题目 二维数组中的查找

热点指数 147187 通过率 23.71% 考点 数组

具体题目

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

解决方案

一种是： 把每一行看成有序递增的数组， 利用二分查找， 通过遍历每一行得到答案， 时间复杂度是nlogn

```
public class Solution {
    public boolean Find(int [][] array,int target) {


        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(target>array[i][mid])
                    low=mid+1;
                else if(target<array[i][mid])
                    high=mid-1;
                else
                    return true;
            }
        }
        return false;
    }
}
```

另外一种思路是： 利用二维数组由上到下，由左到右递增的规律， 那么选取右上角或者左下角的元素a[row][col]与target进行比较， 当target小于元素a[row][col]时，那么target必定在元素a所在行的左边, 即col--； 当target大于元素a[row][col]时，那么target必定在元素a所在列的下边, 即row++；

```
public class Solution {
    public boolean Find(int [][] array,int target) {
        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;
    }
}
```

Java版本

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

最佳答案：没有之一。思路：首先我们选择从左下角开始搜寻，(为什么不从左上角开始搜寻，左上角向右和向下都是递增，那么对于一个点，对于向右和向下会产生一个岔路；如果我们选择从左下脚开始搜寻的话，如果大于就向右，如果小于就向下)。

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


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


        return false;
    }
}
```

```
public class Solution {
    public boolean Find(int [][] array,int target) {
        for(int[] i : array){
            for(int j : i){
                if(j==target)return true;
            }
        }
        return false;
    }
}
```

```
public class Solution {
    public boolean Find(int target, int [][] array) {
        boolean flag = false;
        int x=0;
        int y=0;
        for(int i=0;i < array.length;i++) {
            for(int j=0;j<array[i].length;j++) {
                if(target==array[i][j]){
                    flag = true;
                    break;
                }
            }
        }
        if(flag) {
            System.out.println("exist!"+target+",location:"+"array["+x+"]["+y+"]");
        }
        return flag;
    }
}
```

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

题目 替换空格

考点 字符串 热点指数 136618 通过率 24.44%

具体题目

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

解决方案

查看全部 226 C/C++ 桥deer 思路：从前向后记录‘ ’数目，从后向前替换‘ ’。 重点：从后向前替换的时候的技巧 例如：“we are lucky” 0 1 2 3 4 5 6 7 8 9 10 11 w e a r e l u c k y 可以得知count=2;//空格的个数。 所以在替换的时候7~11的字母要向后移动count×2个位置，3~5字母要向后移动（count-1）×2个位置。 所以得到 ： 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 w e a r e l u c k y w e a r a r e u c k l u c k y 在替换的时候直接在空格处写入%20了 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 w e a r e l u c k y w e % 2 0 a r e % 2 0 l u c k y

/* 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。 */

```
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() + spacenum*2;//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
        for(;indexold>=0 && indexold<newlength;--indexold){
                if(str.charAt(indexold) == ' '){  //
                str.setCharAt(indexnew--, '0');
                str.setCharAt(indexnew--, '2');
                str.setCharAt(indexnew--, '%');
                }else{
                    str.setCharAt(indexnew--, str.charAt(indexold));
                }
        }
        return str.toString();
    }
}
```

```
public class Solution {
    public String replaceSpace(StringBuffer str) {
        StringBuffer out=new StringBuffer();
        for (int i = 0; i < str.toString().length(); i++) {
            char b=str.charAt(i);
            if(String.valueOf(b).equals(" ")){
                out.append("%20");
            }else{
                out.append(b);
            }
        }
        return out.toString();
```

```
    }
}
```

```
public class Solution {
    public String replaceSpace(StringBuffer str) {
        String sti = str.toString();
        char[] strChar = sti.toCharArray();
        StringBuffer stb = new StringBuffer();
        for(int i=0;i<strChar.length;i++){
            if(strChar[i]==' '){
                stb.append("%20");
            }else{
                stb.append(strChar[i]);
            }
        }
        return stb.toString();
    }
}
```

