04 二维数组中的查找

```
@author: sdubrz
@date: 2020.05.02
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

在一个 n * m 的二维数组中，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。

示例:

现有矩阵 matrix 如下：

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

给定 target = 5，返回 true。

给定 target = 20，返回 false。

限制：

- 0 <= n <= 1000
- 0 <= m <= 1000

我的解法

对于给定的一个二维数组，根据题意，其最小值和最大值肯定分别在数组的开头和末尾，我们可以根据这个来初步判断目标数值是否有可能在这个数组中出现。如果有可能，则可以使用二分法进行查找。

比较数组中间位置元素与目标数值的大小，图1表示了比目标数值小的情况，图2表示了比目标数值大的情况。根据不同的情况，分别去继续探索红色、蓝色和绿色的区域。如此递归的实现。

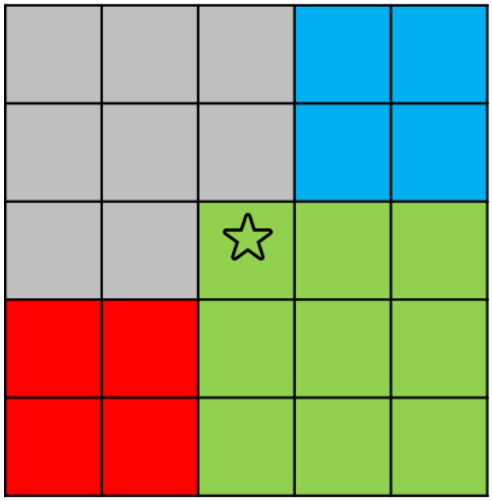


图1

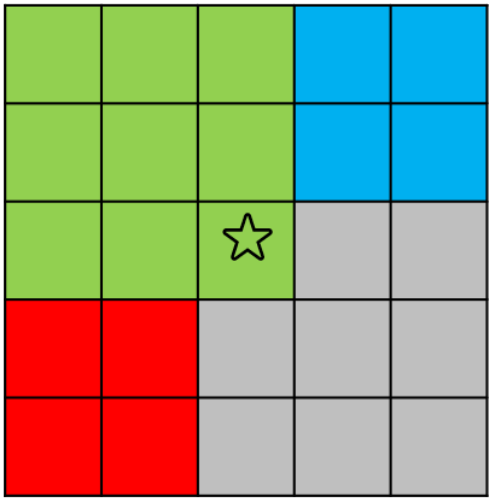


图2

下面是具体给出的Java程序实现。

```
class Solution {
        public boolean findNumberIn2DArray(int[][] matrix, int target) {
        if(matrix.length==0) {
            return false;
        }
        int m = matrix[0].length;
        if(m==0) {
            return false;
        }

        return this.find2d(matrix, target, 0, matrix.length-1, 0, m-1);
    }

    /**
     * 在矩阵中的一个矩形区域内查找是否含有目标数值
     * @param matrix 矩阵
     * @param target 要查找的数值
     * @param left 矩形区域的最左边
     * @param right 矩形区域的最右边
     * @param up 矩形区域的最上方
     * @param bottom 矩形区域的最下方
     * @return
     */
    private boolean find2d(int[][] matrix, int target, int up, int bottom, int
left, int right) {
        if(left==right && up==bottom) {
            return matrix[up][left]==target;
        }

        if(matrix[up][left]>target) {
            return false;
        }

        if(matrix[bottom][right]<target) {
            return false;
        }
```

```
        // 如果范围比较小，直接查找
        if(bottom-up<4 && right-left<4) {
            for(int i=up; i<=bottom; i++) {
                for(int j=left; j<=right; j++) {
                    if(matrix[i][j]==target) {
                        return true;
                    }
                }
            }
            return false;
        }

        // 用二分法
        int col_m = (left+right)/2;
        int row_m = (up+bottom)/2;

        if(matrix[row_m][col_m]==target) {
            return true;
        }

        if(matrix[row_m][col_m]<target) {
            boolean temp = this.find2d(matrix, target, row_m, bottom, col_m,
right);
            if(temp) {
                return true;
            }

//          if(row_m<bottom) {
//              temp = this.find2d(matrix, target, row_m+1, bottom, col_m,
col_m);
//              if(temp) {
//                  return true;
//              }
//          }

            if(row_m<bottom && col_m>left) {
                temp = this.find2d(matrix, target, row_m+1, bottom, left, col_m-
1);
                if(temp) {
                    return true;
                }
            }
            if(row_m>up && col_m<right) {
                temp = this.find2d(matrix, target, up, row_m-1, col_m+1, right);
                if(temp) {
                    return true;
                }
            }
        }

        if(matrix[row_m][col_m]>target) {
            boolean temp = this.find2d(matrix, target, up, row_m, left, col_m);
            if(temp) {
                return temp;
            }
            if(row_m<bottom && col_m>left) {
                temp = this.find2d(matrix, target, row_m+1, bottom, left, col_m-
1);
```

```
                if(temp) {
                    return true;
                }
            }
            if(row_m>up && col_m<right) {
                temp = this.find2d(matrix, target, up, row_m-1, col_m+1, right);
                if(temp) {
                    return temp;
                }
            }
        }


        return false;
    }
}
```

在 LeetCode 系统中提交的结果如下

```
执行结果： 通过 显示详情
执行用时 : 0 ms, 在所有 Java 提交中击败了 100.00% 的用户
内存消耗 : 45.7 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

官方给出的线性解法

下面是官方给出的一种线性时间复杂度的方法。

由于给定的二维数组具备每行从左到右递增以及每列从上到下递增的特点，当访问到一个元素时，可以排除数组中的部分元素。

从二维数组的右上角开始查找。如果当前元素等于目标值，则返回 `true`。如果当前元素大于目标值，则移到左边一列。如果当前元素小于目标值，则移到下边一行。

可以证明这种方法不会错过目标值。如果当前元素大于目标值，说明当前元素的下边的所有元素都一定大于目标值，因此往下查找不可能找到目标值，往左查找可能找到目标值。如果当前元素小于目标值，说明当前元素的左边的所有元素都一定小于目标值，因此往左查找不可能找到目标值，往下查找可能找到目标值。

- 若数组为空，返回 `false`
- 初始化行下标为 0，列下标为二维数组的列数减 1
- 重复下列步骤，直到行下标或列下标超出边界
  - 获得当前下标位置的元素 `num`
  - 如果 `num` 和 `target` 相等，返回 `true`
  - 如果 `num` 大于 `target`，列下标减 1
  - 如果 `num` 小于 `target`，行下标加 1
- 循环体执行完毕仍未找到元素等于 `target`，说明不存在这样的元素，返回 false`

```
class Solution {
    public boolean findNumberIn2DArray(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return false;
        }
        int rows = matrix.length, columns = matrix[0].length;
        int row = 0, column = columns - 1;
        while (row < rows && column >= 0) {
            int num = matrix[row][column];
            if (num == target) {
```

```
                return true;
            } else if (num > target) {
                column--;
            } else {
                row++;
            }
        }
        return false;
    }
}
```

相比较而言，官方的算法代码实现上要稍微更加优美一点。

05 替换空格

```
@author: sdubrz
@date: 2020.05.03
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

请实现一个函数，把字符串 s 中的每个空格替换成"%20"。

示例 1：

```
输入：s = "We are happy."
输出："We%20are%20happy."
```

限制：

```
0 <= s 的长度 <= 10000
```

我的解法

很明显的一种做法是遍历字符串的每个位置，判断是否是空格，然后进行替换。下面是Java程序实现。

```
class Solution {
    public String replaceSpace(String s) {
        if(s.length()==0){
            return s;
        }

        char[] array = s.toCharArray();
        String str = "";
        for(int i=0; i<s.length(); i++){
            if(array[i]==' '){
                str = str + "%20";
            }else{
                str = str + array[i];
            }
        }

        return str;
    }
```

```
}
```

在 LeetCode 系统中提交的结果为

```
执行结果： 通过 显示详情
执行用时 : 8 ms, 在所有 Java 提交中击败了 5.15% 的用户
内存消耗 : 40.1 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

官方解法

官方解法与我的解法思路是一致的，凡是具体的实现代码有所不同。结果是官方代码运行的速度要更快。下面是官方给出的Java程序实现

```java
class Solution {
    public String replaceSpace(String s) {
        int length = s.length();
        char[] array = new char[length * 3];
        int size = 0;
        for (int i = 0; i < length; i++) {
            char c = s.charAt(i);
            if (c == ' ') {
                array[size++] = '%';
                array[size++] = '2';
                array[size++] = '0';
            } else {
                array[size++] = c;
            }
        }
        String newStr = new String(array, 0, size);
        return newStr;
    }
}
```

下面是官方程序的提交结果，明显要比我的速度快了很多。应该是Java内部字符串相加的机制的原因吧。**因为我的代码中 `str=str+array[i]` 这种操作非常耗时，具体可参见《Java编程思想》中字符串的相关章节。**

```
执行结果： 通过 显示详情
执行用时 : 0 ms, 在所有 Java 提交中击败了 100.00% 的用户
内存消耗 : 37.8 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

06 从尾到头打印链表

```
@author: sdubrz
@date: 2020.05.03
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

示例 1：

```
输入：head = [1,3,2]
输出：[2,3,1]
```

限制：

0 <= 链表长度 <= 10000

我的解法

一种比较容易想到的方法就是用栈来把顺序颠倒：

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
import java.util.*;
class Solution {
    public int[] reversePrint(ListNode head) {
        Stack<Integer> stack = new Stack<>();
        while(head!=null){
            stack.push(head.val);
            head = head.next;
        }

        int n = stack.size();
        int[] nums = new int[n];
        for(int i=0; i<n; i++){
            nums[i] = stack.pop();
        }

        return nums;
    }
}
```

在 LeetCode 系统中提交的结果如下所示

```
执行结果： 通过 显示详情
执行用时 : 2 ms, 在所有 Java 提交中击败了 57.13% 的用户
内存消耗 : 40.9 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

官方解法

官方给出的解法也是使用的栈。不过他的栈中存储的是链表节点，而不是链表节点的数值。但是其实际运行速度要比我的稍快一些，这或许可以作为以后解题的技巧。下面是其具体的代码实现

```
class Solution {
    public int[] reversePrint(ListNode head) {
        Stack<ListNode> stack = new Stack<ListNode>();
        ListNode temp = head;
        while (temp != null) {
```

```
            stack.push(temp);
            temp = temp.next;
        }
        int size = stack.size();
        int[] print = new int[size];
        for (int i = 0; i < size; i++) {
            print[i] = stack.pop().val;
        }
        return print;
    }
}
```

在 LeetCode 系统中提交的结果如下，实际运行速度要比我的代码稍快。

```
执行结果： 通过 显示详情
执行用时 : 1 ms, 在所有 Java 提交中击败了 80.18% 的用户
内存消耗 : 40.1 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

07 重建二叉树

