Android面试题解（算法篇）

Q：找出数组中有出现的次数超过数组长度的一半的数字，如果不存在则输出0。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。
Q：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。
Q：输入一个递增排序的数组和一个数字S，在数组中查找两个数，使得他们的和正好是S，如果有多对数字的和等于S，输出两个数的乘积最小的。
Q：输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分、所有的偶数位于数组的后半部分，并保证奇数和奇数、偶数和偶数之间的相对位置不变。
Q：实现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。
Q：给定一棵二叉搜索树，请找出其中的第k小的结点。例如（5，3，7，2，4，6，8）中第三小结点的值为4。
Q：输入一棵二叉树，判断该二叉树是否是平衡二叉树。
Q：一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法（先后次序不同算不同的结果）。
Q：我们可以用*21的小矩形横着或者竖着去覆盖更大的矩形。请问用n个*21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?
Q：地上有一个m行和n列的方格，一个机器人从坐标（0,0）的格子开始移动，每一次只能向左、右、上、下四个方向移动一格，但是不能进入行坐标和列坐标的数位之和大于k的格子。 例如当k为18时，机器人能够进入方格（35,37），因为3+5+3+7 = 18；但是不能进入方格（35,38），因为3+5+3+8 = 19。请问该机器人能够达到多少个格子?
Q：求出从1 到 n 中1出现的次数。
Q：把只包含质因子2、3和5的数称作丑数。例如6、8都是丑数，而14不是，因为它包含质因子7。 习惯上我们把1当做是第一个丑数。求按从小到大的顺序的第N个丑数。
Q：输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。例如如果输入如下4 X 4矩阵，则依次打印出数字1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.

Q：怎么理解数据结构?

技术点：数据结构
思路：数据结构的定义、分类
参考回答：研究数据的逻辑结构和物理结构以及它们之间相互关系，并对这种结构定义相应的运算，而且确保经过这些运算后所得到的新结构仍然是原来的结构类型。

按照逻辑结构分类

线性结构：线性表、栈、队列

非线性结构：树、图

按照存储结构分为顺序结构、链式结构、索引结构、哈希结构

Q：什么是斐波那契数列?

技术点：递归和循环
思路：斐波那契数列的定义
参考回答：斐波那契数列指的是这样的数列1，1，2，3，5，8，13，21，34，55，89，144，...即这个数列从第3项开始，每一项都等于前两项之和，数学表示F(1)=1，F(2)=1，F(3)=2,F(n)=F(n-1)+F(n-2)（n>=4，n∈N*）

Q：迭代和递归的特点，并比较优缺点

技术点：递归和循环
参考回答：递归和迭代都是循环的一种，特点：
递归就是通过重复调用函数自身实现循环；满足终止条件时会逐层返回来结束循环
迭代通过函数内某段代码实现循环；使用计数器结束循环

	优点	**缺点**
递归	代码更简洁清晰，可读性更好	需要调用函数，会造成空间的浪费；使用栈机制，循环次数太多易造成堆栈溢出
迭代	效率高；无额外开销，节省空间	代码不如递归简洁

Q：了解哪些查找算法，时间复杂度都是多少?

- 技术点：查找
- 参考回答：代码详见java实现常见查找算法

名称	类型	特点	时间复杂度	适用
顺序查找	静态查找表	从表中第一个(或最后一个)记录开始，逐个进行记录的关键字和给定值比较	O(n)	无序表
有序查找	静态查找表	根据分隔点的选择不同分以下三种查找方法：	O(logn)	有序表
	1.二分查找	取中间值为比较对象，若等于关键字，则查找成功；若大于关键字，则在比较对象的左半区继续查找；若小于关键字，则在比较对象的右半区继续查找。不断重复上述过程直到查找成功，若所有查找区域无记录则查找失败		
	2.插值查找	是根据要查找的关键字与查找表中最大最小记录的关键字比较后的查找方法，其核心就在于插值的计算公式 (key-a[low])/(a[high]-a[low])*(high-low)		表长较大而关键字分布比较均匀
	3.斐波那契查找	在二分查找的基础上根据斐波那契数列进行分割的		

名称	类型	特点	时间复杂度	适用
线性索引查找	静态查找表	引入索引并将索引项集合组织为线性结构，常用的三种线性索引技术：		数据量极大并按照先后顺序存储
	1.稠密索引	数据集中的每个记录都对应一个索引项，且索引项按照关键码进行有序排列		
	2.分块索引	是把数据集的记录分成了若干块，块内无序、块间有序		
	3.倒排索引	不是由记录来确定属性值，而是由属性值来确定记录的位置		
树表查找	动态查找表	以树结构存储数据		频繁进行插入和删除数据的操作
	1.二叉查找树	左子树结点一定比其双亲结点小，右子树结点一定比其双亲结点大	最好O(logn)、最坏O(n)	
	2.平衡二叉树	是一种二叉排序树，其中每一个节点的左子树和右子树的高度差至多等于1	O(logn)	

名称	类型	特点	时间复杂度	适用
	3.B树	是一种平衡的多路查找树（每一个结点的孩子数可以多于两个且每一个结点处可以存储多个元素）		数据集非常大
	3.B+树	是一种B树的变形树，将所有叶子结点都链接在一起		带有范围的查找
哈希查找		通过一个哈希函数计算出数据元素的存储地址	O(1)	以空间换时间

Q：了解哪些排序算法，并比较一下，以及适用场景

- 技术点：排序
- 参考回答：代码详见十大经典排序算法最强总结（含JAVA代码实现）

(要求排序结果从小到大)

名称	特点	时间复杂度	空间复杂度	稳定性	适用
冒泡排序	重复走访要排序的数列，一次比较两个元素，若较小元素在后则交换，能看到越小的元素会经由交换慢慢浮到数列的顶端	O(n2)	O(1)	稳定	数据规模较小
简单选择排序	每次都在未排序序列中找最小元素	O(n2)	O(1)	稳定	数据规模较小且对稳定性有要求
直接插入排序	对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入	最好O(n)，平均O(n2)	O(1)	稳定	数据规模较小且待排序列基本有序
希尔排序	将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序	O(nlogn)~O(n2)	O(1)	不稳定	数据规模较大
归并排序	先使每个子序列有序，再使子序列段间有序	O(nlogn)	O(n)	稳定	数据规模较大且对稳定性有要求