```
public class Solution {
    public String replaceSpace(StringBuffer str) {
        StringBuffer st = new StringBuffer();
        for(int i = 0; i < str.length() ; i++){
            char s = str.charAt(i);
            if(String.valueOf(s).equals(" ")){
                st.append("%20");
            }
            else{
                st.append(s);
            }
        }
        return st.toString();
    }
}
```

```
public class Solution {
    public String replaceSpace(StringBuffer str) {
        String s = str.toString();
        if(str==null)
            return s;
         char []ss=s.toCharArray();
        StringBuffer sb = new StringBuffer();
        for(int i=0;i<ss.length;i++)
            {
            if(ss[i]==' ')
                {
                 sb.append("%20");
            }
           else
               sb.append(ss[i]);
        }
        return sb.toString();
    }
}
```

```
public class Solution {
     public String replaceSpace(StringBuffer str) {
         if(str==null){
             return null;
         }else{
             return str.toString().replaceAll(" ","%20");
         }
     }
  }
```

题目 从尾到头打印链表

考点 链表 热点指数 118871 通过率 24.44%

具体题目

输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。

有三种思路，第一就是利用栈先入后出的特性完成，第二就是存下来然后进行数组翻转。 第三是利用递归。

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

        while(listNode != null){
            list.add(listNode.val);
            listNode = listNode.next;
        }

        Collections.reverse(list);//使用Collections的reverse方法，直接将list反转
        return list;
    }
}
```

最佳代码：代码思路借助栈，遍历的时候入栈，由于数据结构中栈的特点是先进后出，所以遍历的过程中压栈，推栈，完了弹栈加到ArrayList中。有两个容易出错的地方：第一，第一次测试用例，{}返回[ ],null是null，而[ ]是new ArrayList（ ）但是没有数据。第二，遍历stack用的方法是！stak.isEmpty()方法，而不是for循环size遍历。。 /**

- public class ListNode {
- int val;
- ListNode next = null; *
- ListNode(int val) {
- this.val = val;
- }
- } * */

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

```
    }
}
```

题目 重建二叉树

考点 树 热点指数 84163 通过率 22.82%

具体题目

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。

```
import java.util.*;
/**

 * Definition for binary tree
 * public class TreeNode {
 * int val;
 * TreeNode left;
 * TreeNode right;
 * TreeNode(int x) { val = x; }
 * }
    */
    public class Solution {
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
       if(pre.length == 0||in.length == 0){
            return null;
        }
        TreeNode node = new TreeNode(pre[0]);
        for(int i = 0; i < in.length; i++){
            if(pre[0] == in[i]){
                node.left = reConstructBinaryTree(Arrays.copyOfRange(pre, 1, i+1),
Arrays.copyOfRange(in, 0, i));
                node.right = reConstructBinaryTree(Arrays.copyOfRange(pre, i+1, pre.length),
Arrays.copyOfRange(in, i+1,in.length));
            }
        }
        return node;
    }
```

1.先求出根节点（前序序列第一个元素）。
2.将根节点带入到中序遍历序列中求出左右子树的中序遍历序列。
3.通过左右子树的中序序列元素集合带入前序遍历序列可以求出左右子树的前序序列。
4.左右子树的前序序列第一个元素分别是根节点的左右儿子
5.求出了左右子树的4种序列可以递归上述步骤

```
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 inleft,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,inright);
            }
        }
        return root;
    }
}
```

```
public class Solution {
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        int i=0;
        if(pre.length!=in.length||pre.length==0||in.length==0)
            return null;
        TreeNode root = new TreeNode(pre[0]);
        while(in[i]!=root.val)
            i++;
        int[] preLeft = new int[i];
        int[] inLeft = new int[i];
        int[] preRight = new int[pre.length-i-1];
        int[] inRight = new int[in.length-i-1];
        for(int j = 0;j<in.length;j++) {
            if(j<i) {
                preLeft[j] = pre[j+1];
                inLeft[j] = in[j];
            } else if(j>i) {
                preRight[j-i-1] = pre[j];
                inRight[j-i-1] = in[j];
            }
        }
        root.left = reConstructBinaryTree(preLeft,inLeft);
        root.right = reConstructBinaryTree(preRight,inRight);
        return root;
    }
}
```

```
/* 先序遍历第一个位置肯定是根节点node，
  中序遍历的根节点位置在中间p，在p左边的肯定是node的左子树的中序数组，p右边的肯定是node的右子树的中序数组
  另一方面，先序遍历的第二个位置到p，也是node左子树的先序子数组，剩下p右边的就是node的右子树的先序子数组
  把四个数组找出来，分左右递归调用即可
*/
```