```
@author: sdubrz
@date: 2020.05.03
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。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，给出

```
前序遍历 preorder = [3,9,20,15,7]
中序遍历 inorder = [9,3,15,20,7]
```

返回如下的二叉树：

```
    3
   / \
  9  20
    /  \
   15   7
```

限制：

`0 <= 节点个数 <= 5000`

我的解法

对于一颗二叉树，其前序遍历中第一个元素必然是根节点，并且左子树的元素位于右子树元素的前面。而在中序遍历中，左子树的元素均位于根节点元素的前面，右子树的元素均位于根节点元素的后面。下图表示了这一位置关系，其中红色元素为根节点，绿色的为左子树中的元素，蓝色的为右子树中的元素。

前序遍历：

中序遍历：

根据这一位置关系，我们可以写出递归的解决方案，下面是具体的Java代码实现。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if(preorder.length==0){
            return null;
        }
        int n = preorder.length;
        TreeNode  root = this.buildTree(preorder, inorder, 0, n-1, 0, n-1);
        return root;
    }

    private TreeNode buildTree(int[] preorder, int[] inorder, int preHead, int
preTail, int inHead, int inTail){
        if(preHead==preTail){ // 只有一个节点
            TreeNode node = new TreeNode(preorder[preHead]);
            return node;
        }

        TreeNode root = new TreeNode(preorder[preHead]);
        int rootIndex = -1;
        for(int i=inHead; i<=inTail; i++){
            if(inorder[i]==preorder[preHead]){
                rootIndex = i;
                break;
            }
        }

        int leftSize = rootIndex - inHead;  // 左子树节点数
        int rightSize = inTail - rootIndex;  // 右子树节点数
        if(leftSize>0){
            root.left = this.buildTree(preorder, inorder, preHead+1,
preHead+leftSize, inHead, rootIndex-1);
        }
        if(rightSize>0){
            root.right = this.buildTree(preorder, inorder, preHead+leftSize+1,
preTail, rootIndex+1, inTail);
        }
        return root;
    }
```

```
}
```

在 LeetCode 系统中提交的结果如下

```
执行结果： 通过 显示详情
执行用时 : 4 ms, 在所有 Java 提交中击败了 60.63% 的用户
内存消耗 : 40.1 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

官方递归解法

LeetCode 的题解中官方给出了递归和迭代两种解法。其中，递归解法的思路与我的一致，不过代码的实现略有不同。下面是官方给出的递归解法的Java代码。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if (preorder == null || preorder.length == 0) {
            return null;
        }
        Map<Integer, Integer> indexMap = new HashMap<Integer, Integer>();
        int length = preorder.length;
        for (int i = 0; i < length; i++) {
            indexMap.put(inorder[i], i);
        }
        TreeNode root = buildTree(preorder, 0, length - 1, inorder, 0, length -
1, indexMap);
        return root;
    }

    public TreeNode buildTree(int[] preorder, int preorderStart, int
preorderEnd, int[] inorder, int inorderStart, int inorderEnd, Map<Integer,
Integer> indexMap) {
        if (preorderStart > preorderEnd) {
            return null;
        }
        int rootVal = preorder[preorderStart];
        TreeNode root = new TreeNode(rootVal);
        if (preorderStart == preorderEnd) {
            return root;
        } else {
            int rootIndex = indexMap.get(rootVal);
            int leftNodes = rootIndex - inorderStart, rightNodes = inorderEnd -
rootIndex;
            TreeNode leftSubtree = buildTree(preorder, preorderStart + 1,
preorderStart + leftNodes, inorder, inorderStart, rootIndex - 1, indexMap);
            TreeNode rightSubtree = buildTree(preorder, preorderEnd - rightNodes
+ 1, preorderEnd, inorder, rootIndex + 1, inorderEnd, indexMap);
```

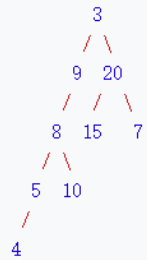
```
            root.left = leftSubtree;
            root.right = rightSubtree;
            return root;
        }
    }
}
```

由于使用了一个Map来存储中序遍历中每个元素与其索引的对应关系，因而在查找节点位置时，官方代码要比我的代码更快一些。

```
执行结果： 通过 显示详情
执行用时 : 3 ms, 在所有 Java 提交中击败了 81.16% 的用户
内存消耗 : 39.8 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

官方迭代解法

例如要重建的是如下二叉树。



其前序遍历和中序遍历如下。

```
preorder = [3, 9, 8, 5, 4, 10, 20, 15, 7]
inorder = [4, 5, 8, 10, 9, 3, 15, 20, 7]
```

前序遍历的第一个元素 `3` 是根节点，第二个元素 `9` 可能位于左子树或者右子树，需要通过中序遍历判断。

中序遍历的第一个元素是 `4` ，不是根节点 `3` ，说明 `9` 位于左子树，因为根节点不是中序遍历中的第一个节点。同理，前序遍历的后几个元素 `8` 、 `5` 、 `4` 也都位于左子树，且每个节点都是其上一个节点的左子节点。

前序遍历到元素 `4` ，和中序遍历的第一个元素相等，说明前序遍历的下一个元素 `10` 位于右子树。那么 `10` 位于哪个元素的右子树？从前序遍历看， `10` 可能位于 `4` 、 `5` 、 `8` 、 `9` 、 `3` 这些元素中任何一个元素的右子树。从中序遍历看， `10` 在 `8` 的后面，因此 `10` 位于 `8` 的右子树。把前序遍历的顺序反转，则在 `10` 之前的元素是 `4` 、 `5` 、 `8` 、 `9` 、 `3` ，其中 `8` 是最后一次相等的节点，因此前序遍历的下一个元素位于中序遍历中最后一次相等的节点的右子树。

同理可知， `20` 位于 `3` 的右子树， `15` 和 `7` 分别是 `20` 的左右子节点。

根据上述例子和分析，可以使用栈保存遍历过的节点。初始时令中序遍历的指针指向第一个元素，遍历前序遍历的数组，如果前序遍历的元素不等于中序遍历的指针指向的元素，则前序遍历的元素为上一个节点的左子节点。如果前序遍历的元素等于中序遍历的指针指向的元素，则正向遍历中序遍历的元素同时反向遍历前序遍历的元素，找到最后一次相等的元素，将前序遍历的下一个节点作为最后一次相等的元素的右子节点。其中，反向遍历前序遍历的元素可通过栈的弹出元素实现。

- 使用前序遍历的第一个元素创建根节点。
- 创建一个栈，将根节点压入栈内。
- 初始化中序遍历下标为 `0` 。
- 遍历前序遍历的每个元素，判断其上一个元素（即栈顶元素）是否等于中序遍历下标指向的元素。
    - 若上一个元素不等于中序遍历下标指向的元素，则将当前元素作为其上一个元素的左子节点，并将当前元素压入栈内。
    - 若上一个元素等于中序遍历下标指向的元素，则从栈内弹出一个元素，同时令中序遍历下标指向下一个元素，之后继续判断栈顶元素是否等于中序遍历下标指向的元素，若相等则重复该操作，直至栈为空或者元素不相等。然后令当前元素为最后一个想等元素的右节点。
- 遍历结束，返回根节点。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if (preorder == null || preorder.length == 0) {
            return null;
        }
        TreeNode root = new TreeNode(preorder[0]);
        int length = preorder.length;
        Stack<TreeNode> stack = new Stack<TreeNode>();
        stack.push(root);
        int inorderIndex = 0;
        for (int i = 1; i < length; i++) {
            int preorderVal = preorder[i];
            TreeNode node = stack.peek();
            if (node.val != inorder[inorderIndex]) {
                node.left = new TreeNode(preorderVal);
                stack.push(node.left);
            } else {
                while (!stack.isEmpty() && stack.peek().val ==
inorder[inorderIndex]) {
                    node = stack.pop();
                    inorderIndex++;
                }
                node.right = new TreeNode(preorderVal);
                stack.push(node.right);
```

```
            }
        }
        return root;
    }
}
```

下面是在 LeetCode 系统中提交的结果

```
执行结果： 通过 显示详情
执行用时 : 3 ms, 在所有 Java 提交中击败了 81.16% 的用户
内存消耗 : 39.6 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

由于每个元素都需要一次新建节点的过程，所以这三种方法的时间复杂度均为 $O(n)$。

09 用两个栈实现队列

```
@author: sdubrz
@date: 2020.05.04
难度： 简单
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead ，分别完成在队列尾部插入整数和在队列头部删除整数的功能。(若队列中没有元素，deleteHead 操作返回 -1)

示例 1：

```
输入：
["CQueue","appendTail","deleteHead","deleteHead"]
[[],[3],[],[]]
输出：[null,null,3,-1]
```

示例 2：

```
输入：
["CQueue","deleteHead","appendTail","appendTail","deleteHead","deleteHead"]
[[],[],[5],[2],[],[]]
输出：[null,-1,null,null,5,2]
```

提示：

- 1 <= values <= 10000
- 最多会对 appendTail、deleteHead 进行 10000 次调用

我的解法

队列是先进先出的，栈是先进后出的，所以用一个栈将另一个栈的顺序颠倒一下，就可以改成先进先出的队列了。这个比较简单，下面是Java程序实现：

```java
import java.util.*;
class CQueue {
    private Stack<Integer> stack1;
```

```
    private Stack<Integer> stack2;
    public CQueue() {
        this.stack1 = new Stack<>();
        this.stack2 = new Stack<>();
    }

    public void appendTail(int value) {
        while(!stack2.isEmpty()){
            stack1.push(stack2.pop());
        }
        stack1.push(value);
    }

    public int deleteHead() {
        if(stack1.isEmpty() && stack2.isEmpty()){
            return -1;
        }

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

        return stack2.pop();
    }
}

/**
 * Your CQueue object will be instantiated and called as such:
 * CQueue obj = new CQueue();
 * obj.appendTail(value);
 * int param_2 = obj.deleteHead();
 */
```

在 LeetCode 系统中提交的结果如下

```
执行结果： 通过 显示详情
执行用时 : 198 ms, 在所有 Java 提交中击败了 39.76% 的用户
内存消耗 : 47.9 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

网友的解法

在我的方法中每次插入和删除都要先把所有元素放入同一个栈中。有网友指出，不需要在插入和删除时都进行顺序颠倒，例如当用 stack1 管理添加，stack2 管理删除时，在删除的时候只需要在 stack2 为空时，才将 stack1 中的元素全部转移到 stack2 即可，而在添加元素时，只需要简单地往 stack1 中添加元素即可，不用管 stack2 。这样可以减少两个栈之间不必要的元素交换。下面是根据这一思路，用Java的具体实现。

```
class CQueue {

    public int size;
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public CQueue() {
        this.size = 0;
```

```
        this.stack1 = new Stack<>();
        this.stack2 = new Stack<>();
    }

    public void appendTail(int value) {
        stack1.push(value);
        size++;
    }

    public int deleteHead() {
        if(size==0){
            return -1;
        }

        if(stack2.isEmpty()){
            while(!stack1.isEmpty()){
                stack2.push(stack1.pop());
            }
        }
        size--;
        return stack2.pop();
    }
}

/**
 * Your CQueue object will be instantiated and called as such:
 * CQueue obj = new CQueue();
 * obj.appendTail(value);
 * int param_2 = obj.deleteHead();
 */
```