名称	特点	时间复杂度	空间复杂度	稳定性	适用
堆排序	近似完全二叉树的结构，子结点的键值或索引总是小于（或大于）其父节点	O(nlogn)	O(1)	不稳定	数据规模较大，相比快排好处是不会出现最坏情况、需要的辅助空间少
快速排序	取一个记录作为枢轴，经过一趟排序将整段序列分为两个部分，使得数轴左侧都小于枢轴、右侧都大于枢轴；再对这两部分继续进行排序使整个序列达到有序	最坏O(n2)，平均O(nlogn)	O(logn)~O(n)	不稳定	数据规模较大且待排序列无序

Q：快排的基本思路是什么？最差的时间复杂度是多少？如何优化？

技术点：排序

参考回答：快速排序使用分治的思想，通过一趟排序将待排序列分割成两部分，其中一部分记录的关键字均比另一部分记录的关键字小；再分别对这两部分记录继续进行排序，以达到整个序列有序的目的。当待排序列有序时会出现最坏时间复杂度O(n2)。几种优化方式：

当待排序序列的长度较小时采用直接插入排序
优化所选取数轴的计算方法，如三数取中
迭代取代递归，效率高
存储数轴值，节省无必要的交换

Q：二叉排序树插入或删除一个节点的过程是怎样的？

技术点：查找
参考回答：

二叉排序树插入操作：先查找该元素是否存在于二叉排列树中并记录其根节点，若没有则比较其和根节点大小后插入相应位置
二叉排序树删除操作：

待删除节点是叶子节点，直接删除即可
待删除节点是仅有左或右子树的节点 ，上移子树即可
待删除节点是左右子树都有的节点 ，用删除节点的直接前驱或直接后继来替换当前节点

Q：什么是红黑树?

技术点：查找
参考回答：红黑树是一种自平衡二叉查找树，包含性质：
节点是红色或黑色
根节点是黑色
叶子节点是黑色
每个红色节点的两个子节点都是黑色
从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

Q：100盏灯问题

- 题目说明：100盏灯问题
- 思路：最后亮着的灯是被拉动次数为奇数次的；由于只有完全平方数的约数个数为奇数，故本题转换为求100内完全平方数的个数

Q：7瓶水1瓶有毒3只老鼠，怎么找有毒的水，再加个条件，必须要求第二天出结果

技术点：查找
思路：若无时间限制，可采用类似二分查找的思路；若要求第二天出结果，用二进制编码的思路
参考回答：

二分查找思路，每次均分两组，每组各取一滴水混合成新溶剂喂给老鼠，继续对导致老鼠死亡的一组水进行同上操作。假如是第1瓶有毒，过程演绎如下，第一只老鼠死于前一堆（mid=(0+6)/2=3，即服用了第1、2、3、4瓶的混合溶剂），第二只老鼠死于前一堆（mid=(0+3)/2=1，即服用了第1、2瓶的混合溶剂），第三只老鼠随意试一瓶，根据服用后状态即可判断有毒的水。
二进制编码思路，对每瓶水二进制编码，所需编码位数正好为三位，将第一位是1的水混为新溶剂喂给第一个老鼠，以此类推，看三只老鼠服用状态（死亡=1，存活=0）得出对应的编码，找到对应的水即可。假如是第1瓶有毒，编码之后，让第一只老鼠服用第4、5、6、7瓶的混合溶剂，第二只老鼠服用第2、3、6、7瓶的混合溶剂，第三只老鼠服用第1、3、5、7瓶的混合溶剂，最终第一只和第二只老鼠存活，第三只老鼠死亡，对应编码为001，对应的水是第一瓶，故此瓶有毒。
**
第一瓶水　　001
第二瓶水　　010
第三瓶水　　011
第四瓶水　　100
第五瓶水　　101
第六瓶水　　110
第七瓶水　　111
**

Q：海量数据问题

- 技术点：海量数据问题
- 思路：分治、哈希、bit、堆
- 参考回答：海量数据处理面试题集锦

Q：（手写算法）二分查找

```
-技术点：查找
-参考代码：
```

```
public static int binarySearch(int[] a, int key) {
    int low, mid, high;
    low = 0;//最小下标
    high = a.length - 1;//最大小标
    while (low <= high) {
        mid = (high + low) / 2;//折半下标
        if (key > a[mid]) {
            low = mid + 1; //关键字比折半值大，则最小下标调成折半下标的下一位
        } else if (key < a[mid]) {
            high = mid - 1;//关键字比折半值小，则最大下标调成折半下标的前一位
        } else {
            return mid; //关键字和折半值相等时返回折半下标
        }
    }
    return -1;
}
```

