast = pfast.next
            if pfast:
                pfast=pfast.next
        return None
    def EntryNodeOfLoop(self, pHead):
        meetingNode = self.meetingNode(pHead)
        if not meetingNode:
            return None
        nodesInLoop = 1
        p1 = meetingNode
        while pnext!=meetingNode:
            p1=pnext
            nodesInLoop +=1
        p1 = pHead
        for i in xrange(0,nodesInLoop):
            p1=pnext
```

```
        p2=pHead
        while p1!=p2:
            p1=pnext
            p2=p2.next
        return p1
```

57. 删除链表中重复的结点（链表）

在一个排序的链表中，如何删除重复的结点？如在下图中重复结点被删除之后，链表如下图所示：

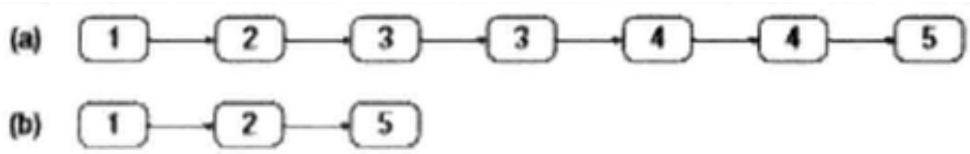


图 8.5　删除链表中的重复结点。（a）一个有 7 个结点的链表；（b）当重复的结点被删除之后，链表中只剩下 3 个结点

从头遍历整个链表。如果当前结点的值与下一个节点的值相同，那么它们就是重复的结点，都可以被删除。为了保证删除之后的链表仍然是相连的而没有中间断开，我们要把当前结点的前一个结点preNode和后面值比当前结点的值要大的结点相连。要确保preNode要始终与下一个没有重复的结点连接在一起。

实现代码如下：

> java递归版

```
/*
 public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}
*/
public class Solution {
    public ListNode deleteDuplication(ListNode pHead) {
        if (pHead == null || pHead.next == null) { // 只有0个或1个结点，则返回
            return pHead;
        }
        if (pHead.val == pHead.next.val) { // 当前结点是重复结点
```

```
            ListNode pNode = pHead.next;
            while (pNode != null && pNode.val == pHead.val) {
                // 跳过值与当前结点相同的全部结点,找到第一个与当前结点不同的结点
                pNode = pNode.next;
            }
            return deleteDuplication(pNode); // 从第一个与当前结点不同的结点开始递归
        } else { // 当前结点不是重复结点
            pHead.next = deleteDuplication(pHead.next); // 保留当前结点，从下一个结点开始递归
            return pHead;
        }
    }
}
```

> python版本

```
# -*- coding:utf-8 -*-
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None
class Solution:
    def deleteDuplication(self, pHead):
        if not pHead or not pHead.next:
            return pHead
        if pHead.val==pHead.next.val:
            pNode = pHead.next
            while pNode and pNode.val == pHead.val:
                 pNode = pNode.next
            return self.deleteDuplication(pNode)
        else:
            pHead.next = self.deleteDuplication(pHead.next)
            return pHead
```

> java非递归