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
    */
    import java.util.ArrayList;
    import java.util.List;
    public class Solution {
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        ArrayList<Integer> preList = new ArrayList<Integer>(pre.length);
        ArrayList<Integer> inList = new ArrayList<Integer>(in.length);
        for (int i : pre)
            preList.add(i);
        for (int i : in)
            inList.add(i);
        return getRootNode(preList, inList);
    }
```

```
    private TreeNode getRootNode(List<Integer> preList, List<Integer> inList) {
        if (preList.size() == 0)
            return null;
        int rootVal = preList.get(0);
        TreeNode root = new TreeNode(rootVal);
        int index = inList.indexOf(rootVal);
        List<Integer> leftInList = inList.subList(0, index);
        List<Integer> rightInList = inList.subList(index+1, inList.size());
        List<Integer> leftPreList = preList.subList(1, leftInList.size()+1);
        List<Integer> rightPreList = preList.subList(preList.size()
                - rightInList.size(), preList.size());


        root.left = getRootNode(leftPreList, leftInList);
        root.right = getRootNode(rightPreList, rightInList);
        return root;
    }
    }
```

```
/* 先序遍历第一个位置肯定是根节点node，
   中序遍历的根节点位置在中间p，在p左边的肯定是node的左子树的中序数组，p右边的肯定是node的右子树的中序数组
   另一方面，先序遍历的第二个位置到p，也是node左子树的先序子数组，剩下p右边的就是node的右子树的先序子数组
   把四个数组找出来，分左右递归调用即可
*/
```

题目描述 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。 解题思路 因为是树的结构，一般都是用递归来实现。 用数学归纳法的思想就是，假设最后一步，就是root的左右子树都已经重建好了，那么我只要考虑将root的左右子树安上去即可。 根据前序遍历的性质，第一个元素必然就是root，那么下面的工作就是如何确定root的左右子树的范围。 根据中序遍历的性质，root元素前面都是root的左子树，后面都是root的右子树。那么我们只要找到中序遍历中root的位置，就可以确定好左右子树的范围。 正如上面所说，只需要将确定的左右子树安到root上即可。递归要注意出口，假设最后只有一个元素了，那么就要返回。 我的答案 import java.util.Arrays;

```
public class Solution {
    public TreeNode reConstructBinaryTree(int [] pre,int [] in) {
        //数组长度为0的时候要处理
        if(pre.length == 0){
            return null;
        }
        int rootVal = pre[0];
        //数组长度仅为1的时候就要处理
        if(pre.length == 1){
            return new TreeNode(rootVal);
        }
        //我们先找到root所在的位置，确定好前序和中序中左子树和右子树序列的范围
        TreeNode root = new TreeNode(rootVal);
        int rootIndex = 0;
        for(int i=0;i<in.length;i++){
            if(rootVal == in[i]){
                rootIndex = i;
                break;
            }
        }
        //递归，假设root的左右子树都已经构建完毕，那么只要将左右子树安到root左右即可
        //这里注意Arrays.copyOfRange(int[],start,end)是[)的区间
        root.left =
reConstructBinaryTree(Arrays.copyOfRange(pre,1,rootIndex+1),Arrays.copyOfRange(in,0,rootIndex
));
```