在 LeetCode 系统中提交结果显示，这一方法要比我前面的方法快一些。提交结果显示，这一方法也不是所有 Java 提交中最快的方法，是因为部分网友没有真正地使用两个栈来实现，而是使用了数组或直接用的库中的队列实现，拉高了排名标准。

```
执行结果： 通过 显示详情
执行用时 : 66 ms, 在所有 Java 提交中击败了 66.77% 的用户
内存消耗 : 47.9 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

10 青蛙跳台阶问题

```
@author: sdubrz
@date: 2020.05.05
难度： 简单
考察内容： 动态规划
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

示例 1：

```
输入：n = 2
输出：2
```

示例 2：

```
输入：n = 7
输出：21
```

提示：

- 0 <= n <= 100

解法

这道题可以用动态规划来解。对于规模为 n 的问题，最后一阶台阶有两种走法：单独走（这时候只需要知道前n-1阶有多少种走法即可），与上一阶同一步走（这时候只需要知道前n-2阶有多少种走法即可）。所以有如下递推式

$$f(n) = f(n-1) + f(n-2)$$

很明显，这是一个斐波那契数列数列的应用问题。下面是具体的Java程序实现

```java
class Solution {
    public int numWays(int n) {
        if(n==0||n==1){
            return 1;
        }

        int[] count = new int[n+1];
        count[0] = 1;
        count[1] = 1;
        for(int i=2; i<=n; i++){
            count[i] = (count[i-1] + count[i-2])%1000000007;
        }

        return count[n];
    }
}
```

在 LeetCode 系统中提交的结果如下

```
执行结果： 通过 显示详情
执行用时 : 0 ms, 在所有 Java 提交中击败了 100.00% 的用户
内存消耗 : 35.9 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

11 旋转数组中的最小数字

```
@author: sdubrz
@date: 2020.05.06
难度： 简单
考察内容： 数组
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

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。输入一个递增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。

示例 1：

```
输入：[3,4,5,1,2]
输出：1
```

示例 2：

```
输入：[2,2,2,0,1]
输出：0
```

通过次数22,758 提交次数48,595

我的解法

由于输入是一个递增排序数组的旋转，因而，一种比较容易想到的方法就是从数组的末尾开始向前遍历，如果某个位置前面的元素比它大，那么它就是整个数组中的最小元素。这样时间复杂度为 $ O(n) $。下面是Java程序的实现

```java
class Solution {
    public int minArray(int[] numbers) {
        if(numbers.length==1){
            return numbers[0];
        }

        int minIndex = numbers.length-1;
        while(minIndex-1>0){
            if(numbers[minIndex-1]>numbers[minIndex]){
                break;
            }
            minIndex--;
        }

        if(minIndex-1==0){
            return Math.min(numbers[0], numbers[1]);
        }else{
            return numbers[minIndex];
        }
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果： 通过 显示详情
执行用时 : 1 ms, 在所有 Java 提交中击败了 50.32% 的用户
内存消耗 : 39.7 MB, 在所有 Java 提交中击败了 100.00% 的用户
```

12 矩阵中的路径

```
@author: sdubrz
@date: 6/14/2020 8:51:16 PM
难度： 中等
考察内容： 数组 深度优先搜索
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

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格开始，每一步可以在矩阵中向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。例如，在下面的3×4的矩阵中包含一条字符串“bfce”的路径（路径中的字母用加粗标出）。

[["a","b","c","e"],
["s","f","c","s"],
["a","d","e","e"]]

但矩阵中不包含字符串“abfb”的路径，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，路径不能再次进入这个格子。

示例 1：

```
输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word =
"ABCCED"
输出：true
```

示例 2：

```
输入：board = [["a","b"],["c","d"]], word = "abcd"
输出：false
```

提示：

- 1 <= board.length <= 200
- 1 <= board[i].length <= 200

注意：本题与LeetCode 79 题相同：https://leetcode-cn.com/problems/word-search/

通过次数24,199 提交次数55,137

我的解法 深度优先搜索

拿到这道题没有想到更加有效的方法，只想起可以用深搜来暴力地寻找，应该不是比较快的方法，因为有很多重复比较的地方。很多的搜索匹配问题，在想不起更加有效的方法时，可以用深搜和广搜来当应急的方法，下面是Java程序的实现：

```java
class Solution {
    public boolean exist(char[][] board, String word) {
        int n = board.length;
        if(n<1) {
            return false;
        }
        int m = board[0].length;
        if(n*m<word.length()) {
            return false;
```

```
        }

        char[] words = word.toCharArray();
        for(int x=0; x<n; x++) {
            for(int y=0; y<m; y++) {
                if(board[x][y]==words[0]) {
                    int[][] visited = new int[n][m];
                    visited[x][y] = 1;
                    boolean current = this.exist(board, words, x, y, visited,
0);
                    if(current) {
                        return true;
                    }
                }
            }
        }


        return false;
    }

    public boolean exist(char[][] board, char[] words, int x, int y, int[][]
visited, int offset) {
        if(offset==words.length-1) {
            return true;
        }
        if(x<board.length-1 && visited[x+1][y]==0) {  // 探索下边的单元
            if(board[x+1][y]==words[offset+1]) {
                int[][] visited2 = new int[board.length][];
                for(int i=0; i<board.length; i++) {
                    visited2[i] = visited[i].clone();
                }
                visited2[x+1][y] = 1;
                boolean left = this.exist(board, words, x+1, y, visited2,
offset+1);
                if(left) {
                    return true;
                }
            }
        }

        if(x>0 && visited[x-1][y]==0) {  // 探索上边的单元
            if(board[x-1][y]==words[offset+1]) {
                int[][] visited2 = new int[board.length][];
                for(int i=0; i<board.length; i++) {
                    visited2[i] = visited[i].clone();
                }
                visited2[x-1][y] = 1;
                boolean left = this.exist(board, words, x-1, y, visited2,
offset+1);
                if(left) {
                    return true;
                }
            }
        }

        if(y<board[0].length-1 && visited[x][y+1]==0) {  // 探索右边的单元
            if(board[x][y+1]==words[offset+1]) {
```

```
                int[][] visited2 = new int[board.length][];
                for(int i=0; i<board.length; i++) {
                    visited2[i] = visited[i].clone();
                }
                visited2[x][y+1] = 1;
                boolean left = this.exist(board, words, x, y+1, visited2,
offset+1);
                if(left) {
                    return true;
                }
            }
        }

        if(y>0 && visited[x][y-1]==0) {  // 探索左边的单元
            if(board[x][y-1]==words[offset+1]) {
                int[][] visited2 = new int[board.length][];
                for(int i=0; i<board.length; i++) {
                    visited2[i] = visited[i].clone();
                }
                visited2[x][y-1] = 1;
                boolean left = this.exist(board, words, x, y-1, visited2,
offset+1);
                if(left) {
                    return true;
                }
            }
        }

        return false;
    }

}
```

在 LeetCode 系统中提交的结果如下：

```
执行结果：通过 显示详情
执行用时 :85 ms, 在所有 Java 提交中击败了5.03%的用户
内存消耗 :44 MB, 在所有 Java 提交中击败了100.00%的用户
```

网友的版本

下面是一个网友实现的版本，基本思想与我是一样的，不过代码实现要比我好，在系统中提交的结果显示，运行速度要快于我的代码。我的代码之所以慢，主要原因应该是在保存数组的副本的时候浪费了大量的时间。做这种编程题，除了要有正确的思路之外，编码的质量也是十分重要的。

```
class Solution {
    public boolean exist(char[][] board, String word) {
        char[] words = word.toCharArray();
        for(int i = 0; i < board.length; i++) {
            for(int j = 0; j < board[0].length; j++) {
                if(dfs(board, words, i, j, 0)) return true;
            }
        }
        return false;
```

```
    }
    boolean dfs(char[][] board, char[] word, int i, int j, int k) {
        if(i >= board.length || i < 0 || j >= board[0].length || j < 0 ||
board[i][j] != word[k]) return false;
        if(k == word.length - 1) return true;
        char tmp = board[i][j];
        board[i][j] = '/';
        boolean res = dfs(board, word, i + 1, j, k + 1) || dfs(board, word, i -
1, j, k + 1) ||
                      dfs(board, word, i, j + 1, k + 1) || dfs(board, word, i ,
j - 1, k + 1);
        board[i][j] = tmp;
        return res;
    }
}
```

作者：jyd
链接：https://leetcode-cn.com/problems/ju-zhen-zhong-de-lu-jing-lcof/solution/mian-shi-ti-12-ju-zhen-zhong-de-lu-jing-shen-du-yo/
来源：力扣（LeetCode）

13 机器人的运动范围

```
@author: sdubrz
@date: 6/14/2020 10:38:29 PM
难度： 中等
考察内容： 数组 深度优先搜索
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

地上有一个m行n列的方格，从坐标 [0,0] 到坐标 [m-1,n-1] 。一个机器人从坐标 [0, 0] 的格子开始移动，它每次可以向左、右、上、下移动一格（不能移动到方格外），也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。但它不能进入方格 [35, 38]，因为3+5+3+8=19。请问该机器人能够到达多少个格子？

示例 1：

```
输入：m = 2, n = 3, k = 1
输出：3
```

示例 2：

```
输入：m = 3, n = 1, k = 0
输出：1
```

提示：

- 1 <= n,m <= 100
- 0 <= k <= 20

通过次数39,759 提交次数81,812

我的解法 深度优先搜索

这种搜索的题可以用深度优先搜索来暴力搜索，不过在该题中需要判断某个点是否可达。Java程序实现如下：

```
import java.util.*;
class Solution {
    public int movingCount(int m, int n, int k) {
        int[][] visited = new int[m][n];
        int count = 0;
        Stack<Integer> stack1 = new Stack<>();
        Stack<Integer> stack2 = new Stack<>();

        stack1.push(0);
        stack2.push(0);
        visited[0][0] = 1;
        count++;

        while(!stack1.isEmpty()) {
            int x = stack1.pop();
            int y = stack2.pop();

            if(x>0 && visited[x-1][y]==0) {  // 尝试左边的
                if(this.available(x-1, y, k)) {
                    stack1.push(x-1);
                    stack2.push(y);
                    visited[x-1][y] = 1;
                    count++;
                }
            }

            if(x<m-1 && visited[x+1][y]==0) {  // 尝试右边的
                if(this.available(x+1, y, k)) {
                    stack1.push(x+1);
                    stack2.push(y);
                    visited[x+1][y] = 1;
                    count++;
                }
            }

            if(y>0 && visited[x][y-1]==0) {
                if(this.available(x, y-1, k)) {
                    stack1.push(x);
                    stack2.push(y-1);
                    visited[x][y-1] = 1;
                    count++;
                }
            }

            if(y<n-1 && visited[x][y+1]==0) {
                if(this.available(x, y+1, k)) {
                    stack1.push(x);
                    stack2.push(y+1);
                    visited[x][y+1] = 1;
                    count++;
                }
```

```
            }
        }

        return count;
    }

    private boolean available(int x, int y, int k) {
        int a = x;
        int b = y;
        int sum = 0;
        while(a>0) {
            sum += a % 10;
            a = a / 10;
        }
        while(b>0) {
            sum += b % 10;
            b = b / 10;
        }
        if(sum>k)
            return false;
        else
            return true;
    }

}
```