```
/*
 public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}
```

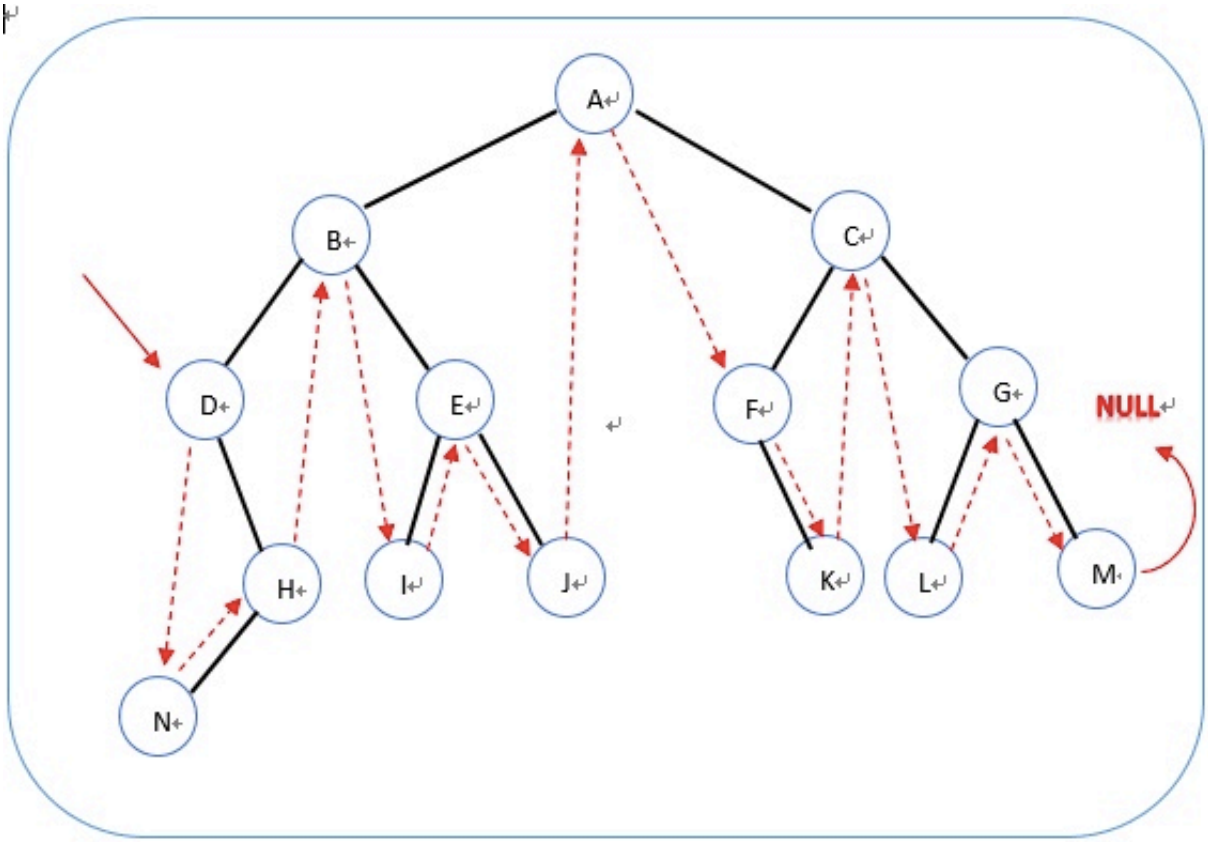
```
*/
public class Solution {
    public ListNode deleteDuplication(ListNode pHead)
    {
        if(pHead==null)return null;
        ListNode preNode = null;
        ListNode node = pHead;
        while(node!=null){
            ListNode nextNode = node.next;
            boolean needDelete = false;
            //需要删除重复节点的情况
            if(nextNode!=null&&nextNode.val==node.val){
                needDelete = true;
            }
            //不重复结点不删除
            if(!needDelete){
                preNode = node;
                node = node.next;
            }
            //重复节点删除
            else{
                int value = node.val;
                ListNode toBeDel = node;
                //连续重复结点
                while(toBeDel != null && toBeDel.val == value){
                    nextNode = toBeDel.next;
                    toBeDel = nextNode;
                    if(preNode==null)
                        pHead = nextNode;
                    else
                        preNode.next = nextNode;
                    node = nextNode;
                }
            }
        }

        return pHead;
    }
}
```

58. 二叉树的下一个结点（二叉树）

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。

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。



我们可发现分成两大类：

1. 有右子树的，那么下个结点就是右子树最左边的点；（eg：D，B，E，A，F，C，G）
2. 没有右子树的，也可以分成两类，a)是父节点左孩子（eg：N，I，L），那么父节点就是下一个节点； b)是父节点的右孩子（eg：H，J，K，M）找他的父节点的父节点的父节点...直到当前结点是其父节点的左孩子位置。如果没有eg：M，那么他就是尾节点。 >java

```
public class Solution {
    public TreeLinkNode GetNext(TreeLinkNode pNode)
    {
        if(pNode==null){return null;}
        if(pNode.right!=null){
            pNode = pNode.right;
            while(pNode.left!=null){
                pNode = pNode.left;
            }
            return pNode;
        }
        while(pNode.next!=null){
            if(pNode.next.left==pNode)return pNode.next;
            pNode = pNode.next;
        }
        return null;
    }
}
```

59. 对称的二叉树（二叉树）

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。

如果先序遍历的顺序分为两种先左后右和先右后左两种顺序遍历，如果两者相等说明二叉树是对称的二叉树

> java