```
        root.right =
reConstructBinaryTree(Arrays.copyOfRange(pre,rootIndex+1,pre.length),Arrays.copyOfRange(in,ro
otIndex+1,in.length));
        return root;
    }
}
```

题目 用两个栈实现队列

考点 栈和队列 热点指数 94419 通过率 35.74%

具体题目

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

栈是只能进出栈都在栈顶，而队列是队尾进队，队头出队。所以很明显，直接用栈是不行的。
　然后进队列是在队尾，而栈进栈也可以看做是栈尾。所以很明显，进队列操作和进栈操作是一样的，所以只需要使用一个栈就行了，这里我们假设使用stack1。
所以接下来我们只要考虑如何使用两个栈使得出队操作是输出栈头部元素。输出头部元素，我们必须将除了第一个进栈的元素外的其他元素全部出栈，比如进栈操作12345，必须将2345出栈。我们很容易想到将2345存在另一个栈中(stack2.push(stack1.pop()))，此时stack2顺序是5432。然后将1出栈赋值给一个变量(result = stack1.pop())，这个变量就是return的值。然后再讲stack2全部出栈赋值给stack1，此时stack1中元素就为2345(stack1.push(stack2.pop()))。
但是这里最容易出错的地方：其实是循环，一开始我使用的是for(int i = 0;i < stack1.size();i++)，发现答案是错误的。因为stack1.pop()之后，satck1.size()也发生了变化。导致执行了2次循环后就满足条件了。因此，在stack1出栈操作之前，使用一个变量存储stack1的初始长度。

```
    public void push(int node) {
        stack1.push(node);
    }

    public int pop() {
        if(stack2.empty()){
            while(!stack1.empty()){
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }
```

题目描述 用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。 解题思路 队列是先进先出，栈是先进后出，如何用两个栈来实现这种先进先出呢？ 其实很简单，我们假设用stack1专门来装元素，那么直接stack1.pop肯定是不行的，这个时候stack2就要发挥作用了。 我们的规则是：只要stack2中有元素就pop，如果stack2为空，则将stack1中所有元素倒进satck2中，就是说，新元素只进stack1，元素出来只从stack2出来。 这样子，就能保证每次从stack2中pop出来的元素就是最老的元素了。 我的答案

```
import java.util.Stack;
public class Solution{
    //负责装元素
    Stack<Integer> stack1 = new Stack<Integer>();
    //负责出元素
    Stack<Integer> stack2 = new Stack<Integer>();
    public void push(int node) {
        stack1.push(node);
    }
    //主要思想是：stack2有元素就pop，没有元素就将stack1中所有元素倒进来再pop
    public int pop() throws Exception{
        if(!stack2.isEmpty()){
            int node = stack2.pop();
            return node;
        }else{
            if(stack1.isEmpty()){
                throw new Exception("no valid element");
            }
            while(!stack1.isEmpty()){
                stack2.push(stack1.pop());
```

```
        }
        return stack2.pop();
    }
}
```

题目 旋转数组的最小数字

考点 查找和排序 热点指数 89987 通过率 31.97%

具体题目

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

```
public class Solution {
 /*
 * 传进去旋转数组，注意旋转数组的特性：
 * 1.包含两个有序序列
 * 2.最小数一定位于第二个序列的开头
 * 3.前序列的值都>=后序列的值
 * 定义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。
 * ^_^这个旋转思想是很经典的
 * 旋转数组实例：
 * {123456}旋转后{456123}
 */

 //用到了快速排序的快速定位范围的思想，
 public int minNumberInRotateArray(int [] array) {
    if(array==null||array.length==0)
    {  return 0;
    }
    int low=0;
    int up=array.length-1;
    int mid=low;

 // 当low和up两个指针相邻时候，就找到了最小值，也就是
 //右边序列的第一个值

 while(array[low]>=array[up]){
 if(up-low==1){
 mid=up;
 break;
 }
 //如果low、up、mid下标所指的值恰巧相等
 //如：{0,1,1,1,1}的旋转数组{1,1,1,0,1}
 if(array[low]==array[up]&&array[mid]==array[low])
 return MinInOrder(array);
    mid=(low+up)/2;
 //这种情况，array[mid]仍然在左边序列中
 if(array[mid]>=array[low])
 low=mid;//注意，不能写成low=mid+1；
 //要是这种情况，array[mid]仍然在右边序列中
 else if(array[mid]<=array[up])
 up=mid;
 }

 return array[mid];