在 LeetCode 系统中提交的结果如下所示

```
执行结果：通过 显示详情
执行用时 :6 ms, 在所有 Java 提交中击败了24.09%的用户
内存消耗 :36.6 MB, 在所有 Java 提交中击败了100.00%的用户
```

14 剪绳子

```
@author: sdubrz
@date: 7/2/2020 3:46:36 PM
难度： 中等
考察内容： 动态规划
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

给你一根长度为 n 的绳子，请把绳子剪成整数长度的 m 段（m、n都是整数，n>1并且m>1），每段绳子的长度记为 k[0],k[1]...k[m-1] 。请问 k[0]*k[1]*...*k[m-1] 可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。

示例 1：

```
输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1
```

示例 2:

```
输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36
```

提示:

- 2 <= n <= 58

通过次数29,381 提交次数53,858

我的解法

这道题可以用动态规划来解：Java程序实现如下

```java
class Solution {
    public int cuttingRope(int n) {
        if(n==2) {
            return 1;
        }
        int[] array = new int[n+1];
        array[1] = 1;
        array[2] = 2;

        for(int i=3; i<n; i++) {
            int temp = i;
            for(int j=1; j<=i/2; j++) {
                int t = array[j] * array[i-j];
                if(t>temp) {
                    temp = t;
                }
            }
            array[i] = temp;
        }

        int temp0 = array[1] * array[n-1];
        for(int i=2; i<=n/2; i++) {
            int t = array[i] * array[n-i];
            if(temp0<t) {
                temp0 = t;
            }
        }

        return temp0;
    }
}
```

在 LeetCode 系统中提交的结果如下所示

```
执行结果：通过 显示详情
执行用时：0 ms, 在所有 Java 提交中击败了100.00%的用户
内存消耗：36.2 MB, 在所有 Java 提交中击败了100.00%的用户
```

剪绳子II

```
@date: 7/16/2020 3:20:14 PM
```

现在对题目进行修改，使 n 的取值范围扩大至 `2<=n<=1000`。这时候就不能再使用动态规划的方法了。这道题可以用贪心的算法来解，规律是尽量分配较多的3。

下面是网友pipi给出的Java实现：

```
class Solution {
    public int cuttingRope(int n) {
         if(n == 2)
            return 1;
        if(n == 3)
            return 2;
        long res = 1;
        while(n > 4){
            res *= 3;
            res = res % 1000000007;
            n -= 3;
        }
        return (int)(res * n % 1000000007);
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：0 ms，在所有 Java 提交中击败了100.00%的用户
内存消耗：36.4 MB，在所有 Java 提交中击败了100.00%的用户
```

这道题，真正在应试过程中从正面推导出做法可能比较困难，或许网友KAI给出的这种暴力找规律的思路更实用一点，或许可解燃眉之急：

```
class Solution {
public:
    void dfs(int n, long sum, long multi, long& ans) {
        if (sum == n) {
            ans = max(ans, multi);
            return;
        } else if (sum > n) {
            return;
        }
        for (long i = 1; i < n; i++) {
            dfs(n, i+sum, i*multi, ans);
        }
        return;
    }

    int cuttingRope(int n) {
        long ans = 0;
        dfs(n, 0, 1, ans);
        return ans;
    }
};

//--->
```

```
n      乘积      子数字
2        1        1 1
3        2        1 2
4        4        2 2
5        6        2 3
6        9        3 3
7        12       2 2 3
8        18       2 3 3
9        27       3 3 3
10       36       2 2 3 3
11       54       2 3 3 3
12       81       3 3 3 3
13       108      2 2 3 3 3
14       162      2 3 3 3 3
15       243      3 3 3 3 3
16       324      2 2 3 3 3 3
17       486      2 3 3 3 3 3
18       729      3 3 3 3 3 3
19       972      2 2 3 3 3 3 3
20       1458     2 3 3 3 3 3 3
21       2187     3 3 3 3 3 3 3
22       2916     2 2 3 3 3 3 3 3
23       4374     2 3 3 3 3 3 3 3
24       6561     3 3 3 3 3 3 3 3
25       8748     2 2 3 3 3 3 3 3 3
26       13122    2 3 3 3 3 3 3 3 3
27       19683    3 3 3 3 3 3 3 3 3
28       26244    2 2 3 3 3 3 3 3 3 3
29       39366    2 3 3 3 3 3 3 3 3 3
```

15 二进制中1的个数

```
@author: sdubrz
@date: 7/16/2020 3:48:10 PM
难度： 简单
考察内容： 二进制
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

请实现一个函数，输入一个整数，输出该数二进制表示中 1 的个数。例如，把 9 表示成二进制是 1001，有 2 位是 1。因此，如果输入 9，则该函数输出 2。

示例 1：

```
输入：00000000000000000000000000001011
输出：3
解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。
```

示例 2：

```
输入：00000000000000000000000010000000
输出：1
解释：输入的二进制串 00000000000000000000000010000000 中，共有一位为 '1'。
```

示例 3:

```
输入：11111111111111111111111111111101
输出：31
解释：输入的二进制串 11111111111111111111111111111101 中，共有 31 位为 '1'。
```

解法

下面是网友Satan给出的解法：

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代码实现如下

```
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0;
        while(n!=0){
            count++;
            n = n & (n-1);
        }
        return count;
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：1 ms, 在所有 Java 提交中击败了99.27%的用户
内存消耗：36.6 MB, 在所有 Java 提交中击败了100.00%的用户
```

另：Java中的无符号右移运算符为 `>>>`

16 数值的整数次方（未完成）

```
@author: sdubrz
@date: 8/13/2020 16:48:23 PM
难度： 中等
考察内容：
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

实现函数double Power(double base, int exponent)，求base的exponent次方。不得使用库函数，同时不需要考虑大数问题。

示例 1:

```
输入: 2.00000, 10
输出: 1024.00000
```

示例 2:

```
输入: 2.10000, 3
输出: 9.26100
```

示例 3:

```
输入: 2.00000, -2
输出: 0.25000
解释: 2^-2 = 1/2^2 = 1/4 = 0.25
```

说明:

- -100.0 < x < 100.0
- n 是 32 位有符号整数，其数值范围是 [−231, 231 − 1] 。

我的第一种解法

我设计的解法是通过对指数进行分析，以达到减少乘法次数的目的的。但是提交之后显示超出内存限制。

```
class Solution {
    public double myPow(double x, int n) {
        if(n==0) {
            return 1;
        }
        boolean inv = false;
        if(n<0) {
            n = -1*n;
            inv = true;
        }

        ArrayList<Double> list = new ArrayList<>();
        //ArrayList<Integer> list2 = new ArrayList<>();
        //list.add(1.0);
        list.add(x);
        //list2.add(1);
        int index = 0;  // 2的指数
        int temp = 1;  // pow(2, index)
        while(temp*2<=n) {
            double current = list.get(index) * list.get(index);
            list.add(current);
            temp *= 2;
            //list2.add(temp);
            index++;
        }

        double result = 1.0;

        while(n>0) {
            result *= list.get(index);
            n = n - temp;
            while(temp>n) {
                temp /= 2;
                index--;
            }
        }
```

```
        // System.out.println(list);

        if(inv) {
            return 1.0/result;
        }else {
            return result;
        }
    }
}
```

提交结果显示超出内存限制。

18 删除链表的节点

```
@author: sdubrz
@date: 7/25/2020 9:41:33 AM
难度： 简单
考察内容： 二进制
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

给定单向链表的头指针和一个要删除的节点的值，定义一个函数删除该节点。

返回删除后的链表的头节点。

注意：此题对比原题有改动

示例 1:

```
输入: head = [4,5,1,9], val = 5
输出: [4,1,9]
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.
```

示例 2:

```
输入: head = [4,5,1,9], val = 1
输出: [4,5,9]
解释: 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.
```

说明:

- 题目保证链表中节点的值互不相同
- 若使用 C 或 C++ 语言，你不需要 free 或 delete 被删除的节点

通过次数40,406提交次数68,572

解法

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
```

```
 * }
 */
class Solution {
    public ListNode deleteNode(ListNode head, int val) {
        if(head==null) {
            return null;
        }
        if(head.val==val) {
            head = head.next;
            return head;
        }

        ListNode pre = head;
        ListNode current = head.next;
        while(current!=null) {
            if(current.val==val) {
                pre.next = current.next;
                break;
            }else {
                pre = pre.next;
                current = current.next;
            }
        }

        return head;

    }
}
```

在LeetCode系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：0 ms, 在所有 Java 提交中击败了100.00%的用户
内存消耗：39.4 MB, 在所有 Java 提交中击败了100.00%的用户
```

21 调整数组顺序使奇数位于偶数前面

```
@author: sdubrz
@date: 7/25/2020 9:59:53 AM
难度： 简单
考察内容： 二进制
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

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，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

示例：

```
输入：nums = [1,2,3,4]
输出：[1,3,2,4]
注：[3,1,2,4] 也是正确的答案之一。
```

提示：

- 1 <= nums.length <= 50000
- 1 <= nums[i] <= 10000

通过次数41,445提交次数64,453

解法

```
class Solution {
    public int[] exchange(int[] nums) {
        if(nums.length<2) {
            return nums;
        }
        int pre = 0;
        int back = nums.length-1;

        while(pre<back) {
            while(pre<nums.length &&nums[pre]%2==1) {  // 从前往后找偶数
                pre++;
            }
            while(back>=0 &&nums[back]%2==0) {  // 从后往前找奇数
                back--;
            }

            if(pre<back) {
                int a = nums[pre];
                nums[pre] = nums[back];
                nums[back] = a;
            }
        }

        return nums;
    }
}
```

在LeetCode系统中提交的结果为

```
执行结果：通过  显示详情
执行用时：2 ms，在所有 Java 提交中击败了99.82%的用户
内存消耗：47.6 MB，在所有 Java 提交中击败了100.00%的用户
```

22 链表中倒数第k个节点

```
@author: sdubrz
@date: 7/25/2020 10:11:41 AM
难度： 简单
考察内容： 二进制
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

