剑指Offer笔记

> 笔者：**Peter Ho**
>
> 版本：**Version 2.0**
>
> 完结日期：**2020年10月14日**
>
> 说明：**本笔记记录了笔者在刷《剑指Offer》题目过程中的心得体会以及代码注释的整理，便于后续复习，需结合Leetcode的剑指Offer题目使用。**
>
> 相关链接：**LeetCode** 、 **牛客网**

剑指Offer 33（二叉搜索树的后序遍历序列）
剑指Offer 34（树的路径和）
剑指Offer 35（链表拷贝）
剑指Offer 36（二叉搜索树与双向链表）
剑指Offer 37（序列化二叉树）
剑指Offer 38（不重复全排列）
剑指Offer 39（ 数组中出现次数超过一半的数字）
剑指Offer 40（最小的 K 个数）
剑指Offer 41（数据流中的中位数）
剑指Offer 42（连续子数组的最大和）
剑指Offer 43（1 ～n 整数中 1 出现的次数）
剑指Offer 44（数字序列中的某一位数字）
剑指Offer 45（把数组排成最小的数）
剑指Offer 46（把数字翻译成字符串）
剑指Offer 47（礼物的最大价值）
剑指Offer 48（最长不含重复字符的子字符串）
剑指Offer 49（丑数）
剑指Offer 50（第一个只出现一次的字符位置）
剑指Offer 51（数组中的逆序对）
剑指Offer 52（相交链表）
剑指Offer 53 - 1（有序数组同一元素的个数）
剑指Offer 53 - 2（0～n-1中缺失的数字）
剑指Offer 54（二叉搜索树的倒数第k大节点）
剑指Offer 55-1（树的深度）
剑指Offer 55-2（树的平衡）
剑指Offer 56-1（数组中数字出现的次数）
剑指Offer 56-2（数组中数字出现的次数）
剑指Offer 57-1（和为 S 的两个数字）
剑指Offer 57-2（和为 S 的连续正数序列）
剑指Offer 58-1（翻转单词顺序）
剑指Offer 58-2（左旋转字符串）
剑指Offer 59-1（滑动窗口的最大值）
剑指Offer 59-2（队列的最大值）
剑指Offer 60（n 个骰子的点数）
剑指Offer 61（扑克牌顺子）
剑指Offer 62（圆圈中最后剩下的数）
剑指Offer 63（股票的最大利润）
剑指Offer 64（求 1+2+3+...+n）
剑指Offer 65（不用加减乘除做加法）
剑指Offer 66（构建乘积数组）
剑指Offer 67（把字符串转换成整数）
剑指Offer 68-1（二叉搜索树最近公共父节点）
剑指Offer 68-2（二叉树最近公共父节点）

剑指Offer 03（数组中的重复数字）

找出数组中重复的数字。

在一个长度为 n 的数组 nums 里的所有数字都在 0～n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

示例 1：

```
输入：
[2, 3, 1, 0, 2, 5, 3]
输出：2 或 3
```

限制：

`2 <= n <= 100000`

```java
// 只用了访问标记数组，思路简单
class Solution {
    public int findRepeatNumber(int[] nums) {
        int len = nums.length;
        boolean[] flag = new boolean[len];
        for (int num : nums) {
            if (flag[num]) {
                return num;
            } else{
                flag[num] = true;
            }
        }
        return 0;
    }
}
```

剑指Offer 04（二维数组的查找）

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`

```java
// 站在右上角看。这个矩阵其实就像是一个Binary Search Tree。然后，聪明的大家应该知道怎么做了。
class Solution {
    public boolean findNumberIn2DArray(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0) {
            return false;
        }
        int row = matrix.length, col = matrix[0].length;
        int curRow = 0, curCol = col - 1;
        while (curRow < row && curCol >=0){
            if (matrix[curRow][curCol] == target){
                return true;
            }else if (matrix[curRow][curCol] < target){
                curRow++;
            }else {
                curCol--;
            }
        }
        return false;
    }
}
```

剑指Offer 05（替换空格）

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`

```
// 写得有点简单了
class Solution {
    public String replaceSpace(String s) {
        StringBuffer stringBuffer = new StringBuffer();
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == ' ')
                stringBuffer.append("%20");
            else
                stringBuffer.append(s.charAt(i));
        }
        return stringBuffer.toString();
    }
}
```

