1、二维数组中的查找

题目描述

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。

题目分析

解法一：

利用该二维数组的性质：

- 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

改变个说法，即对于左下角的值 m，m 是该行最小的数，是该列最大的数

每次将 m 和目标值 target 比较：

1. 当 m < target，由于 m 已经是该行最大的元素，想要更大只有从列考虑，取值右移一位
2. 当 m > target，由于 m 已经是该列最小的元素，想要更小只有从行考虑，取值上移一位
3. 当 m = target，找到该值，返回 true

用某行最小或某列最大与 target 比较，每次可剔除一整行或一整列。

代码实现

解法一：

```
public boolean Find(int target, int [][] array) {
 if (array == null || array.length == 0) return false;
 if (array[0].length == 0) return false;
 int row = array.length - 1;
 int col = 0;
 do {
 if (array[row][col] < target) {
 col++;
 } else if (array[row][col] > target) {
  row--;
  } else {
  return true;
  }
```

```
14   } while (row >= 0 && col < array[0].length);
15   return false;
16  }
```

2、替换空格

题目描述

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。

题目分析

解法一：用Java自带的函数str.toString().replace(" ","%20")。

解法二：在当前字符串上进行替换。

1. 先计算替换后的字符串需要多大的空间，并对原字符串空间进行扩容；
2. 从后往前替换字符串的话，每个字符串只需要移动一次；
3. 如果从前往后，每个字符串需要多次移动，效率较低。

解法三：开辟一个新的字符串。

代码实现

解法一：

```
1  public static String replaceSpace(StringBuffer str) {
2   if (str == null || str.length() == 0) return "";
3   for (int i = 0; i < str.length(); i++) {
4   if (str.charAt(i) == ' ') {
5   str.deleteCharAt(i);
6   str.insert(i, "%20");
7   }
8   }
9   return str.toString();
10  }
```

3、从尾到头打印链表

题目描述

输入一个链表，按链表从尾到头的顺序返回一个ArrayList。

题目分析

解法一：ArrayList 中有个方法是 add(index,value)，可以指定 index 位置插入 value 值。所以我们在遍历 listNode 的同时将每个遇到的值插入到 list 的 0 位置，最后输出 listNode 即可得到逆序链表。

解法二：利用递归，借助系统的栈帮忙打印。

代码实现

解法一：

```
public class Solution {
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode)
{
        ArrayList<Integer> list = new ArrayList<>();
        ListNode tmp = listNode;
        while(tmp!=null){
            list.add(0,tmp.val);
            tmp = tmp.next;
        }
        return list;
     }
}
```

解法二：

```
public class Solution {
    ArrayList<Integer> list = new ArrayList();
    public ArrayList<Integer> printListFromTailToHead(ListNode listNode)
{
        if(listNode!=null){
            printListFromTailToHead(listNode.next);
            list.add(listNode.val);
        }
        return list;
    }
}
```

4、重建二叉树

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。

题目分析

解法一：根据中序遍历和前序遍历可以确定二叉树，具体过程为：

1. 根据前序序列第一个结点确定根结点
2. 遍历中序序列，查找前序序列第一个节点在中序序列所在位置
3. 根据根结点在中序序列中的位置分割出左右两个子序列
4. 对左子树和右子树分别递归使用同样的方法继续分解

代码实现

解法一：

```
public TreeNode reConstructBinaryTree(int [] pre, int [] in) {
 if (pre.length == 0 || in.length == 0) {
 return null;
 }
 TreeNode h = new TreeNode(pre[0]);
 int index = -1;
 for (int i = 0; i < in.length; i++) {
 if (in[i] == pre[0]) {
 index = i;
  }
  }
  int[] in1 = Arrays.copyOfRange(in, 0, index);
  int[] in2 = Arrays.copyOfRange(in, index + 1, in.length);
  int[] p1 = Arrays.copyOfRange(pre, 1, in1.length + 1);
  int[] p2 = Arrays.copyOfRange(pre, in1.length + 1, pre.length);
  h.left = reConstructBinaryTree(p1, in1);
  h.right = reConstructBinaryTree(p2, in2);
  return h;
}
```

5、用两个栈实现队列

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。

题目分析

解法一：栈A只负责push；当栈B没有元素时，将栈A元素全部pop到栈B，然后对栈B进行pop。

代码实现

解法一：

```
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() {
 if (stack2.empty()) {
 while (!stack1.empty()) {
  stack2.push(stack1.pop());
  }
  }
  if (!stack2.empty()) {
  return stack2.pop();
  } else {
  return -1;
  }
  }
}
```

6、旋转数组的最小数字

题目描述

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。

NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。

题目分析

解法一：暴力破解。在两段范围内都是非降序，当不符合这个规律时，就找到了最小数字。

解法二：二分查找。二分查找用于查找有序的数组中的值，题目所给数组在两段范围内有序，我们可以将给定数组分为两种情况：

1. 其实并没有旋转，例如 {1,2,3,4,5}，旋转后也是 {1,2,3,4,5}，这样可以直接使用二分查找；
2. 如题所示，旋转了一部分，例如 {1,2,3,4,5}，旋转后为 {3,4,5,1,2}，需要限定特殊条件后使用二分查找。