输入一个链表，输出该链表中倒数第k个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。例如，一个链表有6个节点，从头节点开始，它们的值依次是1、2、3、4、5、6。这个链表的倒数第3个节点是值为4的节点。

示例：

```
给定一个链表: 1->2->3->4->5, 和 k = 2.

返回链表 4->5.
```

通过次数49,771提交次数63,088

解法 遍历两次链表

- 第一次遍历获得链表的长度，计算出要返回的节点的正序位置。
- 第二次遍历得到结果。

下面是Java程序的实现

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode getKthFromEnd(ListNode head, int k) {
        int n = 0;
        ListNode current = head;
        // 第一次遍历，获知链表长度
        while(current!=null) {
            n++;
            current = current.next;
        }

        if(k>n) {  // 链表中元素个数不够
            return null;
        }

        int aim = n-k+1;  // 目标节点的正序次序
        int ptr = 1;
        current = head;
        // 第二次遍历到达目标节点
        while(ptr<aim) {
            current = current.next;
            ptr++;
        }

        return current;
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过显示详情
执行用时：0 ms，在所有 Java 提交中击败了100.00%的用户
内存消耗：37.6 MB，在所有 Java 提交中击败了100.00%的用户
```

24 反转链表

```
@author: sdubrz
@date: 7/25/2020 10:46:14 AM
难度： 简单
考察内容： 链表
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

示例:

```
输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL
```

限制:

0 <= 节点个数 <= 5000

解法一 栈

对于颠倒顺序的问题，最容易想到的解法就是用栈来实现：

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head==null) {
            return null;
        }
        if(head.next==null) {
            return head;
        }

        // 剩下的情况就是链表中至少有两个节点的情况
        Stack<ListNode> stack = new Stack<>();
        ListNode current = head;
        while(current!=null) {
            stack.push(current);
            current = current.next;
        }

        ListNode list2 = stack.pop();
        ListNode current2 = list2;
        while(stack.size()>1) {
            current2.next = stack.pop();
            current2 = current2.next;
        }
```

```
        ListNode lastNode = stack.pop();
        lastNode.next = null;
        current2.next = lastNode;

        return list2;
    }
}
```

但是对于链表的反转问题，栈的解法效率并不高。

```
执行结果：通过 显示详情
执行用时：1 ms，在所有 Java 提交中击败了6.67%的用户
内存消耗：39.5 MB，在所有 Java 提交中击败了100.00%的用户
```

解法二 一次遍历链表

遍历一次链表，并在遍历的过程中改变next的指向关系，具体实现如下：

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head==null) {
            return null;
        }
        if(head.next==null) {
            return head;
        }

        // 剩下的情况就是链表中至少有两个节点的情况
        ListNode pre = head;  // 前一个节点
        ListNode current = pre.next;  // 当前节点
        pre.next = null;
        while(current.next!=null) {
            ListNode back = current.next;
            current.next = pre;
            pre = current;
            current = back;
        }
        current.next = pre;

        return current;
    }
}
```

这种方法的效率要明显高于栈实现的版本：

```
执行结果：通过 显示详情
执行用时：0 ms，在所有 Java 提交中击败了100.00%的用户
内存消耗：40 MB，在所有 Java 提交中击败了100.00%的用户
```

25 合并两个排序的链表

```
@author: sdubrz
@date: 7/25/2020 3:23:12 PM
难度： 简单
考察内容： 链表
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

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节点仍然是递增排序的。

示例1：

```
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

限制：

0 <= 链表长度 <= 1000

解法

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1==null){
            return l2;
        }
        if(l2==null){
            return l1;
        }

        ListNode head;
        ListNode ptr1;
        ListNode ptr2;
        if(l1.val>l2.val){
            head = l2;
            ptr1 = l1;
            ptr2 = l2.next;
        }else{
            head = l1;
            ptr1 = l1.next;
            ptr2 = l2;
```

```
        }

        ListNode current = head;
        while(ptr1!=null && ptr2!=null){
            if(ptr1.val>ptr2.val){
                current.next = ptr2;
                ptr2 = ptr2.next;
            }else{
                current.next = ptr1;
                ptr1 = ptr1.next;
            }
            current = current.next;
        }

        if(ptr1!=null){
            current.next = ptr1;
        }else{
            current.next = ptr2;
        }

        return head;
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：1 ms, 在所有 Java 提交中击败了99.44%的用户
内存消耗：39.9 MB, 在所有 Java 提交中击败了100.00%的用户
```

26 树的子结构

```
@author: sdubrz
@date: 7/25/2020 4:03:31 PM
难度： 中等
考察内容： 树
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

输入两棵二叉树A和B，判断B是不是A的子结构。(约定空树不是任意一个树的子结构)

B是A的子结构， 即 A中有出现和B相同的结构和节点值。

例如:

```
给定的树 A:

     3
    / \
   4   5
  / \
 1   2
给定的树 B:

   4
  /
 1
返回 true，因为 B 与 A 的一个子树拥有相同的结构和节点值。
```

示例 1：

```
输入：A = [1,2,3], B = [3,1]
输出：false
```

示例 2：

```
输入：A = [3,4,5,1,2], B = [4,1]
输出：true
```

限制：

```
0 <= 节点个数 <= 10000
```

通过次数29,795提交次数64,231

解法

这道题做的时候需要搞清楚是判断树的子结构还是树的子树。这里是子结构，也就是说判断A中有没有一颗子树包含B，并且这颗子树可能有不属于B的其他叶子节点。

这道题我的思路是首先在A中寻找与B的根节点值相等的节点，然后在判断以这个节点为根的子树中是否包含B。下面是具体的Java实现：

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        if(B==null || A==null){
            return false;
        }

        if(A.val==B.val){
```

```
            boolean find = isSubStructureRoot(A, B);
            if(find){
                return true;
            }
        }

        return isSubStructure(A.left, B) || isSubStructure(A.right, B);
    }

    // A和B树根相同的情况下，A是否含有B
    public boolean isSubStructureRoot(TreeNode A, TreeNode B){
        if(A==null && B!=null){
            return false;
        }
        if(B==null){
            return true;
        }
        if(A.val!=B.val){
            return false;
        }

        boolean a = isSubStructureRoot(A.left, B.left);
        if(!a){
            return false;
        }
        boolean b = isSubStructureRoot(A.right, B.right);
        if(!b){
            return false;
        }

        return true;
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：0 ms, 在所有 Java 提交中击败了100.00%的用户
内存消耗：41.5 MB, 在所有 Java 提交中击败了100.00%的用户
```

27 二叉树的镜像

```
@author: sdubrz
@date: 7/25/2020 4:18:02 PM
难度： 简单
考察内容： 二叉树
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

请完成一个函数，输入一个二叉树，该函数输出它的镜像。

例如输入：

```
     4
   /   \
  2     7
 / \   / \
1   3 6   9
```

镜像输出：

```
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

示例 1：

```
输入：root = [4,2,7,1,3,6,9]
输出：[4,7,2,9,6,3,1]
```

限制：

```
0 <= 节点个数 <= 1000
```

解法

这道题比较简单了，递归大法好：

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if(root==null){
            return root;
        }

        TreeNode temp = root.left;
        root.left = mirrorTree(root.right);
        root.right = mirrorTree(temp);
        return root;
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：0 ms，在所有 Java 提交中击败了100.00%的用户
内存消耗：37.1 MB，在所有 Java 提交中击败了100.00%的用户
```

28 对称的二叉树

```
@author: sdubrz
@date: 7/25/2020 11:53:59 PM
难度： 简单
考察内容： 二叉树
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

例如，二叉树 `[1,2,2,3,4,4,3]` 是对称的。

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

但是下面这个 `[1,2,2,null,3,null,3]` 则不是镜像对称的:

```
    1
   / \
  2   2
   \   \
   3    3
```

示例 1：

```
输入：root = [1,2,2,3,4,4,3]
输出：true
```

示例 2：

```
输入：root = [1,2,2,null,3,null,3]
输出：false
```

限制：

- 0 <= 节点个数 <= 1000

解法

对于这道题，当二叉树的根节点非空时，我们只需要判断根节点的左子树和右子树是否对称就可以了。而当这两个子树都不为空时，只需要判断左子树的左子树与右子树的右子树是否对称，以及左子树的右子树和右子树的左子树是否对称就可以了。如此，是一个递归的过程。

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if(root==null){
            return true;
        }
        if(root.left==null && root.right==null){
            return true;
        }

        return isSymmetric(root.left, root.right);
    }

    // 判断两颗子树是不是对称的
    public boolean isSymmetric(TreeNode leftTree, TreeNode rightTree){
        if(leftTree==null && rightTree==null){
            return true;
        }
        if(leftTree==null || rightTree==null){
            return false;
        }
        if(leftTree.val != rightTree.val){
            return false;
        }

        return isSymmetric(leftTree.left, rightTree.right) &&
isSymmetric(leftTree.right, rightTree.left);

    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：0 ms，在所有 Java 提交中击败了100.00%的用户
内存消耗：38 MB，在所有 Java 提交中击败了100.00%的用户
```

29 顺时针打印矩阵

```
@author: sdubrz
@date:  7/31/2020 10:50:03 AM
难度： 简单
考察内容： 矩阵
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

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。

示例 1：

```
输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[1,2,3,6,9,8,7,4,5]
```

示例 2：

```
输入：matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
输出：[1,2,3,4,8,12,11,10,9,5,6,7]
```

限制：

- 0 <= matrix.length <= 100
- 0 <= matrix[i].length <= 100

解法

可以螺旋式的一层一层剥皮：

```
class Solution {
    public int[] spiralOrder(int[][] matrix) {

        int n = matrix.length;
        if(n==0) {
            return new int[0];
        }
        int m = matrix[0].length;

        int[] result = new int[n*m];
        int direct = 1;  // 1:left, 2:down, 3:left, 4:right

        int count = 0;
        // int start = 0;
        int n_loop = Math.min((n+1)/2, (m+1)/2);
        for(int start=0; start<n_loop; start++) {
            // 一圈的上面一行
            for(int i=start; i<m-start; i++) {
                result[count] = matrix[start][i];
                count++;
            }
            if(count>=n*m) {
                break;
            }
            // 右边一列
            for(int i=start+1; i<n-start-1; i++) {
                result[count] = matrix[i][m-start-1];
                count++;
            }
            if(count>=n*m) {
                break;
            }
            // 下边一行
            for(int i=m-start-1; i>=start; i--) {
                result[count] = matrix[n-start-1][i];
```

```
                count++;
            }
            if(count>=n*m) {
                break;
            }
            // 左边一列
            for(int i=n-start-2; i>start; i--) {
                result[count] = matrix[i][start];
                count++;
            }
            if(count>=n*m) {
                break;
            }

        }

        return result;

    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：1 ms, 在所有 Java 提交中击败了97.04%的用户
内存消耗：40.9 MB, 在所有 Java 提交中击败了72.20%的用户
```

30 包含min函数的栈