```
public class Solution {
    boolean isSymmetrical(TreeNode pRoot){
        return isSymmetrical(pRoot,pRoot);
    }
    boolean isSymmetrical(TreeNode pRoot1,TreeNode pRoot2){
        if(pRoot1==null&&pRoot2==null)return true;
        if(pRoot1==null||pRoot2==null)return false;
        if(pRoot1.val==pRoot2.val){return
            isSymmetrical(pRoot1.left,pRoot2.right)&&isSymmetrical(pRoot1.right,pRo
ot2.left);
        }else
            return false;
    }
}
```

60. 把二叉树打印成多行（二叉树）

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。
用end记录每层结点数目，start记录每层已经打印的数目，当start=end，重新建立list，开始下一层打印。

> java

```
public class Solution {
    ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        if(pRoot==null){return result;}
        LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
        ArrayList<Integer> list = new ArrayList<Integer>();
        queue.add(pRoot);
        int start = 0,end = 1;
```

```
        while(!queue.isEmpty()){
            TreeNode treenode = queue.remove();
            list.add(treenode.val);
            start++;
            if(treenode.left!=null){queue.add(treenode.left);}
            if(treenode.right!=null){queue.add(treenode.right);}
            if(start==end){
                end = queue.size();
                start = 0;
                result.add(list);
                list = new ArrayList<Integer>();
            }
        }
        return result;
    }
}
```

61. 按S型打印二叉树（二叉树）

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。

表 8.1　按之字形顺序打印图 8.9 中二叉树的最初 7 步。Stack1 和 Stack2 中最右边的结点位于栈顶

步骤	操作	Stack1 中的结点	Stack2 中的结点
1	打印结点 1	2, 3	
2	打印结点 3	2	7, 6
3	打印结点 2		7, 6, 5, 4
4	打印结点 4	8, 9	7, 6, 5
5	打印结点 5	8, 9, 10, 11	7, 6
6	打印结点 6	8, 9, 10, 11, 12, 13	7
7	打印结点 7	8, 9, 10, 11, 12, 13, 14, 15	

> java

```
import java.util.ArrayList;
import java.util.Stack;
```

```
public class Solution {
    public ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {
        ArrayList<ArrayList<Integer>> alist =new ArrayList<ArrayList<Integer>>();
        if(pRoot==null)return alist;
        Stack<TreeNode> stack1 = new Stack<TreeNode>();
        stack1.add(pRoot);
        Stack<TreeNode> stack2 = new Stack<TreeNode>();

        while(!stack1.isEmpty()||!stack2.isEmpty()){
            if(!stack1.isEmpty()){
                ArrayList<Integer> alist2 = new ArrayList<Integer>();
                while(!stack1.isEmpty()){
                    TreeNode treenode=stack1.pop();
                    alist2.add(treenode.val);
                    if(treenode.left!=null){
                        stack2.add(treenode.left);
                    }
                    if(treenode.right!=null){
                        stack2.add(treenode.right);
                    }
                }
                alist.add(alist2);
            }

            else{
                ArrayList<Integer> alist2 = new ArrayList<Integer>();
                while(!stack2.isEmpty()){
                    TreeNode treenode = stack2.pop();
                    alist2.add(treenode.val);
                    if(treenode.right!=null){
                        stack1.add(treenode.right);
                    }
                    if(treenode.left!=null){
                        stack1.add(treenode.left);
                    }
                }
                alist.add(alist2);

            }
        }
        return alist;

    }
}
```

62. 序列化与反序列化二叉树（二叉树）

请实现两个函数，分别用来序列化和反序列化二叉树
算法思想：根据前序遍历规则完成序列化与反序列化。所谓序列化指的是遍历二叉树为字符串；所谓反序列化指的是依据字符串重新构造成二叉树。

依据前序遍历序列来序列化二叉树，因为前序遍历序列是从根结点开始的。当在遍历二叉树时碰到Null指针时，这些Null指针被序列化为一个特殊的字符“#”。另外，结点之间的数值用逗号隔开。

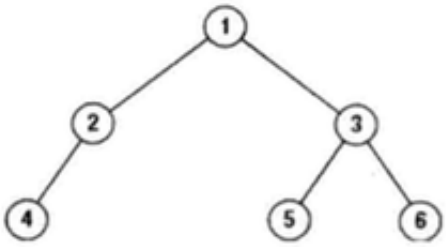


图 8.10　一颗被序列化成字符串“1,2,4,$,$,$,3,5,$,$,6,$,$”的二叉树

> java