Q：（手写算法）反转链表

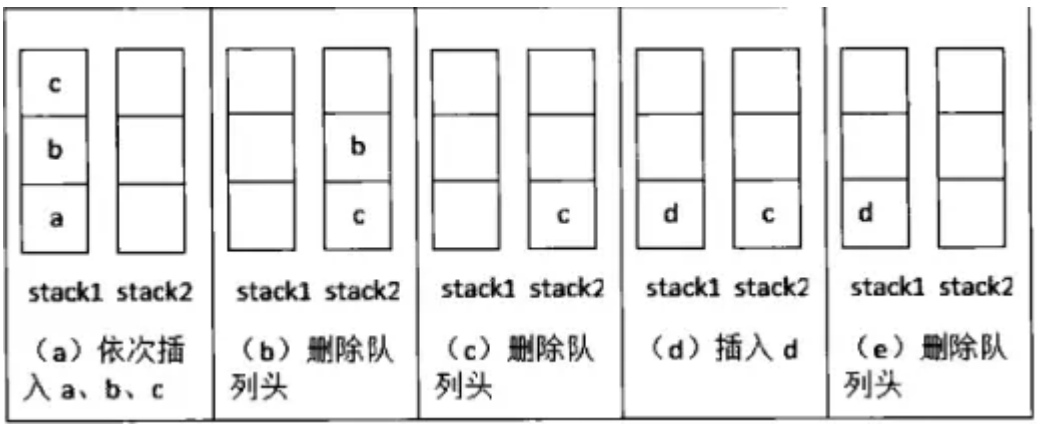
- 技术点：链表
- 思路：
  - 方法1：重复将首节点的下一个节点调整到最前面，如链表1->2->3->4，调整过程为2->1->3->4，3->2->1->4，4->3->2->1
  - 方法2：递归，使链表从尾节点开始指向前一个节点
- 参考代码：

```
//节点类
public class ListNode {
     int val;
     ListNode next = null;
     ListNode(int val) {
          this.val = val;
    }
}
//方法1
public ListNode reverseLinkedList(ListNode head){
```

```
        if (head == null || head.next == null){
            return head;
        }
        ListNode p = new ListNode(-1);//拟一个头节点
    p.next = head;
        ListNode nextNode = head.next;
        while (nextNode != null){
                //后一个节点调整到最前
            head.next = nextNode.next;
            nextNode.next = p.next;
            p.next = nextNode;
            nextNode = head.next;
        }
        return p.next;
}
//方法2，递归
public ListNode reverseLinkedList(ListNode head) {
    if (head == null || head.next == null) {
         return head;
    }
    ListNode pNode = reverseLinkedList(head.next);
    head.next.next = head;
    head.next = null;
    return pNode;
}
```

Q：（手写算法）用两个栈实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。

- 技术点：栈和队列
- 思路：
  - 入队：将元素进栈A
  - 出队：判断栈B是否为空，如果为空，则将栈A中所有元素pop，并push进栈B，栈B出栈， 反之栈B直接出栈



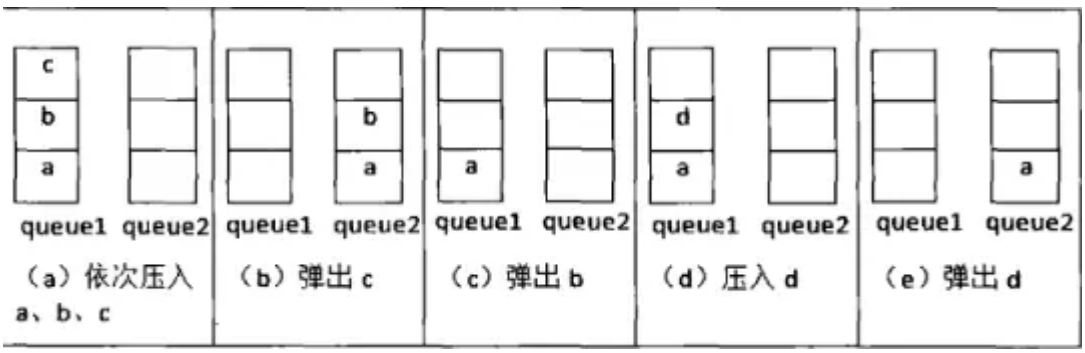
- 参考代码：

```
public class Solution {
    Stack<Integer> stack1 = new Stack<Integer>();
    Stack<Integer> stack2 = new Stack<Integer>();
```

```
    //入队
    public void add(int node) {
        stack1.push(node);
    }
    //出队
    public int poll() {
        if(stack1.empty()&&stack2.empty()){
            throw new RuntimeException("Queue is empty!");
        }
        if(stack2.empty()){
            while(!stack1.empty()){
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }
}
```

Q：（手写算法）用两个队列实现一个栈，完成栈的Push和Pop操作。 队列中的元素为int类型。

- 技术点：栈和队列
- 思路：
  - 入栈：将元素进队列A
  - 出栈：判断队列A中元素的个数是否为1，如果等于1则直接出队列A，否则将队列A中的元素依次出队列并放入队列B、直到队列A中只剩一个原色，然后队列A出队列，再把队列B中的元素以此出队列并放入队列A



- 参考代码：

```
public class Solution {
    Queue<Integer> queue1 = new ArrayDeque<Integer>();
    Queue<Integer> queue2 = new ArrayDeque<Integer>();
    //入栈
    public void push(int node) {
        queue1.add(node);
    }
    //出栈
    public int pop() {
         if(queue1.isEmpty()&&queue2.isEmpty()){
            throw new RuntimeException("Stack is empty!");
         }
         if(queue1.isEmpty()){
```

```
            while(!queue2.isEmpty()){
                queue1.add(queue2.poll());
            }
        }
        if(queue1.size()!=1){
            while(queue1.size()!=1){
                queue2.add(queue1.poll());
            }
        }
        return queue1.poll();
    }
}
```

Q：（手写算法）用三个线程，顺序打印字母A-Z，输出结果是1A、2B、3C、1D 2E...

> - 技术点：线程同步
> - 思路：加锁进行限制，并配合wait()和notifyAll()
> - 参考代码：

```
private static char c = 'A';
private static int i = 0;
public static void main(String[] args) {
    Runnable runnable = new Runnable() {
           public void run() {
              synchronized (this) {//加锁
                try {
                    int threadId = Integer.parseInt(Thread.currentThread().getName());
                    while (i < 26) {
                         if (i % 3 == threadId - 1) {
                             System.out.println(threadId +""+ (char) c++);
                             i++;
                             notifyAll();// 唤醒处于等待状态的线程
                         } else {
                             wait();// 释放当前锁并进入等待状态
                         }
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
              }//执行结束释放当前锁
           }
        };
        Thread t1 = new Thread(runnable, "1");
        Thread t2 = new Thread(runnable, "2");
        Thread t3 = new Thread(runnable, "3");
        t1.start();
        t2.start();
        t3.start();
}
```