当数组如情况 1，有个鲜明的特征，即数组左边元素 < 数组右边元素，这时我们直接返回首元素即可；

当数组如情况 2，此时有三种可能找到最小值：

1. 下标为 n+1 的值小于下标为 n 的值，则下标为 n+1 的值肯定是最小元素；
2. 下标为 n 的值小于下标为 n-1 的值，则下标为 n 的值肯定是最小元素；
3. 由于不断查找，数组查找范围内的值已经全为非降序（退化为情况 1）；

再讨论每次二分查找时范围的变化，由于情况数组的情况 1 能直接找到最小值，需要变化范围的肯定是情况 2：

1. 当下标为 n 的值大于下标为 0 的值，从 0 到 n 这一段肯定是升序，由于是情况 2，最小值肯定在后半段；
2. 当下标为 n 的值小于下标为 0 的值，从 0 到 n 这一段不是升序，最小值肯定在这一段。

代码实现

解法一：*O(n)*

```
public int minNumberInRotateArray(int [] array) {
if (array.length == 0) return 0;
int index = 0;
for (int i = 1; i < array.length; i++) {
if (array[i] < array[i - 1]) {
index = i;
break;
}
}
```

```
 return array[index];
 }
```

解法二：*O(logn)*

```
public static int minNumberInRotateArray(int [] array) {
 if (array.length == 1) return array[0];
 int l = 0;
 int r = array.length - 1;
 while (l < r) {
 int m = l + ((r - l) >> 1);
 if (array[l] < array[r]) {
 return array[l];
 }
  if (array[m] > array[m + 1]) {
  return array[m + 1];
  }
  if (array[m] < array[m - 1]) {
  return array[m];
  }
  if (array[m] > array[0]) {
  l = m + 1;
  } else {
  r = m - 1;
  }
  }
  return 0;
}
```

7、斐波那契数列

题目描述

大家都知道斐波那契数列F(n)=F(n-1)+F(n-2)，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，第1项为1）。n<=39.

题目分析

解法一：递归。根据F(n)=F(n-1)+F(n-2)递归调用即可。

解法二：递归优化。类似于动态规划的优化，遍历数组即可。由于只使用到n-1和n-2两个值，所以可以只使用两个变量对它们进行存储，优化空间。

代码实现

解法一：*O(2^n)*

```
public int Fibonacci(int n) {
        if (n <= 1) return n;
        return Fibonacci(n-1) + Fibonacci(n-2);
}
```

解法一：*O(n)*

```
public int Fibonacci(int n) {
if (n == 0) return 0;
if (n <= 2) return 1;
int pre1 = 1;
int pre2 = 1;
int res = 0;
for (int i = 3; i <= n; i++) {
res = pre1 + pre2;
pre1 = pre2;
 pre2 = res;
 }
 return res;
 }
```

8、跳台阶

题目描述

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）。

题目分析

解法一：动态规划。斐波那契数列的变种。

代码实现

解法一：*O(n)*

```
public int JumpFloor(int target) {
if (target < 2) return 1;
int pre = 1;
int sum = 1;
for (int i = 2; i <= target; i++) {
sum = sum + pre;
pre = sum - pre;
}
```

```
9   return sum;
10   }
```

9、变态跳台阶

题目描述

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。

题目分析

解法一：动态规划。斐波那契数列的变种。由于青蛙每次可以跳n级台阶，因此青蛙跳上n级台阶的跳法为f(n)=f(n-1)+f(n-2)+……f(1)+f(0)。由于数字十分特殊，f(0)=1，f(1)=1，f(2)=2...通过观察可以得到，f(n)=2^(n-1)。

代码实现

解法一：*O(1)*

```
1 public static int JumpFloorII(int target) {
2  return target == 0 ? 1 : (int) Math.pow(2, target - 1);
3 }
```

10、矩形覆盖

题目描述

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？

题目分析

解法一：动态规划。斐波那契数列的变种。其实和第9题是一样的，f(n)=f(n-1)+f(n-2)。因为每次多加1*2的矩形时，无非就是f(n-1)种然后加上一条竖着的矩阵，或者f(n-2)种然后加上两条横着的矩阵。在该解法中使用的是递归方法求解，时间复杂度极高。在这仅仅是给出递归解的实现，建立使用时间复杂度为O(n)、空间复杂度为O(1)的最优解法(参考第7-9题)。

代码实现

解法一：*O(2^n)*

```
1 public static int RectCover(int target) {
2  if (target == 0) return 0;
3  if (target == 1) return 1;
```

```
4   if (target == 2) return 2;
5   return RectCover(target - 1) + RectCover(target - 2);
6 }
```