    }
 private int MinInOrder(int[] array) {
 // TODO Auto-generated method stub
 int min =array[0];
 for(int i=1;i<array.length;i++){
```

```
 if(array[i]<min){
 min=array[i];

 }
 }
 return min;
 }
 public static void main(String[] args) {

 }
}
```

```
public class 旋转数组的最小数字 {
    public int minNumberInRotateArray(int[] array) {
        if (array == null || array.length == 0) {
            return -1;
        }
        int len = array.length;
        int index1 = 0;
        int index2 = len - 1;
        int indexMid = index1;
        while (array[index1] >= array[index2]) {
            if (index1 == index2 - 1) {
                indexMid = index2;
                break;
            }
            indexMid = index1 + (index2 - index1) / 2;
            if (array[index1] <= array[indexMid]) {
                index1 = indexMid;
            } else if (array[indexMid] <= array[index2]) {
                index2 = indexMid;
            }
        }
        return array[indexMid];
    }
}
```

```
import java.util.ArrayList;
public class Solution {
    public int minNumberInRotateArray(int [] array) {
        if(array.length==0)
            return 0;
        else
            return partition(array,0,array.length-1);
    }
    //递归的目的是寻找乱序的子数组
    private int partition(int [] array,int start,int end){
        if( array[start] < array[end] || start == end )  //如果第一个元素小于最后一个元素，说明数组从头到尾都是非减的；如果只剩下一个元素，则直接返回
            return array[start];
        else {
            int mid=start+(end-start)/2;
            if( array[mid] < array[end]){    //如果中间值下于最后的值，说明后半部分为非减序列，所以在前半部分继续寻找；
                                //另外，之所以是mid而不是mid-1，是为了防止出现越界的情况,例如，array=[3,4],那么start=0，mid=0,end=1; (mid-1)等于-1,不可行
                return partition(array,start,mid);
            }else if(array[mid] == array[end]){    // 如果array=[1,0,1,1,1]或者[1,1,1,0,1]，那没办法判断乱序子数组的位置，所以只能削减一步
```

```
                return partition(array,start,end-1);
            }else{                  //如果中间值大于最后值，那么说明乱序的部分在后半段，所以在后半段寻找。可以使用mid+1是因为，中间值都比最后值大了，那还要它干嘛？
                return partition(array,mid+1,end);
            }
        }
    }
}
```

题目 斐波那契数列

考点 递归和循环 热点指数 99449 通过率 29.53%

具体题目

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

```
/*
整体思路：考虑负数，大数，算法的复杂度，空间的浪费
*/
public class Solution {
    public int Fibonacci(int n) {
        //方法1：用递归，系统会让一个超大的n来让Stack Overflow，所以
        //递归就不考虑了

        //使用迭代法，用fn1和fn2保存计算过程中的结果，并复用起来
        int fn1 = 1;
        int fn2 = 1;

        //考虑出错情况
        if (n <= 0) {
            return 0;
        }
        //第一和第二个数直接返回
        if (n == 1 || n == 2) {
            return 1;
        }
        //当n>=3时，走这里，用迭代法算出结果
        //这里也说明了，要用三个数操作的情况，其实也可以简化为两
        //个数，从而节省内存空间
        while (n-- > 2) {
            fn1 += fn2;
            fn2 = fn1 - fn2;
        }
        return fn1;
    }
}


//就记录前面计算的n-1和n-2的值嘛
public class Solution {
   public static int Fibonacci(int n) {
         if (n <= 1)
             return n;
         int res = 0;
         int n1 = 0;
         int n2 = 1;
         for (int i=2; i<=n; i++){
             res = (n1 + n2);
             n1 = n2;
             n2 = res;
         }
        return res;
     }
}