```
public class Solution {
    int index = -1;   //计数变量

    String Serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();//新建字符串
        if(root == null){
            sb.append("#,");
            return sb.toString();
        }
        //递归
        sb.append(root.val + ",");
        sb.append(Serialize(root.left));
        sb.append(Serialize(root.right));
        return sb.toString();
  }
    TreeNode Deserialize(String str) {
        index++;
        //int len = str.length();
        //if(index >= len){
        //    return null;
       // }
        String[] strr = str.split(",");
        TreeNode node = null;
        if(!strr[index].equals("#")){
            node = new TreeNode(Integer.valueOf(strr[index]));
            node.left = Deserialize(str);
            node.right = Deserialize(str);
```

```
        }
        return node;
    }
}
```

63. 二叉搜索树的第K个结点（二叉树）

给定一颗二叉搜索树，请找出其中的第k大的结点。例如在下图二叉搜索树里，按结点数值大小顺序第三个结点的值是4.

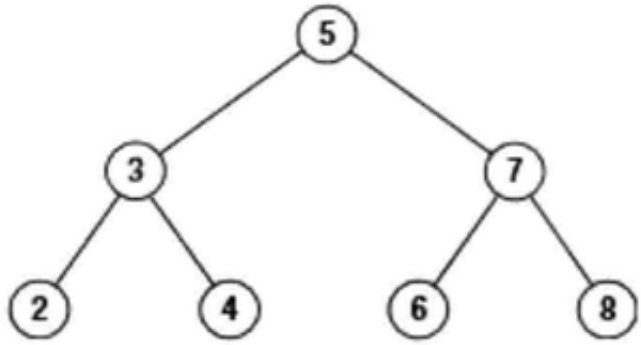


图 8.11　一个有 7 个结点二叉搜索树，其中按结点数值大小顺序第三个结点的值是 4

如果按照中序遍历的顺序遍历一颗二叉搜索树，遍历序列的数值是递增排序的，只需要用中序遍历算法遍历一颗二叉搜索树，就很容易找出它的第K大的结点。

> java

```
public class Solution {
   int index = 0; //计数器
    TreeNode KthNode(TreeNode root, int k)
    {
        if(root != null){ //中序遍历寻找第k个
            TreeNode node = KthNode(root.left,k);
            if(node != null)
                return node;
            index ++;
            if(index == k)
                return root;
            node = KthNode(root.right,k);
            if(node != null)
                return node;
        }
        return null;
    }
}
```

64. 数据流中的中位数（二叉树）

Java的PriorityQueue是从JDK1.5开始提供的新的数据结构接口，默认内部是自然排序，结果为小顶堆，也可以自定义排序器，比如下面反转比较，完成大顶堆。

为了保证插入新数据和取中位数的时间效率都高效，这里使用大顶堆+小顶堆的容器，并且满足：

1. 两个堆中的数据数目差不能超过1，这样可以使中位数只会出现在两个堆的交接处；
2. 大顶堆的所有数据都小于小顶堆，这样就满足了排序要求。

表 8.2　使用没有排序的数组、排序的数组、排序的链表、二叉搜索树、AVL 树、最大堆和最小堆等不同数据结构的时间效率

数据结构	插入的时间效率	得到中位数的时间效率
没有排序的数组	O(1)	O(n)
排序的数组	O(n)	O(1)
排序的链表	O(n)	O(1)
二叉搜索树	平均 O(logn)，最差 O(n)	平均 O(logn)，最差 O(n)
AVL 数	O(logn)	O(1)
最大堆和最小堆	O(logn)	O(1)

> java

```
import java.util.Comparator;
import java.util.PriorityQueue;

public class Solution {
    int count=0;
    PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();
    PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(11, new Comparator<
Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            //PriorityQueue默认是小顶堆，实现大顶堆，需要反转默认排序器
            return o2.compareTo(o1);
        }
    });