Q：（手写算法）如何判断一个链有环，请找出该链表的环的入口结点，否则输出null

- 技术点：链表
- 思路：用p1、p2指向链表头部，然后p1每次走一步、p2每次走两步；显然，当p1和p2第一次相遇时，p2正好比p1多走了一个环；设p2此时走了2x步，p1走了x步，则环长=2x-x=x，因此p1和p2相遇点正好是环的入口点，只要找到p1==p2时的节点即可
- 参考代码：

```
//节点类ListNode同上
public class Solution {
    public ListNode EntryNodeOfLoop(ListNode pHead){
        if(pHead == null || pHead.next == null){
            return null;
        }
        ListNode p1 = pHead;
        ListNode p2 = pHead;
        while(p2 != null && p2.next != null ){
            p1 = p1.next;//p1每次走一步
            p2 = p2.next.next;//p2每次走两步
            if(p1 == p2){
                p2 = pHead;
                while(p1 != p2){
                    p1 = p1.next;
                    p2 = p2.next;
                }
                if(p1 == p2){
                    return p1;//p1和p2相遇点正好是环的入口点
                }
            }
        }
        return null;
    }
}
```

Q：（手写算法）如何判断两条链交叉

- 技术点：链表
- 思路：分别考虑链表上是否存在环的情况
- 参考代码：JAVA 判断两个单链表是否相交并求交点

Q：（手写算法）快速从一组无序数中找到第k大的数（或前k个大的数）

- 技术点：排序
- 思路：利用快排思想，直至找到一个排在第k位置的枢轴，因为左边所有数据都比它大，右边都比它小。
- 参考代码：

```
public class QuickSort {
    public int partition(int[] arr,int low,int high){
        int temp=arr[low];
        while(low<high){
            while(arr[high]<=temp&&high>low){
                high--;
            }
```

```
            arr[low]=arr[high];
            while(arr[low]>=temp&&low<high){
                low++;
            }
            arr[high]=arr[low];
        }
        arr[high]=temp;
        return high;
    }
    public void findK(int k,int[] arr,int low,int high){
        int temp=partition(arr,low,high);
        if(temp==k-1){
            System.out.print("第"+k+"大的数是: "+arr[temp]);
        }else if(temp>k-1){
            findK(k,arr,low,temp-1);
        }else{
            findK(k,arr,temp+1,high);
        }
    }
}
```

Q: （手写算法）从字符串中找出一个最长的不包含重复数字的子字符串的长度。例如在字符串中"arabcacfr"，最长非重复子字符串为"rabc"或"acfr"，长度为4。

> - 技术点：字符串
> - 思路：使用动态规划。先用长度为26的数组positions来存储当前字符上次出现的位置；再定义字符串长度的数组lines表示以当前字母为结尾的最长不含重复字符的子字符串的长度。依次遍历字符串中的字符：
>   - 若当前字符是第一次出现，说明可以直接添加到前一个非重复子字符串，因此lines[i]=lines[i-1]+1；
>   - 若当前字符非第一次出现，需计算当前字符与它上次出现位置之间的距离d：
>   - 若d大于lines[i-1]，说明前一个非重复子字符串中没有包含当前字符，可以添加当前字符到前一个非重复子字符串中，因此lines[i]=lines[i-1]+1；
>   - 若d小于或等于f(i-1)，说明如果加入当前字符会存在重复字符串，需要把上次出现的字符截开，因此lines[i] = d
> - 参考代码：

```
private int findLongestSubstringLength(String string){
        if (string == null || string.equals("")) {
            return 0;
        }
        int maxLength = 0;//最长不重复子字符串的长度
        int[] positions = new int[26];//存储当前字符上次出现的位置，-1表示没有出现过
        for (int i = 0; i < positions.length; i++){
            positions[i] = -1;
        }
        int[] lines = new int[string.length()];//存储以当前字符为尾的最长不重复子字符串的长度
        lines[0]=1;
        positions[string.charAt(0) - 'a']=0;
        for (int i = 1; i < string.length(); i++){
            int prePosition = positions[string.charAt(i) - 'a'];
            if(prePosition>=0){//当前字符非第一次出现
                if((i-prePosition)>lines[i-1]){
```

```
                    lines[i]=lines[i-1]+1;
                }else{
                    lines[i]=i-prePosition;//若加入当前字符会出现重复，需要截断
                }
            }else{//当前字符是第一次出现
                lines[i]=lines[i-1]+1;
            }
            positions[string.charAt(i) - 'a'] = i;
            if(lines[i]>maxLength){
                maxLength=lines[i];
            }
        }
        return maxLength;
}
```

2.剑指offer题解

注：题目源于[牛客网_剑指Offer]

Q：输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。

> - 技术点：链表
> - 思路：通过递归实现链表节点从尾到头依次插入到ArrayList
> - 参考代码：

```
//节点类ListNode同上
import java.util.ArrayList;
public class Solution {
    ArrayList<Integer> arrayList=new ArrayList<Integer>();
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {
        if(listNode!=null){
            this.printListFromTailToHead(listNode.next);//递归
            arrayList.add(listNode.val);
        }
        return arrayList;
    }
}
```

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

> - 技术点：链表
> - 思路：从头节点开始如果下个节点与当前节点重复，则找到第一个与当前不同的节点开始递归，反之，保留当前结点并从下一个结点开始递归
> - 参考代码：

```
//节点类ListNode同上
public class Solution {
    public ListNode deleteDuplication(ListNode pHead)
    {
```

```
        if (pHead == null || pHead.next == null) {
            return pHead;//停止递归条件
        }
        if (pHead.val == pHead.next.val) {
            //如果下个节点与当前节点重复，则找到第一个与当前不同的节点开始递归
            ListNode pNode = pHead.next;
            while (pNode != null && pNode.val == pHead.val) {
                pNode = pNode.next;
            }
            return deleteDuplication(pNode);
        } else {
            //反之，保留当前结点，并从下一个结点开始递归
            pHead.next = deleteDuplication(pHead.next);
            return pHead;
        }
    }
}
```

Q：输入一个链表，输出该链表中倒数第k个结点。

> - 技术点：链表
> - 思路：用p1、p2指向链表头部，先让p1走(k-1)步到达第k个节点，此时p1和p2相隔k个节点；然后p1、p2同时往后移动，当p1到达链尾时，p2所在位置正是倒数第k个节点
> - 参考代码：

```
//节点类ListNode同上
public class Solution {
    public ListNode FindKthToTail(ListNode head,int k) {
        if(head==null||k<=0){
            return null;
        }
        ListNode p1=head;
        ListNode p2=head;
        //p1先到达第k个节点处
        for(int i=1;i<k;i++){
            if(p1.next!=null){
                p1=p1.next;
            }else{
                return null;
            }
        }
        //p1走到链尾时p2正为倒数第k个节点
        while(p1.next!=null){
            p1=p1.next;
            p2=p2.next;
        }
        return last;
    }
}
```