```
@author: sdubrz
@date:  8/1/2020 10:48:07 PM
难度： 简单
考察内容： 栈
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

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

示例:

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.min();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.min();   --> 返回 -2.
```

提示:

- 各函数的调用总次数不超过 20000 次

解法

可以用两个栈实现，第一个栈用于正常存储数据，第二个栈依次存储当前遇到的最小值。下面是具体的Java程序实现：

```
import java.util.*;

class MinStack {
    Stack<Integer> stack1;
    Stack<Integer> stack2;

    /** initialize your data structure here. */
    public MinStack() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }

    public void push(int x) {
        stack1.push(x);
        if(stack2.isEmpty() || stack2.peek()>=x){
            stack2.push(x);
        }
    }

    public void pop() {
        int temp = stack1.pop();
        if(temp==stack2.peek()){
            stack2.pop();
        }
    }

    public int top() {
        return stack1.peek();
    }

    public int min() {
        return stack2.peek();
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack obj = new MinStack();
 * obj.push(x);
 * obj.pop();
 * int param_3 = obj.top();
 * int param_4 = obj.min();
 */
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过显示详情
执行用时：19 ms, 在所有 Java 提交中击败了85.82%的用户
内存消耗：41.5 MB, 在所有 Java 提交中击败了85.80%的用户
```

31 栈的压入、弹出序列

```
@author: sdubrz
@date:  8/2/2020 9:02:57 AM
难度： 中等
考察内容： 栈
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

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} 是某栈的压栈序列，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。

示例 1：

```
输入：pushed = [1,2,3,4,5], popped = [4,5,3,2,1]
输出：true
解释：我们可以按以下顺序执行：
push(1), push(2), push(3), push(4), pop() -> 4,
push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1
```

示例 2：

```
输入：pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
输出：false
解释：1 不能在 2 之前弹出。
```

提示：

- 0 <= pushed.length == popped.length <= 1000
- 0 <= pushed[i], popped[i] < 1000
- pushed 是 popped 的排列。

解法

最容易想到的解决方法就是用一个真实的栈来进行测试。时间复杂度是 `O(n)` 的。

```
class Solution {
    public boolean validateStackSequences(int[] pushed, int[] popped) {
        int n = pushed.length;
        int index2 = 0;

        Stack<Integer> stack = new Stack<>();
        for(int i=0; i<n; i++) {
            stack.push(pushed[i]);
            while(index2<n && (!stack.isEmpty()) &&
stack.peek()==popped[index2]) {
                stack.pop();
                index2++;
            }
        }

        if(stack.isEmpty()) {
```

```
            return true;
        }else {
            return false;
        }
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过显示详情
执行用时：3 ms，在所有 Java 提交中击败了82.39%的用户
内存消耗：39.6 MB，在所有 Java 提交中击败了12.36%的用户
```

32 从上到下打印二叉树

```
@author: sdubrz
@date:  8/2/2020 9:24:21 AM
难度： 中等
考察内容： 二叉树
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

例如:
给定二叉树: [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

返回:

```
[3,9,20,15,7]
```

提示:

```
节点总数 <= 1000
```

解法

这个问题其实是一个广度优先搜索，可以用一个队列来实现:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
```

```
 * }
 */
class Solution {
    public int[] levelOrder(TreeNode root) {
        if(root==null) {
            return new int[0];
        }
        Queue<TreeNode> queue = new LinkedList<>();
        ArrayList<Integer> list = new ArrayList<>();
        queue.add(root);

        while(!queue.isEmpty()) {
            TreeNode current = queue.poll();
            list.add(current.val);
            if(current.left!=null) {
                queue.add(current.left);
            }
            if(current.right!=null) {
                queue.add(current.right);
            }
        }

        int[] result = new int[list.size()];
        Iterator<Integer> iter = list.iterator();
        int index = 0;
        while(iter.hasNext()) {
            result[index] = (int) iter.next();
            index++;
        }
        return result;
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：1 ms，在所有 Java 提交中击败了99.65%的用户
内存消耗：40 MB，在所有 Java 提交中击败了35.23%的用户
```

32 从上到下打印二叉树II

```
@author: sdubrz
@date:  8/2/2020 9:42:54 AM
难度： 简单
考察内容： 二叉树
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。

例如:
给定二叉树: [3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其层次遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

提示：

- 节点总数 <= 1000

解法

这道题似乎比上一题要麻烦一点点，但不知道什么原因，LeetCode把上一题标记成中等难度，这道题简单难度。

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> list = new LinkedList<>();
        if(root==null) {
            return list;
        }

        LinkedList<TreeNode> currentList = new LinkedList<>();
        currentList.add(root);
        while(!currentList.isEmpty()) {
            Iterator<TreeNode> iter = currentList.iterator();
            LinkedList<TreeNode> nextList = new LinkedList<>();
            // int[] currentArray = new int[currentList.size()];
            List<Integer> subList = new LinkedList<>();
            int index = 0;
            while(iter.hasNext()) {
                TreeNode current = iter.next();
                subList.add(current.val);
                if(current.left!=null) {
                    nextList.add(current.left);
                }
                if(current.right!=null) {
                    nextList.add(current.right);
```

```
                }
            }
            list.add(subList);
            currentList = nextList;
        }

        return list;
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：1 ms, 在所有 Java 提交中击败了93.41%的用户
内存消耗：39.9 MB, 在所有 Java 提交中击败了60.66%的用户
```

33 二叉搜索树的后序遍历序列

```
@author: sdubrz
@date:  8/2/2020 10:46:58 AM
难度： 中等
考察内容： 二叉搜索树
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

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结果。如果是则返回 true，否则返回 false。假设输入的数组的任意两个数字都互不相同。

参考以下这颗二叉搜索树：

```
     5
    / \
   2   6
  / \
 1   3
```

示例 1：

```
输入: [1,6,3,2,5]
输出: false
```

示例 2：

```
输入: [1,3,2,6,5]
输出: true
```

提示：

- 数组长度 <= 1000

通过次数25,327 提交次数48,585

解法

二叉搜索树的后序遍历数组中可以分为三部分：

- 第一部分位于最左侧，是二叉搜索树根节点的左子树的元素，所有的元素都小于根节点。该部分可能为空。
- 第二部分位于第一部分的左侧，是二叉搜索树根节点的右子树中的所有元素，均大于根节点。该部分可能为空。
- 第三部分是最后一个元素，也就是二叉搜索树的根节点元素。

所以，这道题类似于快速排序中的思想，只需要判断给定的数组中，所有比根节点小的元素均位于比根节点大的元素的左侧就可以了。用递归的思想来做，时间复杂度为 `O(nlog(n))`。需要注意的是，当数组中的元素不足3个的时候，一定是某个二叉搜索树的后序遍历。以下是具体的Java程序实现：

```
class Solution {
    public boolean verifyPostorder(int[] postorder) {
        int n = postorder.length;
        if(n<3) {
            return true;
        }

        return verifyPostorder(postorder, 0, n-1);
    }

    private boolean verifyPostorder(int[] postorder, int head, int tail) {
        if(tail-head<2) {
            return true;
        }

        int ptr1 = head-1;
        int ptr2 = tail;
        while(ptr1+1<tail && postorder[ptr1+1]<postorder[tail]) {
            ptr1++;
        }
        while(ptr2-1>=head && postorder[ptr2-1]>postorder[tail]) {
            ptr2--;
        }

        if(ptr1==ptr2-1) {
            return verifyPostorder(postorder, head, ptr1) &&
verifyPostorder(postorder, ptr2, tail-1);
        }else {
            return false;
        }
    }
}
```

在 LeetCode 系统中提交的结果为

```
执行结果：通过 显示详情
执行用时：0 ms, 在所有 Java 提交中击败了100.00%的用户
内存消耗：37.2 MB, 在所有 Java 提交中击败了34.81%的用户
```

34 二叉树中和为某一值的路径

```
@author: sdubrz
@date:  8/2/2020 2:54:10 PM
难度： 中等
考察内容： 二叉树
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社
```

输入一棵二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径。

示例:
给定如下二叉树，以及目标和 sum = 22，

```
              5
             / \
            4   8
           /   / \
          11  13  4
         /  \    / \
        7    2  5   1
```

返回:

```
[
   [5,4,11,2],
   [5,8,4,5]
]
```

提示:

- 节点总数 <= 10000

解法

判断以 root 节点为根的二叉树中是否有和为 sum 的路径，就是判断 root 的左子树和右子树中是否有和为 sum-root.val 的路径。根据这一思路，可以写出递归的实现：

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> list = new LinkedList<>();
        if(root==null) {
            return list;
        }

        if(root.val==sum && root.left==null && root.right==null) {
            List<Integer> tempList = new LinkedList<>();
```

```
            tempList.add(root.val);
            list.add(tempList);
            return list;
        }

        if(root.left!=null) {
            List<List<Integer>> leftList = pathSum(root.left, sum-root.val);
            Iterator<List<Integer>> iter = leftList.iterator();
            while(iter.hasNext()) {
                List<Integer> tempList = iter.next();
                tempList.add(0, root.val);
                list.add(tempList);
            }
        }
        if(root.right!=null) {
            List<List<Integer>> rightList = pathSum(root.right, sum-root.val);
            Iterator<List<Integer>> iter = rightList.iterator();
            while(iter.hasNext()) {
                List<Integer> tempList = iter.next();
                tempList.add(0, root.val);
                list.add(tempList);
            }
        }


        return list;
    }
}
```

在 LeetCode 系统中提交的结果为

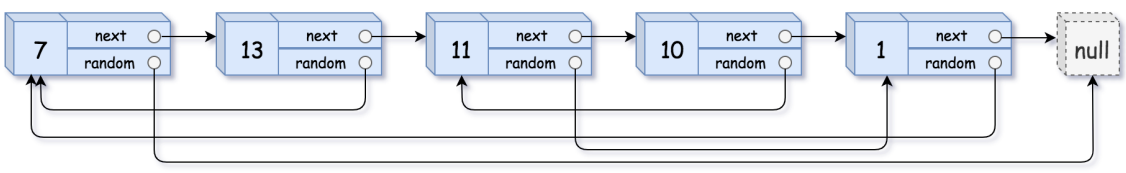
```
执行结果：通过 显示详情
执行用时：2 ms，在所有 Java 提交中击败了37.96%的用户
内存消耗：40 MB，在所有 Java 提交中击败了72.31%的用户
```

35 复杂链表的复制

```
@author: sdubrz
@date:  8/2/2020 3:22:32 PM
难度： 中等
考察内容： 链表
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社，图片来自LeetCode
```

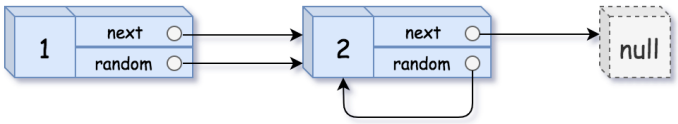
请实现 copyRandomList 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 next 指针指向下一个节点，还有一个 random 指针指向链表中的任意节点或者 null。