    public void Insert(Integer num) {
```

```
    if (count %2 == 0) {//当数据总数为偶数时，新加入的元素，应当进入小根堆
        //（注意不是直接进入小根堆，而是经大根堆筛选后取大根堆中最大元素进入小根堆）
        //1.新加入的元素先入到大根堆，由大根堆筛选出堆中最大的元素
        maxHeap.offer(num);
        int filteredMaxNum = maxHeap.poll();
        //2.筛选后的【大根堆中的最大元素】进入小根堆
        minHeap.offer(filteredMaxNum);
    } else {//当数据总数为奇数时，新加入的元素，应当进入大根堆
        //（注意不是直接进入大根堆，而是经小根堆筛选后取小根堆中最大元素进入大根堆）
        //1.新加入的元素先入到小根堆，由小根堆筛选出堆中最小的元素
        minHeap.offer(num);
        int filteredMinNum = minHeap.poll();
        //2.筛选后的【小根堆中的最小元素】进入大根堆
        maxHeap.offer(filteredMinNum);
    }
    count++;
}

public Double GetMedian() {
    if (count %2 == 0) {
        return new Double((minHeap.peek() + maxHeap.peek())) / 2;
    } else {
        return new Double(minHeap.peek());
    }
}
}
```

65. 滑动窗口的最大值（数组）

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]}。

表 8.4　找出数组{2, 3, 4, 2, 6, 2, 5, 1}中大小为 3 的滑动窗口的最大值的步骤。在“队列中的下标”一列中，小括号前面的数字表示一个数字在输入数组中的下标。为了方便读者理解，下标对应的在数组中的数字在后面的小括号中标出。

步骤	插入数字	滑动窗口	队列中的下标	最大值
1	2	2	0(2)	N/A
2	3	2, 3	1(3)	N/A
3	4	2, 3, 4	2(4)	4
4	2	3, 4, 2	2(4), 3(2)	4
5	6	4, 2, 6	4(6)	6
6	2	2, 6, 2	4(6), 5(2)	6
7	5	6, 2, 5	4(6), 6(5)	6
8	1	2, 5, 1	6(5), 7(1)	5

> java

```java
import java.util.ArrayList;
import java.util.LinkedList;
public class Solution {

    public ArrayList<Integer> maxInWindows(int [] num, int size)
    {
        ArrayList<Integer> ret = new ArrayList<>();
        if (num == null) {
            return ret;
        }
        if (num.length < size || size < 1) {
            return ret;
        }

        LinkedList<Integer> indexDeque = new LinkedList<>();
        //前size-1个中，前面比num[i]小的，对应下标从下标队列移除；
        for (int i = 0; i < size - 1; i++) {
            if (!indexDeque.isEmpty() && num[i] > num[indexDeque.getLast()]) {
                indexDeque.removeLast();
            }
            indexDeque.addLast(i);
        }
        //从第size-1个开始；前面比num[i]小的，对应下标从下标队列移除；
        for (int i = size - 1; i < num.length; i++) {
            while(!indexDeque.isEmpty() && num[i] > num[indexDeque.getLast()]) {
```

```
                indexDeque.removeLast();
            }
            //把下一个下标加入队列中
            indexDeque.addLast(i);
            //当第一个数字的下标与当前处理的数字的下标之差大于或者等于滑动窗口的大小时，这个数字已经从窗口划出，可以移除了；
            if (i - indexDeque.getFirst() + 1 > size) {
                indexDeque.removeFirst();
            }
            //下标队列的第一个是滑动窗口最大值对应的下标；
            ret.add(num[indexDeque.getFirst()]);
        }
        return ret;
    }
}
```

66. 矩阵中的路径（数组）

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，则该路径不能再进入该格子。 例如下面的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。

a b c e
s f c s
a d e e

> java

```
public class Solution {
  public int movingCount(int threshold, int rows, int cols)
    {
        boolean[] visited=new boolean[rows*cols];
        return movingCountCore(threshold, rows, cols, 0,0,visited);
    }
    private int movingCountCore(int threshold, int rows, int cols,
            int row,int col,boolean[] visited) {
        if(row<0||row>=rows||col<0||col>=cols) return 0;
        int i=row*cols+col;
        if(visited[i]||!checkSum(threshold,row,col)) return 0;
        visited[i]=true;
        return 1+movingCountCore(threshold, rows, cols,row,col+1,visited)
```

```
                +movingCountCore(threshold, rows, cols,row,col-1,visited)
                +movingCountCore(threshold, rows, cols,row+1,col,visited)
                +movingCountCore(threshold, rows, cols,row-1,col,visited);
    }
    private boolean checkSum(int threshold, int row, int col) {
        int sum=0;
        while(row!=0){
            sum+=row%10;
            row=row/10;
        }
        while(col!=0){
            sum+=col%10;
            col=col/10;
        }
        if(sum>threshold) return false;
        return true;
    }
}
```