Q：输入两个单调递增的链表，输出两个链表合成后的链表，并保证满足单调不减规则。

> - 技术点：链表
> - 思路：每次都取俩链表中最靠前的俩元素中的更小元素到重组链表中，被取走元素的旧链表后移一个元素继续进行“车轮战”；注意如果只剩一条未结束链表要记得全部插入到重组链表的最后
> - 参考代码：

```
//方法1:非递归
public class Solution {
    public ListNode Merge(ListNode list1,ListNode list2) {
        //新建一个头节点，用来存合并的链表
        ListNode head=new ListNode(-1);
        head.next=null;
        ListNode root=head;
        while(list1!=null&&list2!=null){
            if(list1.val<list2.val){
                head.next=list1;
                head=list1;
                list1=list1.next;
            }else{
                head.next=list2;
                head=list2;
                list2=list2.next;
            }
        }
        //把未结束的链表连接到合并后的链表尾部
        if(list1!=null){
            head.next=list1;
        }
        if(list2!=null){
            head.next=list2;
        }
        return root.next;
    }
}
//方法2:递归
public class Solution {
    public ListNode Merge(ListNode list1,ListNode list2) {
       if(list1 == null){
           return list2;
       }
       if(list2 == null){
           return list1;
       }
       if(list1.val <= list2.val){
           list1.next = Merge(list1.next, list2);
           return list1;
       }else{
           list2.next = Merge(list1, list2.next);
           return list2;
       }
    }
}
```

Q：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。假设输入数组{2,3,4,2,6,2,5,1}及滑动窗口大小3，那么一共存在6个滑动窗口：{[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]}，且他们的最大值分别为{4,4,6,6,6,5}。

> - 技术点：栈和队列
> - 思路：用一个双端队列（头尾都可push和pop）保存下标值，且队头保存当前窗口最大值的下标。遍历数组，先从队头开始把不存在当前窗口的都移除队列；然后从队尾开始把比当前元素小的都移除队列；最后把当前元素的下标值插入到队列中。时间复杂度**O(n)**。
> - 参考代码：

```
public class Solution {
    public ArrayList<Integer> maxInWindows(int [] num, int size)
    {
        ArrayList<Integer> res = new ArrayList<>();
        if(size == 0){
            return res;
        }
        int begin;
        ArrayDeque<Integer> q = new ArrayDeque<>();//双端队列
        for(int i = 0; i < num.length; i++){
            begin = i - size + 1;//表示当前窗口最左元素在原始数组中的下标
            if(q.isEmpty()){
                q.add(i);
            }
            //从队头开始把不存在当前窗口的都移除队列
            while((!q.isEmpty()) &&begin > q.peekFirst()){
                q.pollFirst();
            }
            //从队尾开始把比当前元素小的都移除队列
            while((!q.isEmpty()) && num[q.peekLast()] <= num[i]){
                q.pollLast();
            }
            q.add(i);
            //窗口有效时插入最大值
            if(begin >= 0){
                res.add(num[q.peekFirst()]);
            }
        }
        return res;
    }
}
```

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

> - 技术点：栈
> - 思路：自定义一个最小元素栈，进行push操作时, 如果压入的元素小于等于栈顶元素, 则压入最小元素栈；进行pop操作时, 如果弹出的元素和栈顶元素相等, 就把最小元素栈顶也弹出。
> - 参考代码：

```
public class Solution {
```

```
    Stack<Integer> dataStack = new Stack<Integer>();
    Stack<Integer> minStack = new Stack<Integer>();

    public void push(int node) {
        dataStack.push(node);
        if(minStack.isEmpty() || node <= minStack.peek()){
            minStack.push(node);
        }
    }
    public void pop() {
        if(dataStack.peek()==minStack.peek()){
            minStack.pop();
        }
        dataStack.pop();
    }
    public int top() {
        return dataStack.peek();
    }
    public int min() {
        return minStack.peek();
    }
}
```

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

> - 技术点：栈
> - 思路：借用一个辅助栈。遍历压栈顺序，将元素放入辅助栈，然后判断栈顶元素与弹出顺序第一个元素相等，若相等则出栈并将弹出顺序向后移动一位，直到不相等；最后若辅助栈不为空，说明弹出序列不是该栈的弹出顺序
> - 参考代码：

```
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>();//辅助栈
        int popIndex = 0;//用于标识弹出序列的位置
        for(int i = 0; i< pushA.length;i++){
            s.push(pushA[i]);
            //如果栈不为空且栈顶元素等于弹出序列，则出栈并将弹出序列向后移动一位
            while(!s.empty() &&s.peek() == popA[popIndex]){
                s.pop();
                popIndex++;
            }
        }
        return s.empty();
    }
}
```

Q：翻转单词顺序列，如输入“student. a am I”，输出“I am a student.”

> - 技术点：字符串
> - 思路：先翻转整个句子，再依次翻转每个单词，根据空格来确定单词的起始和终止位置
> - 参考代码：

```
//方法1
public class Solution {
    public String ReverseSentence(String str) {
        char[] chars = str.toCharArray();
        reverse(chars,0,chars.length - 1);//整体翻转
        int blank = -1;
        for(int i = 0;i < chars.length;i++){
            if(chars[i] == ' '){
                int nextBlank = i;
                reverse(chars,blank + 1,nextBlank - 1);//各个单词翻转
                blank = nextBlank;
            }
        }
        reverse(chars,blank + 1,chars.length - 1);//最后一个单词单独进行反转
        return new String(chars);

    }
    public void reverse(char[] chars,int low,int high){
        while(low < high){
            char temp = chars[low];
            chars[low] = chars[high];
            chars[high] = temp;
            low++;
            high--;
        }
    }
}
//方法2:使用java提供的字符串方法split
public class Solution {
    public String ReverseSentence(String str) {
        if(str==null||str.trim().equals("")){
            return str;
        }
        //将字符串按照空格分割成字符串数组
        String[] a = str.split(" ");
        StringBuffer o = new StringBuffer();
        int i;
        //倒序输出字符串数组
        for (i = a.length; i >0;i--){
            o.append(a[i-1]);
            if(i > 1){
                o.append(" ");
            }
        }
        return o.toString();
    }
}
```