示例 1：



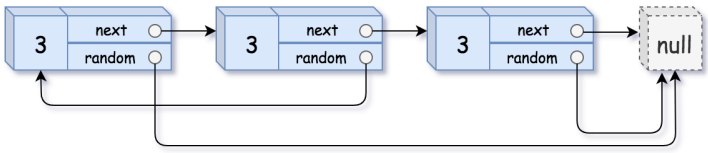
```
输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
```

示例 2：




```
输入：head = [[1,1],[2,1]]
输出：[[1,1],[2,1]]
```

示例 3：




```
输入：head = [[3,null],[3,0],[3,null]]
输出：[[3,null],[3,0],[3,null]]
```

示例 4：

```
输入：head = []
输出：[]
解释：给定的链表为空（空指针），因此返回 null。
```

提示：

- -10000 <= Node.val <= 10000
- Node.random 为空（null）或指向链表中的节点。
- 节点数目不超过 1000 。

一个错误的解法

先根据每个节点的 next 信息，复制所有的节点，然后根据节点的值逐个确定其 random 信息。这是一个时间复杂度为 `O(n2)` 的方法，并且当节点的取值存在重复时，无法正确确定每个节点的random信息。这是我第一时间想到的方法，提交之后运行结果错误。

```
/*
// Definition for a Node.
class Node {
    int val;
    Node next;
    Node random;

    public Node(int val) {
        this.val = val;
        this.next = null;
        this.random = null;
```

```
    }
}
*/
class Solution {
    public Node copyRandomList(Node head) {
        if(head==null) {
            return null;
        }

        Node headNode = new Node(head.val);
        Node current = headNode;
        Node ptrNode = head.next;
        while(ptrNode!=null) {
            Node temp = new Node(ptrNode.val);
            current.next = temp;
            current = current.next;
            ptrNode = ptrNode.next;
        }

        current = headNode;
        ptrNode = head;
        while(current!=null) {
            if(ptrNode.random==null) {
                ptrNode = ptrNode.next;
                current = current.next;
                continue;
            }

            Node temp = headNode;
            while(temp.val!=ptrNode.random.val) {
                temp = temp.next;
            }
            current.random = temp;
            ptrNode = ptrNode.next;
            current = current.next;
        }

        return headNode;
    }
}
```

第二种方法 借助Map

我想到的第二种思路是将两个链表上相同位置上的节点建立映射关系，可以用HashMap帮助实现：

```
/*
// Definition for a Node.
class Node {
    int val;
    Node next;
    Node random;

    public Node(int val) {
        this.val = val;
        this.next = null;
        this.random = null;
```

```
    }
}
*/
class Solution {
    public Node copyRandomList(Node head) {
        if(head==null) {
            return null;
        }

        HashMap<Node, Node> map = new HashMap<>();

        Node headNode = new Node(head.val);
        map.put(head, headNode);
        Node current = headNode;
        Node ptrNode = head.next;
        while(ptrNode!=null) {
            Node temp = new Node(ptrNode.val);
            current.next = temp;
            map.put(ptrNode, temp);
            current = current.next;
            ptrNode = ptrNode.next;
        }

        current = headNode;
        ptrNode = head;
        while(current!=null) {
            if(ptrNode.random!=null) {
                current.random = map.get(ptrNode.random);
            }
            current = current.next;
            ptrNode = ptrNode.next;
        }

        return headNode;
    }
}
```

在 LeetCode 系统中提交的结果为

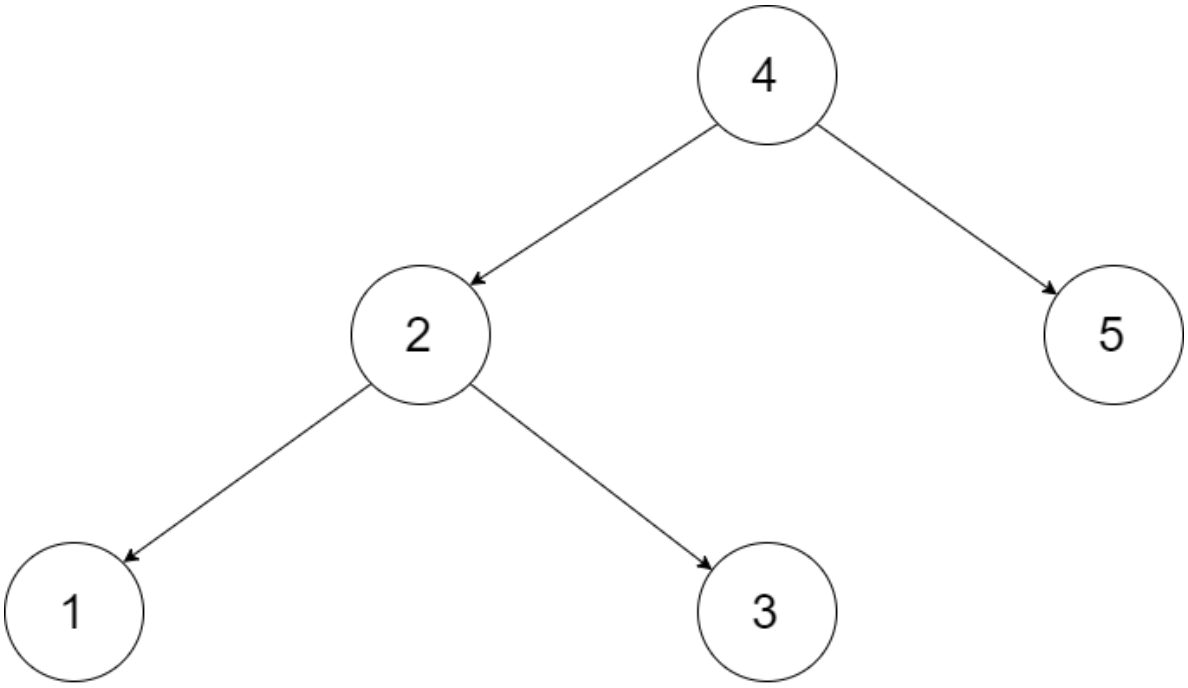
```
执行结果：通过 显示详情
执行用时：0 ms, 在所有 Java 提交中击败了100.00%的用户
内存消耗：39.3 MB, 在所有 Java 提交中击败了88.41%的用户
```

36 二叉搜索树与双向链表

```
@author: sdubrz
@date:  8/3/2020 4:22:22 PM
难度： 中等
考察内容： 二叉搜索树 链表
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社，图片来自LeetCode
```

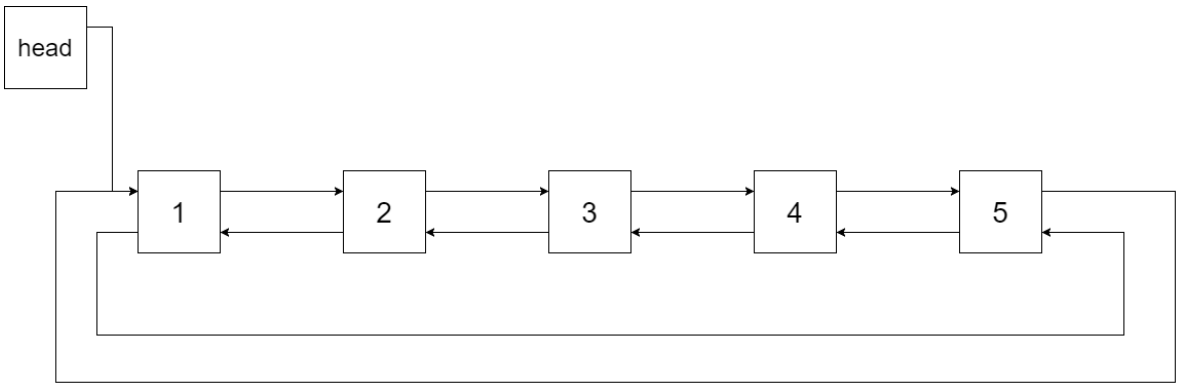
输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。

为了让您更好地理解问题，以下面的二叉搜索树为例：



我们希望将这个二叉搜索树转化为双向循环链表。链表中的每个节点都有一个前驱和后继指针。对于双向循环链表，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

下图展示了上面的二叉搜索树转化成的链表。“head” 表示指向链表中有最小元素的节点。



特别地，我们希望可以就地完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中的第一个节点的指针。

解法

一种比较容易理解的思路是使用递归。在得到左子树的链表与右子树的链表之后将其与根节点合并。下面是具体的Java程序实现：

```
/*
// Definition for a Node.
class Node {
    public int val;
    public Node left;
    public Node right;

    public Node() {}

    public Node(int _val) {
        val = _val;
    }
```

```
    public Node(int _val,Node _left,Node _right) {
        val = _val;
        left = _left;
        right = _right;
    }
};
*/
class Solution {
    public Node treeToDoublyList(Node root) {
        if(root==null) {
            return null;
        }

        //System.out.println("根节点为 "+root.val);

        if(root.left==null && root.right==null) {
            root.left = root;
            root.right = root;
            return root;
        }

        return treeToDoublyList0(root);
    }

    public Node treeToDoublyList0(Node root) {
        if(root==null) {
            return null;
        }

        //System.out.println("根节点为 "+root.val);

        if(root.left==null && root.right==null) {
            return root;
        }

        Node leftHead = null;
        Node rightHead = null;
        if(root.left!=null) {
            leftHead = treeToDoublyList(root.left);
        }
        if(root.right!=null) {
            rightHead = treeToDoublyList(root.right);
        }

        root.left = null;
        root.right = null;
        Node head = root;
        if(leftHead!=null) {
            if(leftHead.left!=null) {  // 不是叶子节点
                Node tempNode = leftHead.left;
                leftHead.left = root;
                root.left = tempNode;
                root.right = leftHead;
                tempNode.right = root;
            }else {  // 叶子节点
                leftHead.left = root;
                leftHead.right = root;
```

```
                root.left = leftHead;
                root.right = leftHead;
            }
            head = leftHead;
        }

//        System.out.println("根节点为 "+root.val+" 时，加上左子树的结果");
//        printList(head);

        if(rightHead!=null) {
            if(head.right==null) {  // 左子树为空
                if(rightHead.left!=null) {  // 不是叶子节点
                    Node tempNode = rightHead.left;
                    rightHead.left = head;
                    head.right = rightHead;
                    head.left = tempNode;
                    tempNode.right = head;
                }else {  // 是叶子节点
                    head.right = rightHead;
                    head.left = rightHead;
                    rightHead.left = head;
                    rightHead.right = head;
                }
            }else {  // 左子树不为空
                if(rightHead.left!=null) {  // 不是叶子节点
                    Node tempNode1 = head.left;
                    Node tempNode2 = rightHead.left;
                    head.left = tempNode2;
                    rightHead.left = tempNode1;
                    tempNode1.right = rightHead;
                    tempNode2.right = head;
                    //System.out.println("RightHead = "+rightHead.val);
                }else {  // 是叶子节点
                    Node tempNode = head.left;
                    head.left = rightHead;
                    rightHead.right = head;
                    rightHead.left = tempNode;
                    tempNode.right = rightHead;
                    //System.out.println("rightHead = "+rightHead.val);
                }
            }
        }

//        System.out.println("根节点为 "+root.val+" 时，加上右子树的结果");
//        printList(head);

        return head;
    }
}
```