  /*
```

```
  *方法一：递归，不考虑，有大量的重复计算，会导致内存溢出
  /*
  public class Solution {
      public int Fibonacci(int n) {
          if(n<=0) {
              return 0;
          }
if(n==1) {
              return 1;
          }
          return Fibonacci(n-2)+Fibonacci(n-1);
      }
  }
  */
  /*
  *方法二：使用迭代法，用fn1和fn2保存计算过程中的结果，并复用起来
  */
  public class Solution {
      public int Fibonacci(int n) {
          int fn1 = 1;
          int fn2 = 1;
          if(n <= 0 ) {
              return 0;
          }
          if(n==1 || n==2) {
              return 1;
          }
          while(n>2) {
              fn1 += fn2;
              fn2 = fn1-fn2;
              n--;
          }
          return fn1;
      }
  }
```

题目 跳台阶

考点 递归和循环 热点指数 98398 通过率 34.40%

具体题目

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）。

可以用动态规划来求解该题 跳到第n个台阶，只有两种可能 从第n-1个台阶跳1个台阶 从第n-2个台阶跳2个台阶 只需求出跳到第n-1个台阶和第n-2个台阶的可能跳法即可 F（n）:n个台阶的跳法 递推公式：F（n）=F（n-1）+F（n-2）不难发现这是一个斐波那契数列 起始条件为F（0）=1，F（1）=1 解法一：自底向上，使用迭代

```
public class Solution {
  public int JumpFloor(int target) {
      if(target==0)
          return 1;
      if(target==1)
      return 1;
  int si_1=1;
  int si_2=1;
  int result=0;
  for(int i=2;i<=target;i++){
      result=si_1+si_2;
      si_2=si_1;
      si_1=result;
  }
  return result;
  }
}
```

解法二：自顶向下，使用递归

```
public class Solution {
  public int JumpFloor(int target) {
if(target==1)
          return 1;
      else if(target==2)
          return 2;
      return JumpFloor(target-1)+JumpFloor(target-2);
  }
}
```

/** *1.假设当有n个台阶时假设有f(n)种走法。 *2.青蛙最后一步要么跨1个台阶要么跨2个台阶。 *3.当最后一步跨1个台阶时即之前有n-1个台阶，根据1的假设即n-1个台阶有f(n-1)种走法。 *4. 当最后一步跨2个台阶时即之前有n-2个台阶，根据1的假设即n-2个台阶有f(n-2)种走法。 *5.显然n个台阶的走法等于前两种情况的走法之和即f(n)=f(n-1)+f(n-2)。 *6.找出递推公式后要找公式出口，即当n为1、2时的情况，显然n=1时f(1)等于1，f(2)等于2 *7. 1, (n=1) *f(n) = 2, (n=2)

- f(n-1)+f(n-2) ,(n>2,n为整数) */

```
public class Solution {
    public int JumpFloor(int target) {
int fn1 = 1;
        int fn2 = 2;
```

```
        if(target <= 0) {
            return 0;
        }
        if(target == 1) {
            return fn1;
        }
        if(target == 2) {
            return fn2;
        }

        while(target>2) {
            fn2 += fn1;
            fn1 = fn2-fn1;
            target--;
        }
        return fn2;
    }
}
```

对于N级台阶，可以从N-1级和N-2级上来，所以JumpFloor(N) = JumpFloor(N-1)+ JumpFloor(N-2) N=1时，只有一种 N=2时，有两种：一次2级；两次1级

```
public class Solution {
    public int JumpFloor(int target) {
        int result = 0;
        if(target > 0){
        if(target<=2)
            return target;
        else
            return result=JumpFloor(target-1)+JumpFloor(target-2);
    }
    return result;
    }
}
```

题目 变态跳台阶

考点 递归和循环 热点指数 89727 通过率 39.95%

具体题目

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。

```
public class Solution {
    public int JumpFloorII(int target) {
        // 假设：f(n)表示：n个台阶第一次1,2,...n阶的跳法数;
        // 若第一次跳了1阶，则还剩n-1阶，
        // 假设：f(n-1)表示：n-1个台阶第一次1,2,...n-1阶的跳法数;
        // 若第一次跳了2阶，则还剩n-2阶，
        // 假设：f(n-2)表示：n-1个台阶第一次1,2,...n-2阶的跳法数;
        // ...
        // 把所以可能的情况（第一次可能跳1,2,...,n阶）加起来：
        // 可以求出：f(n) = f(n-1) + f(n-2) + ... + f(1)
        // 递归：f(n-1) = f(n-2) + ... + f(1)
        // 可以求出：f(n) = 2*f(n-1)