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

> - 技术点：字符串
> - 思路：类似哈希表，key-value对英于出现字符-出现次数
> - 参考代码：

```
public class Solution {

    int[] hashtable=new int[256];
    StringBuffer s=new StringBuffer();
    public void Insert(char ch)
    {
        s.append(ch);
        hashtable[ch]++;
    }
    public char FirstAppearingOnce()
    {
        char[] str=s.toString().toCharArray();
        for(char c:str){
          if(hashtable[c]==1)
              return c;
        }
        return '#';
    }
}
```

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

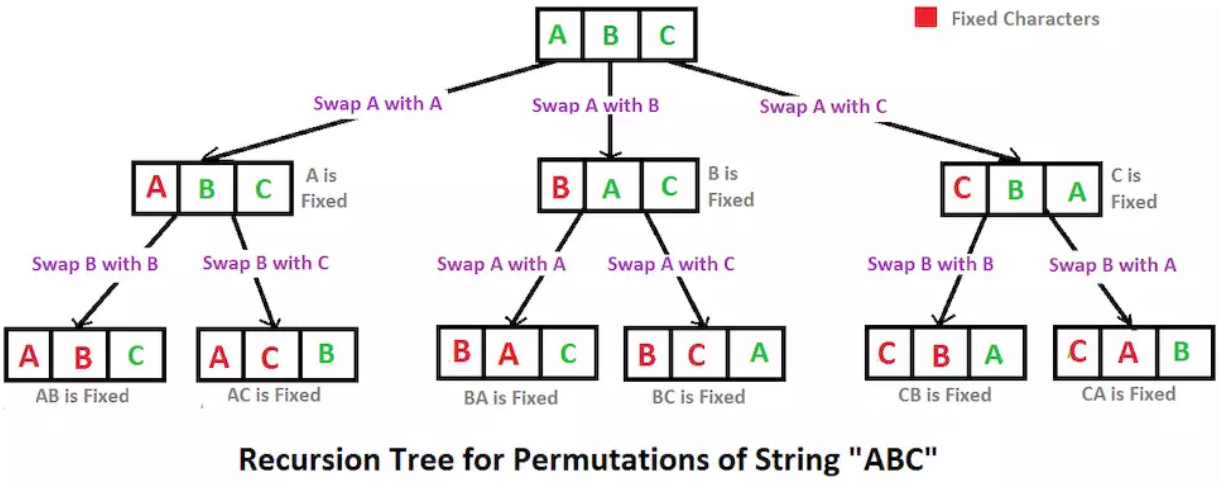
> - 技术点： 字符串
> - 思路：
> - 参考代码：

```
public class Solution {
    public int FirstNotRepeatingChar(String str) {
        HashMap <Character, Integer> map = new HashMap<Character, Integer>();
        for(int i=0;i<str.length();i++){
            if(map.containsKey(str.charAt(i))){
                int time = map.get(str.charAt(i));
                map.put(str.charAt(i), ++time);
            }
            else {
                map.put(str.charAt(i), 1);
            }
        }
        int pos = -1;
        int i=0;
        for(;i<str.length();i++){
```

```
            char c = str.charAt(i);
            if (map.get(c) == 1) {
                return i;
            }
        }
        return pos;
    }
}
```

Q：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abc所能排列出来的所有字符串abc、acb、bac、bca、cab和cba。

> - 技术点：字符串
> - 思路：基于回溯法思想



Recursion Tree for Permutations of String "ABC"

> - 参考代码：

```
public class Solution {
    public ArrayList<String> Permutation(String str) {
        List<String> resultList = new ArrayList<>();
        if(str.length() == 0){
            return (ArrayList)resultList;
        }
        fun(str.toCharArray(),resultList,0);//递归的初始值为（str数组，空的list，初始下标0）
        Collections.sort(resultList);
        return (ArrayList)resultList;
    }
    private void fun(char[] ch,List<String> list,int i){
        if(i == ch.length-1){
            if(!list.contains(new String(ch))){
                list.add(new String(ch));
                return;
            }
        }else{
            //回溯法
            for(int j=i;j<ch.length;j++){
```

```
                swap(ch,i,j);
                fun(ch,list,i+1);
                swap(ch,i,j);
            }
        }
    }
    private void swap(char[] str, int i, int j) {
            if (i != j) {
                char t = str[i];
                str[i] = str[j];
                str[j] = t;
            }
        }
    }
```

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

> - 技术点： 数组
> - 思路：从左下角开始查找，当比关键字小时右移，反之上移
> - 参考代码：

```
public class Solution {
    public boolean Find(int target, int [][] array) {
        int rowCount = array.length;
        int colCount = array[0].length;
        int i,j;
        for(i=rowCount-1,j=0;i>=0&&j<colCount;){
            if(target == array[i][j]){
                return true;
            }else if(target < array[i][j]){
                i--;
                continue;
            }else if(target > array[i][j]){
                j++;
                continue;
            }
        }
        return false;
    }
}
```

Q：找出数组中有出现的次数超过数组长度的一半的数字，如果不存在则输出0。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。

> - 技术点： 数组
> - 思路：在遍历数组时保存两个值，一是数组中一个数字，一是次数；遍历下一个数字时，若它与之前保存的数字相同，则次数加1，否则减1；若次数为0，则保存下一个数字，并将次数置为1；遍历结束后，所保存的数字即为所求；然后再判断它是否符合条件即可。
> - 参考代码：

```
public class Solution {
    public int MoreThanHalfNum_Solution(int [] array) {
        int length=array.length;
        if(array==null||length<=0){
            return 0;
        }
        //找到最后还留住的元素，可能是目标对象
        int result=array[0];
        int times=1;
        for(int i=1;i<length;i++){
            if(times==0){
                result=array[i];
                times=1;
            }else{
                if(array[i]==result){
                    times++;
                }else{
                    times--;
                }
            }
        }
        //对可能的目标对象进行验证
        times=0;
        for(int i=0;i<length;i++){
            if(result==array[i]){
                times++;
            }
       }
       if(times*2<=length){
           result=0;
       }
       return result;
    }
}
```