下面是在LeetCode系统中提交的结果

```
执行结果：通过 显示详情
执行用时：1 ms, 在所有 Java 提交中击败了18.34%的用户
内存消耗：39.6 MB, 在所有 Java 提交中击败了11.89%的用户
```

这个方法效率不高，其实中序遍历是更好的实现思路。

37 序列化二叉树

```
@author: sdubrz
@date:  8/3/2020 6:47:45 PM
难度： 困难
考察内容： 二叉树，字符串
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社，图片来自LeetCode
```

请实现两个函数，分别用来序列化和反序列化二叉树。

示例:

```
你可以将以下二叉树：

    1
   / \
  2   3
     / \
    4   5

序列化为 "[1,2,3,null,null,4,5]"
```

我的解法

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Codec {

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        // 需要先得知树的层数
        if(root==null) {
            return "[]";
        }
        Queue<TreeNode> queue1 = new LinkedList<>();
        queue1.add(root);
        int level = 0;
        while(!queue1.isEmpty()) {
            Queue<TreeNode> queue2 = new LinkedList<>();
            while(!queue1.isEmpty()) {
                TreeNode current = queue1.poll();
                if(current.left!=null) {
                    queue2.add(current.left);
                }
                if(current.right!=null) {
                    queue2.add(current.right);
```

```
                }
            }
            queue1 = queue2;
            level++;
        }

        int n = (int) Math.pow(2, level) - 1;
        int[] nums = new int[n];
        int[] labels = new int[n];

        serialize(root, nums, labels, 0);
        String data = "[";
        for(int i=0; i<n-1; i++) {
            if(labels[i]==1) {
                data = data + nums[i] + ",";
            }else {
                data = data + "null" + ",";
            }
        }
        if(labels[n-1]==1) {
            data = data + nums[n-1];
        }else {
            data = data + "null";
        }
        data = data + "]";


        return data;
    }

    private void serialize(TreeNode root, int[] nums, int[] labels, int index) {
        if(root==null) {
            return;
        }

        nums[index] = root.val;
        labels[index] = 1;

        int leftIndex = index*2+1;
        int rightIndex = index*2+2;
        if(leftIndex<nums.length && root.left!=null) {
            serialize(root.left, nums, labels, leftIndex);
        }
        if(rightIndex<nums.length && root.right!=null) {
            serialize(root.right, nums, labels, rightIndex);
        }
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if(data.length()<3) {
            return null;
        }
        String str = data.substring(1, data.length()-1);
        String[] items = str.split(",");
        int n = items.length;
        int[] nums = new int[n];
        int[] labels = new int[n];
```

```
        for(int i=0; i<n; i++) {
            if(!items[i].equals("null")) {
                nums[i] = Integer.parseInt(items[i]);
                labels[i] = 1;
            }
        }

        if(labels[0]!=1) {
            return null;
        }

        TreeNode root = new TreeNode(nums[0]);
        root.left = getLeft(nums, labels, 0);
        root.right = getRight(nums, labels, 0);

        return root;
    }

    // 生成左孩子
    private TreeNode getLeft(int[] nums, int[] labels, int index) {
        int leftIndex = index*2+1;
        if(leftIndex>=nums.length || labels[leftIndex]==0) {
            return null;
        }

        TreeNode root = new TreeNode(nums[leftIndex]);
        root.left = getLeft(nums, labels, leftIndex);
        root.right = getRight(nums, labels, leftIndex);
        return root;
    }

    // 生成右孩子
    private TreeNode getRight(int[] nums, int[] labels, int index) {
        int rightIndex = index*2+2;
        if(rightIndex>=nums.length || labels[rightIndex]==0) {
            return null;
        }

        TreeNode root = new TreeNode(nums[rightIndex]);
        root.left = getLeft(nums, labels, rightIndex);
        root.right = getRight(nums, labels, rightIndex);
        return root;
    }
}

// Your Codec object will be instantiated and called as such:
// Codec codec = new Codec();
// codec.deserialize(codec.serialize(root));
```

提交后结果显示超时。

网友的解法

```
public class Codec {
    public String serialize(TreeNode root) {
        if(root == null) return "[]";
```

```
        StringBuilder res = new StringBuilder("[");
        Queue<TreeNode> queue = new LinkedList<>() {{ add(root); }};
        while(!queue.isEmpty()) {
            TreeNode node = queue.poll();
            if(node != null) {
                res.append(node.val + ",");
                queue.add(node.left);
                queue.add(node.right);
            }
            else res.append("null,");
        }
        res.deleteCharAt(res.length() - 1);
        res.append("]");
        return res.toString();
    }

    public TreeNode deserialize(String data) {
        if(data.equals("[]")) return null;
        String[] vals = data.substring(1, data.length() - 1).split(",");
        TreeNode root = new TreeNode(Integer.parseInt(vals[0]));
        Queue<TreeNode> queue = new LinkedList<>() {{ add(root); }};
        int i = 1;
        while(!queue.isEmpty()) {
            TreeNode node = queue.poll();
            if(!vals[i].equals("null")) {
                node.left = new TreeNode(Integer.parseInt(vals[i]));
                queue.add(node.left);
            }
            i++;
            if(!vals[i].equals("null")) {
                node.right = new TreeNode(Integer.parseInt(vals[i]));
                queue.add(node.right);
            }
            i++;
        }
        return root;
    }
}
```

作者：jyd
链接：https://leetcode-cn.com/problems/xu-lie-hua-er-cha-shu-lcof/solution/mian-shi-ti-37-xu-lie-hua-er-cha-shu-ceng-xu-bian-/
来源：力扣（LeetCode）

42 连续子数组的最大和

```
@author: sdubrz
@date:  8/3/2020 6:47:45 PM
难度：  8/4/2020 10:56:03 PM
考察内容：  动态规划，分而治之
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社，图片来自LeetCode
```

输入一个整型数组，数组中的一个或连续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)。

示例1:

```
输入: nums = [-2,1,-3,4,-1,2,1,-5,4]
输出: 6
解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。
```

提示:

- 1 <= arr.length <= 10^5
- -100 <= arr[i] <= 100

动态规划解法

凡是符合条件的子数组必定存在最后一个元素，并且最后这个元素肯定是所给的数组中的某一个元素。所以我们只需要知道以每个数组结尾的所有子数组中和最大的即可。

假设以第`i`个元素结尾的子数组中和最大的值为`f(i)`。则有`f(0)=nums[0]`。当`i>0`时，如果`f(i-1)>0`则`f(i)=f(i-1)+nums[i]`，而如果`f(i-1)<=0`，则`f(i)=nums[i]`。最后，我们从`f(0...n-1)`中选出最大的返回即可。据此可以写出时间复杂度为`O(n)`的动态规划实现：

```
class Solution {
    public int maxSubArray(int[] nums) {
        if(nums.length==1){
            return nums[0];
        }

        int n = nums.length;
        int[] count = new int[n];
        count[0] = nums[0];
        int result = count[0];

        for(int i=1; i<n; i++){
            if(count[i-1]<0){
                count[i] = nums[i];
            }else{
                count[i] = count[i-1] + nums[i];
            }
            if(result<count[i]){
                result = count[i];
            }
        }

        return result;
    }
}
```

在 LeetCode 系统中提交的结果为：

```
执行结果：通过 显示详情
执行用时：1 ms, 在所有 Java 提交中击败了99.32%的用户
内存消耗：46.4 MB, 在所有 Java 提交中击败了58.18%的用户
```

分治解法

在《算法导论》中给出了一个时间复杂度同样为`O(n)`的分治解法。《算法导论》第4章 4.1节讲的最大子数组问题正文中给出了时间复杂度为`O(nlog(n))`的分治方法，在课后练习题4.1-5中又给出了线性复杂度的分治方法。其主要的思路是：对于一个数组的最大子数组，在这个最大子数组有多于两个元素的情况下，任意将这个最大子数组切分为前后两部分，这两部分各自的和必然是正的。这部分应该比较好理解，因为如果有一部分的和是负的或0，我们可以把这部分删掉，剩下的部分的和一定不小于原来的子数组。最大子数组的左右两部分的和都是正的，也就意味着左右两部分的和都要小于总的和。基于这个思路，可以写出如下代码

```
class Solution {
    public int maxSubArray(int[] nums) {
        int tempLeft = 0;
        int tempRight = 0;
        int left = 0;
        int right = 0;
        int sum = nums[0];
        int maxSum = nums[0];

        for(int i=1; i<nums.length; i++){
            if(sum+nums[i]>nums[i]){
                sum = sum + nums[i];
                tempRight = i;
            }else{
                sum = nums[i];
                tempLeft = i;
            }

            if(maxSum<sum){
                maxSum = sum;
                left = tempLeft;
                right = tempRight;
            }
        }

        return maxSum;
    }
}
```

在LeetCode系统中提交的结果，这种方法的空间复杂度应该比前面的动态规划方法小吧，但是不知道为什么提交结果显示反而不如动态规划。

```
执行结果：通过
执行用时 : 1 ms, 在所有 Java 提交中击败了 97.36% 的用户
内存消耗 : 42.1 MB, 在所有 Java 提交中击败了 5.40% 的用户
```

55 二叉树的深度

```
@author: sdubrz
@date:  8/9/2020 8:43:53 PM
难度： 简单
考察内容： 二叉树
@e-mail: lwyz521604#163.com
题目来自《剑指offer》 电子工业出版社，图片来自LeetCode
```

输入一棵二叉树的根节点，求该树的深度。从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度。

例如：

给定二叉树 [3,9,20,null,null,15,7]，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回它的最大深度 3 。

提示：

- 节点总数 <= 10000

广度优先搜索解法

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int maxDepth(TreeNode root) {
        if(root==null){
            return 0;
        }

        int depth = 0;
        Queue<TreeNode> queue1 = new LinkedList<>();
        queue1.add(root);
        while(!queue1.isEmpty()){
            depth++;
            Queue<TreeNode> queue2 = new LinkedList<>();
            while(!queue1.isEmpty()){
                TreeNode temp = queue1.poll();
                if(temp.left!=null){
                    queue2.add(temp.left);
                }
                if(temp.right!=null){
                    queue2.add(temp.right);
                }
            }
            queue1 = queue2;
        }

        return depth;
    }
}
```

提交结果为

```
执行用时：1 ms, 在所有 Java 提交中击败了21.41%的用户
内存消耗：39.2 MB, 在所有 Java 提交中击败了98.38%的用户
```