        /*
        if (target <= 0) {
            return 0;
        } else if (target == 1) {
            return 1;
        } else {
            return 2 * JumpFloorII(target - 1);
        }
        */
        // 更实用的解法是：从下往上计算，避免了递归的多余计算量
        int a = 1, b = 0;
        if (target <= 0) {
            return 0;
        } else if (target == 1) {
            return 1;
        } else {
            for (int i = 2; i <= target; i++) {
                b = 2 * a;
                a = b;
            }
            return b;
        }
    }
}
```

题目 矩形覆盖

考点 递归和循环 热点指数 80347 通过率 34.42%

具体题目

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？

假设：n块矩形有f(n)种覆盖方法。进行逆向分析，要完成最后的搭建有两种可能。 第一种情况等价于情形1中阴影部分的n-1块矩形有多少种覆盖方法，为f(n-1); 第二种情况等价于情形2中阴影部分的n-2块矩形有多少种覆盖方法，为f(n-2); 故f(n) = f(n-1) + f(n-2)，还是一个斐波那契数列。。。。 且f(1) = 1, f(2) = 2，代码如下

```
  public class Solution {
    public int RectCover(int target) {
        if(target <= 0){
            return 0;
        }
        if(target == 1){
            return 1;
        }
        if(target == 2){
            return 2;
        }
        int first = 1;
        int second = 2;
        int result = 0;
        for(int i = 3; i <= target; i++){
            result = first + second;
            first = second;
            second = result;
        }
        return result;
    }
}
```

这里必须要吐槽一下，target为0的时候怎么就返回1了？？？出题者你出来解释一下，我保证不打残你。。。 /** * 其实就是一个斐波那契数列，满足公式：d(n) = d(n-1) + d(n-2) * @param target * @return */

```
        public int RectCover(int target) {
                int tempNum = 1;
                int result = 2;

                if (target == 0) {
                        return 1;
                }

                if (target == 1 || target == 2) {
                        return target;
                }

                int count = 2;
                while (count < target) {
                        result += tempNum;
                        tempNum = result - tempNum;
                        count ++;
                }
```

```
                return result;
        }
```

思路： f(1) = 1; f(2) = 2; 当n>2时，画图可知，第一块小矩形可横放和竖放。横放后剩余的长度为n-2，竖放后剩余的长度为n-1。 所以：f(n) = f(n-1) + f(n-2); (n > 2)

```
 public class Solution {
    public int RectCover(int target) {
        if (target <= 2) {
            return target;
        }
        int one = 1;
        int two = 2;
        int result = 0;
        for (int i = 3; i <= target; i++) {
            result = one + two;
            one = two;
            two = result;
        }
        return result;
    }
}
```

薛定谔的矩形：我们不用管矩形放在哪，只关注矩形本身。 很容易发现，当矩形横着放时，它下面必然还有一个横着放的矩形，那么就相当于一次放了两个矩形； 当矩形竖着放时，相当于一次放了一个矩形，那么结果就出来了，n个矩形可投放的方式只有一次放一块或者一次放两块， 所以得到f(n) = f(n-1) + f(n-2)，至于矩形放在哪，不放下去我们也不知道矩形在哪，但是那不重要。 我们只要知道当n小于等于2时的具体情况就可以。

```
 public class Solution {
    public int RectCover(int target) {
        if(target <= 2){
            return target;
        }else{
            return RectCover(target-1)+RectCover(target-2);
        }
    }
}
```

题目 二进制中1的个数

考点 位运算 热点指数 85618 通过率 34.00%

具体题目

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。

绝对最佳答案及分析：

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

答案正确:恭喜！您提交的程序通过了所有的测试用例 分析一下代码： 这段小小的代码，很是巧妙。 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。

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

/**

- 思路：将n与n-1想与会把n的最右边的1去掉，比如
- 1100&1011 = 1000
- 再让count++即可计算出有多少个1
- @author skyace * */

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

            return count;
        }
}
```