Q：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。

> - 技术点： 数组
> - 思路：通过一个比较器对数据进行两两比较，规则为若a + b<b+a，则a排在前
> - 参考代码：

```
public class Solution {
    public String PrintMinNumber(int [] numbers) {
        String s="";
        ArrayList<Integer> list= new ArrayList<Integer>();
        for(int i=0;i<numbers.length;i++){
            list.add(numbers[i]);
        }
        Collections.sort(list, new Comparator<Integer>(){
            public int compare(Integer str1,Integer str2){
```

```
                String s1=str1+""+str2;
                String s2=str2+""+str1;
                return s1.compareTo(s2);//如果s1>s2，则str1和str2交换，反之不变
            }
        });
        for(int j:list){
            s+=j;
        }
        return s;
    }
}
```

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

> - 技术点：数组
> - 思路：用两个指针一头一尾；若加和比S大，则尾指针下移一个，若比S小，则头指针上移一个；最先找到的两个数，差最大乘积最小
> - 参考代码：

```
public class Solution {
    public ArrayList<Integer> FindNumbersWithSum(int [] array,int sum) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        if (array == null || array.length < 2) {
            return list;
        }
        int i=0,j=array.length-1;
        while(i<j){
            if(array[i]+array[j]==sum){
                list.add(array[i]);
                list.add(array[j]);
                return list;
            }else if(array[i]+array[j]>sum){
                j--;
            }else{
                i++;
            }
        }
        return list;
    }
}
```

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

> - 技术点：代码的完整性
> - 思路：
>   - 用排列：插排思想，如果当前数是奇数，就一直往前找，遇到偶数就往它前面插

    - 空间换时间：先统计奇数的个数；再新建一个等长数组，设置两个指针，奇数指针从0开始，偶数指针从奇数个数的末尾开始
- 参考代码：

```
//方法1:时间复杂度O(n2)、空间复杂度O(1)
public class Solution {
    public void reOrderArray(int [] array) {
        if(array.length==0||array.length==1) return;
        for(int i=1;i<array.length;i++){
            int temp = array[i];
            if(array[i] % 2 == 1){
                int j = i;
                while(j >= 1 && array[j-1] % 2 == 0){
                    array[j] = array[j-1];
                    j--;
                }
                array[j] = temp;
            }
        }
    }
}
//方法2:时间复杂度O(n)、空间复杂度O(n)
public class Solution {
    public void reOrderArray(int [] array) {
        if(array.length==0||array.length==1) return;
        int oddCount=0,oddBegin=0;
        int[] newArray=new int[array.length];
        for(int i=0;i<array.length;i++){
            if((array[i]&1)==1) oddCount++;//统计奇数个数
        }
        for(int i=0;i<array.length;i++){
            if(array[i]%2==1){
                newArray[oddBegin++]=array[i];//奇数项从0开始
            }else{
                newArray[oddCount++]=array[i];//偶数项从奇数个数的末尾开始
            }
        }
        for(int i=0;i<array.length;i++){
            array[i]=newArray[i];
        }
    }
}
```

Q：实现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。

- 技术点： 树
- 思路：BFS
- 参考代码：

```
//节点类TreeNode
public class TreeNode {
```

```
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;
    public TreeNode(int val) {
        this.val = val;
    }
public class Solution {
    ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {
        ArrayList<ArrayList<Integer>> alist = new ArrayList<ArrayList<Integer>>();// 结果集
        if (pRoot == null) return alist;
        Queue<TreeNode> queue = new LinkedList<TreeNode>();// 队列
        queue.add(pRoot);
        while (!queue.isEmpty()) {
            ArrayList<Integer> list = new ArrayList<Integer>();// 每层结果
            int count=queue.size();//每层节点个数
            for (int i = 0; i < count; i++) {
                TreeNode node = queue.peek();
                list.add(node.val);
                if (node.left != null)  queue.add(node.left);
                if (node.right != null)  queue.add(node.right);
                queue.poll();
            }
            alist.add(list);
        }
        return alist;
    }
}
```

Q：给定一棵二叉搜索树，请找出其中的第k小的结点。例如（5，3，7，2，4，6，8）中第三小结点的值为4。

> - 技术点： 树
> - 思路：二叉搜索树的中序遍历结果正好是从小到大排序好的，按照中序遍历顺序找第k个节点
> - 参考代码：

```
//节点类TreeNode同上
public class Solution {
    int index = 0; //计数器
    TreeNode KthNode(TreeNode root, int k){
        if(root != null){
            TreeNode node = KthNode(root.left,k);
            if(node != null) return node;//注意需要判断是否为空，否则如果找到符合要求的节点只能返回到上一层，而不能返回到顶层，使得输出结果为null
            index ++;
            if(index == k) return root;
            node = KthNode(root.right,k);
            if(node != null) return node;//理由同上
        }
        return null;
    }
}
```

Q：输入一棵二叉树，判断该二叉树是否是平衡二叉树。

> - 技术点：树
> - 思路：从下往上遍历，如果子树是平衡二叉树，则返回子树的高度，反之直接停止遍历，这样至多只对每个结点访问一次
> - 参考代码：

```
//节点类TreeNode同上
public class Solution {
    public boolean IsBalanced_Solution(TreeNode root) {
        return getDepth(root) != -1;
    }
    private int getDepth(TreeNode root) {
        if (root == null) return 0;
        int left = getDepth(root.left);
        if (left == -1) return -1;
        int right = getDepth(root.right);
        if (right == -1) return -1;
        return Math.abs(left - right) > 1 ? -1 : 1 + Math.max(left, right);
    }
}
```