剑指Offer 06（从头到尾打印链表）

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`

```
// 注意边界值就好
class Solution {
    public int[] reversePrint(ListNode head) {
        ListNode node = head;
        int len = 0;
        while (node != null){
            node = node.next;
            len++;
        }
        int[] ans = new int[len];
        for (int i = len - 1; i >= 0; i--) {
```

```
            ans[i] = head.val;
            head = head.next;
        }
        return ans;
    }
}
```

剑指Offer 07（重构二叉树）

输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。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`

```
// 不会，没理解
class Solution {
    //利用原理,先序遍历的第一个节点就是根。在中序遍历中通过根 区分哪些是左子树的，哪些是右子树的
    //左右子树，递归
    HashMap<Integer, Integer> map = new HashMap<>();//标记中序遍历
    int[] preorder;//保留的先序遍历

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        this.preorder = preorder;
        for (int i = 0; i < preorder.length; i++) {
            map.put(inorder[i], i);
        }
        return recursive(0,0,inorder.length - 1);
    }

     /**
     * @param pre_root_idx  先序遍历的索引
     * @param in_left_idx  中序遍历的索引
     * @param in_right_idx 中序遍历的索引
     */
```

```
    public TreeNode recursive(int pre_root_idx, int in_left_idx, int
in_right_idx) {
        //相等就是自己
        if (in_left_idx > in_right_idx) {
            return null;
        }
        //root_idx是在先序里面的
        TreeNode root = new TreeNode(preorder[pre_root_idx]);
        // 有了先序的,再根据先序的，在中序中获 当前根的索引
        int idx = map.get(preorder[pre_root_idx]);

        //左子树的根节点就是 左子树的(前序遍历）第一个，就是+1,左边边界就是left，右边边界是
中间区分的idx-1
        root.left = recursive(pre_root_idx + 1, in_left_idx, idx - 1);

        //由根节点在中序遍历的idx 区分成2段,idx 就是根

        //右子树的根，就是右子树（前序遍历）的第一个,就是当前根节点 加上左子树的数量
        // pre_root_idx 当前的根  左子树的长度 = 左子树的左边-右边 (idx-1 -
in_left_idx +1) 。最后+1就是右子树的根了
        root.right = recursive(pre_root_idx + (idx-1 - in_left_idx +1)  + 1,
idx + 1, in_right_idx);
        return root;
    }
}
```

剑指Offer 09（用两个栈实现队列）

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

```
// 很简单的思路，但只击败10%
```

```java
class CQueue {
    Stack<Integer> mainQueue;
    Stack<Integer> temp;

    public CQueue() {
        mainQueue = new Stack<>();
        temp = new Stack<>();
    }
    public void appendTail(int value) {
        while (!mainQueue.isEmpty()){
            temp.push(mainQueue.pop());
        }
        mainQueue.push(value);
        while (!temp.isEmpty()){
            mainQueue.push(temp.pop());
        }
    }

    public int deleteHead() {
        if(mainQueue.isEmpty()){
            return -1;
        }
        return mainQueue.pop();
    }
}
```

剑指Offer 10-1（斐波那契数列）

写一个函数，输入 `n` ，求斐波那契（Fibonacci）数列的第 `n` 项。斐波那契数列的定义如下：

```
F(0) = 0,   F(1) = 1
F(N) = F(N - 1) + F(N - 2), 其中 N > 1.
```

斐波那契数列由 0 和 1 开始，之后的斐波那契数就是由之前的两数相加而得出。

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：1
```

示例 2：

```
输入：n = 5
输出：5
```

提示：

- `0 <= n <= 100`

```java
// 别读错题，要模的
class Solution {
    public int fib(int n) {
        if (n == 0 || n == 1) return n;
        int[] dp = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = (dp[i - 1] + dp[i - 2]) % 1000000007;
        }
        return dp[n];
    }
}
```

剑指Offer 10-2（青蛙跳台阶）

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
```

示例 3：

```
输入：n = 0
输出：1
```

提示：

- `0 <= n <= 100`

```java
// 斐波那契小改
class Solution {
    public int numWays(int n) {
        if (n == 0 || n == 1) return 1;
        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = (dp[i - 1] + dp[i - 2]) % 1000000007;
        }
        return dp[n];
    }
}
```