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

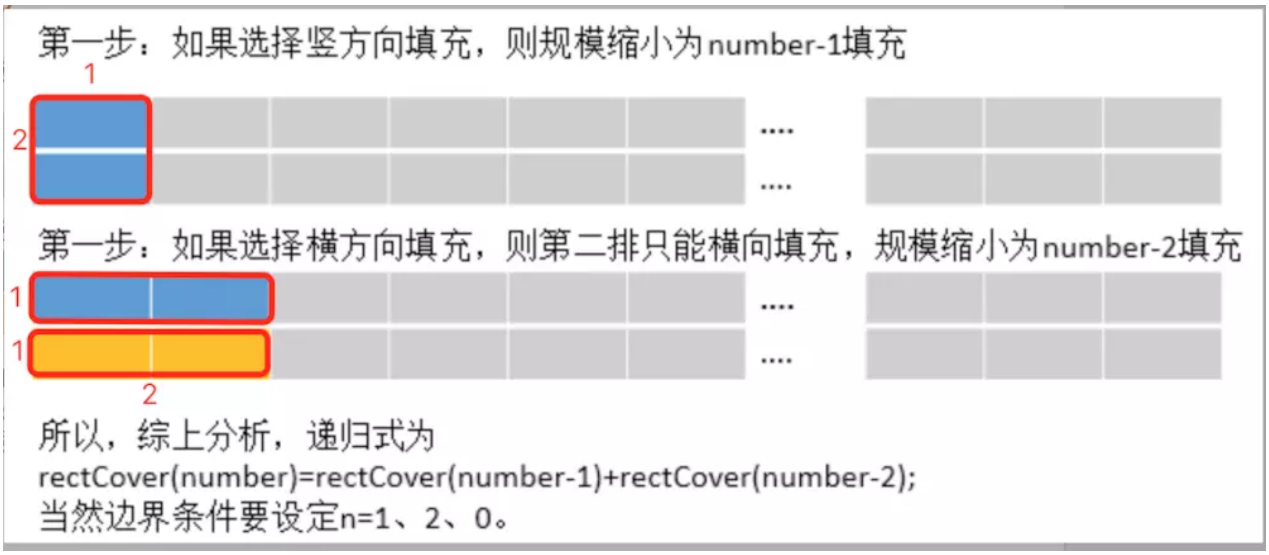
> - 技术点： 递归和循环
> - 思路：对于第n个台阶来说，只能从n-1或n-2的台阶跳上来，因此跳到n阶的跳法=跳到n-1阶的方案+跳到n-2阶的方案，即F(n) = F(n-1) + F(n-2)，是个斐波拉契数序列
> - 参考代码：

```
//递归法
public class Solution {
    public int JumpFloor(int target) {
        if (target <= 2) {
            return target;
        } else {
            return  JumpFloor(target-1)+JumpFloor(target-2);
        }
    }
}
//迭代法
public class Solution {
    public int JumpFloor(int target) {
        if (target <= 2) {
            return target;
        }
        int f1=1;
        int f2=2;
        int f=0;
        for(int i=3;i<=target;i++){
            f=f1+f2;
```

```
            f1=f2;
            f2=f;
        }
        return f;
    }
}
```

Q：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?

- 技术点：递归和循环
- 思路：2*n*的覆盖方案=2(n-1)的覆盖方案+2*(n-2)的覆盖方案，即F(n) = F(n-1) + F(n-2)，是个斐波拉契数序列



- 参考代码：代码同上

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

- 技术点：知识迁移能力
- 思路：DFS
- 参考代码：

```
public class Solution {
    public int movingCount(int threshold, int rows, int cols){
        if(rows <= 0 || cols <= 0 || threshold < 0) return 0;
        boolean[] visited = new boolean[rows * cols];
        return dfs(threshold,rows,cols,visited,0,0);
    }
    private  int dfs(int threshold, int rows, int cols, boolean[] visited, int x, int y) {
        //满足终止条件的格子：不属于方格、已经标记过、位数和大于阈值
        if(x < 0 || x >= cols || y < 0 || y >= rows || getDigitSum(x) + getDigitSum(y) >
```

```
threshold || visited[x + y * cols]) return 0;
        visited[x + y * cols] = true;
        return 1 + dfs(threshold, rows, cols, visited, x, y - 1)
                 + dfs(threshold, rows, cols, visited, x + 1, y)
                 + dfs(threshold, rows, cols, visited, x, y + 1)
                 + dfs(threshold, rows, cols, visited, x - 1, y);
    }
    //位数和
    private int getDigitSum(int i) {
        int sum = 0;
        while(i > 0) {
            sum += i % 10;
            i /= 10;
        }
        return sum;
    }
}
```

Q：求出从1 到 n 中1出现的次数。

> - 技术点： 时间效率
> - 思路：从1到n整数中1出现的次数：O(logn)算法

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

> - 技术点： 时间空间效率的平衡
> - 思路：对于第i个数，它一定是之前已存在数的2倍、3倍或5倍
> - 参考代码：

```
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;
        for (int i = 1; i < index; ++i){
            res[i] = Math.min(res[t2] * 2, Math.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];
    }
}
```

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

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

- 技术点： 画图让抽象形象化
- 思路：用左上和右下的坐标定位出一次要旋转打印的数据，一次旋转打印结束后，往对角分别前进和后退一个单位。注意单行或者单列的情况。
- 参考代码：

```
public class Solution {
    public ArrayList<Integer> printMatrix(int [][] matrix) {
        int row = matrix.length;
        int col = matrix[0].length;
        ArrayList<Integer> res = new ArrayList<>();
        if (row == 0 || col == 0){
            return res;
        }
        int left = 0, top = 0, right = col - 1, bottom = row - 1; // 定义四个关键变量，表示左上和右下的打印范围
        while (left <= right && top <= bottom) {
            for (int i = left; i <= right; ++i){
                res.add(matrix[top][i]);//从左到右
            }
            for (int i = top + 1; i <= bottom; ++i){
                res.add(matrix[i][right]);//从上到下
            }
            if (top != bottom)
                for (int i = right - 1; i >= left; --i){
                    res.add(matrix[bottom][i]);//从右到左
                }
            if (left != right)
                for (int i = bottom - 1; i > top; --i){
                    res.add(matrix[i][left]);//从下到上
                }
            left++;top++;right--;bottom--;//缩小一圈，重新定位打印范围
        }
        return res;
    }
}
```