剑指Offer 11（旋转数组中的最小数字）

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
```

```java
// 无语
class Solution {
    public int minArray(int[] numbers) {
        Arrays.sort(numbers);
        return numbers[0];
    }
}
// 二分法
class Solution {
    public int minArray(int[] numbers) {
        int l = 0, r = numbers.length - 1;
        while (l < r) {
            int mid = ((r - l) >> 1) + l;
            //只要右边比中间大，那右边一定是有序数组
            if (numbers[r] > numbers[mid]) {
                r = mid;
            } else if (numbers[r] < numbers[mid]) {
                l = mid + 1;
             //去重
            } else r--;
        }
        return numbers[l];
    }
}
```

剑指Offer 12（矩阵单词查找）

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
```

```
class Solution {
    private int[][] direction = new int[][]{{1, 0}, {0, 1}, {-1, 0}, {0, -1}};
    private int row;
    private int col;

    public boolean exist(char[][] board, String word) {
        if (board == null || board.length ==0)
            return false;
        row = board.length;
        col = board[0].length;

        boolean[][] vis = new boolean[row][col];
        for (int r = 0; r < row; r++) {
            for (int c = 0; c < col; c++) {
                if (dfs(0, r, c, vis, board, word)){
                    return true;
                }
            }
        }
        return false;
    }

    private boolean dfs(int curLen, int r, int c, boolean[][] vis, char[][]
board, String word){
        // 长度相等return true
        if (curLen == word.length())
            return true;
        // 边界过滤条件
        // 判断单词的此位是否相等
        if (r < 0 || r >= row || c < 0 || c >= col || board[r][c] !=
word.charAt(curLen) || vis[r][c]) {
            return false;
        }
        // 标记为已访问
```

```
            vis[r][c] = true;
            // 对四个方向回溯
            for (int[] d : direction){
                if (dfs(curLen + 1, r + d[0], c + d[1], vis, board, word))
                    return true;
            }
            // 下一步要return，所以要除去访问标记
            vis[r][c] = false;
            return false;
        }
    }
```

剑指Offer 13（机器人的运动范围）

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`

```
class Solution {
    boolean[][] visit;
    public int movingCount(int m, int n, int k) {
        // 第一步：先明确递归参数
        // 第二步：明确递归终止条件
        visit = new boolean[m][n];
        return dfs(0, 0, m, n, k);
    }

    private int dfs(int row, int col, int m, int n, int k) {
        // 第一步：先明确递归参数
        int a = sums(row);
        int b = sums(col);
        // 第二步：明确递归终止条件
        // 边界判断、位数和与k比较、当前点是否访问过
        if (row < 0 || row >= m || col < 0 || col >= n || k < a + b ||
visit[row][col]) {
            return 0;
```

```
        }
        // 第三步：递推工作
        visit[row][col] = true;
        return 1 + dfs(row + 1, col, m, n, k) + dfs(row, col + 1, m, n, k);
    }

    // 位数之和计算
    private int sums(int num) {
        int sums = 0;
        while (num != 0) {
            sums += num % 10;
            num = num / 10;
        }
        return sums;
    }
}
```

剑指Offer 14-1（剪绳子1）

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`

```
// 动态规划
class Solution {
    public int cuttingRope(int n) {
        int[] dp = new int[n + 1];
        dp[1] = 1;
        for (int i = 2; i <= n; i++)
            for (int j = 1; j < i; j++)
                dp[i] = Math.max(dp[i], Math.max(j * (i - j), dp[j] * (i -
j)));
        return dp[n];
    }
}
//递归，反而更快
class Solution {
```

```
    public int cuttingRope(int n) {
        if(n == 2) return 1;
        if(n == 3) return 2;
        if(n == 4) return 4;
        if(n == 5) return 6;
        if(n == 6) return 9;
        return 3 * cuttingRope(n - 3);
    }
}
```

剑指Offer 14-2（剪绳子2）

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。

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：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 <= 1000`

```
// 再看看吧
class Solution {
    public int cuttingRope(int n) {
        if(n <= 3) return n - 1;
        long res = 1L;
        int p = (int) 1e9+7;
        //贪心算法，优先切三，其次切二
        while(n>4){
            res = res * 3 % p;
            n -= 3;
        }
        //出来循环只有三种情况，分别是n=2、3、4
        return (int)(res * n % p);
    }
}
```

剑指Offer 15（二进制中 1 的个数）

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'。
```

```java
// 位运算
class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int count = 0;
        while (n != 0) {
            // 如果是0结尾，向前借1，就会导致‘与’运算之后相较于运算之前相比少一个1，如果是1同样也导致运算之后相较于运算之前少一个1。循环往复。
            count += n & 1;
            n = n >>> 1;
        }
        return count;
    }
}

class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        return Integer.bitCount(n);
    }
}
```

剑指Offer 16（数值的整数次方）

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 $[-2^{31}, 2^{31} - 1]$ 。

```java
// cv大法
// 递归
class Solution {
    public double myPow(double x, int n) {
        if(n == 0) return 1;
        if(n == 1) return x;
        if(n == -1) return 1 / x;
        double half = myPow(x, n / 2);
        double mod = myPow(x, n % 2);
        return half * half * mod;
    }
}

//
class Solution {
    public double myPow(double x, int n) {
        if(x == 0) return 0;
        long b = n;
        double res = 1.0;
        if(b < 0) {
            x = 1 / x;
            b = -b;
        }
        while(b > 0) {
            if((b & 1) == 1) res *= x;
            x *= x;
            b >>= 1;
        }
        return res;
    }
}
```

剑指Offer 17（打印数组）

输入数字 `n`，按顺序打印出从 1 到最大的 n 位十进制数。比如输入 3，则打印出 1、2、3 一直到最大的 3 位数 999。

示例 1:

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

说明:

- 用返回一个整数列表来代替打印
- n 为正整数

```java
// 这个太简单，考虑一下大数问题
class Solution {
    public int[] printNumbers(int n) {
        int limit = (int) Math.pow(10, n) - 1;
        int[] ans = new int[limit];
        for (int i = 0; i < limit; i++) {
            ans[i] = i + 1;
        }
        return ans;
    }
}
```

剑指Offer 18（在 O(1) 时间内删除链表节点）

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

```java
// 利用了两个指针记录
class Solution {
```

```
    public ListNode deleteNode(ListNode head, int val) {
        ListNode curr = head;
        ListNode pre = new ListNode(-1);
        if (head.val == val)
            return head.next;
        while (curr != null){
            if (curr.val == val){
                pre.next = curr.next;
                break;
            }
            pre = curr;
            curr = curr.next;
        }
        return head;
    }
}
```

剑指Offer 19（正则表达式匹配）

请实现一个函数用来匹配包含 `'.'` 和 `'*'` 的正则表达式。模式中的字符 `'.'` 表示任意一个字符，而 `'*'` 表示它前面的字符可以出现任意次（含0次）。在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串 `"aaa"` 与模式 `"a.a"` 和 `"ab*ac*a"` 匹配，但与 `"aa.a"` 和 `"ab*a"` 均不匹配。

示例 1:

```
输入:
s = "aa"
p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。
```

示例 2:

```
输入:
s = "aa"
p = "a*"
输出: true
解释: 因为 '*' 代表可以匹配零个或多个前面的那一个元素, 在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。
```

示例 3:

```
输入:
s = "ab"
p = ".*"
输出: true
解释: ".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。
```

示例 4:

```
输入:
s = "aab"
p = "c*a*b"
输出: true
解释: 因为 '*' 表示零个或多个，这里 'c' 为 0 个, 'a' 被重复一次。因此可以匹配字符串 "aab"。
```

示例 5:

```
输入:
s = "mississippi"
p = "mis*is*p*."
输出: false
```

- `s` 可能为空，且只包含从 `a-z` 的小写字母。
- `p` 可能为空，且只包含从 `a-z` 的小写字母以及字符 `.` 和 `*`，无连续的 `'*'`。

```
// 直接放弃
class Solution {
    public boolean isMatch(String s, String p) {
        int n = s.length();
        int m = p.length();
        boolean[][] f = new boolean[n + 1][m + 1];

        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= m; j++) {
                //分成空正则和非空正则两种
                if (j == 0) {
                    f[i][j] = i == 0;
                } else {
                    //非空正则分为两种情况 * 和 非*
                    if (p.charAt(j - 1) != '*') {
                        if (i > 0 && (s.charAt(i - 1) == p.charAt(j - 1) ||
p.charAt(j - 1) == '.')) {
                            f[i][j] = f[i - 1][j - 1];
                        }
                    } else {
                        //碰到 * 了，分为看和不看两种情况
                        //不看
                        if (j >= 2) {
                            f[i][j] |= f[i][j - 2];
                        }
                        //看
                        if (i >= 1 && j >= 2 && (s.charAt(i - 1) == p.charAt(j
- 2) || p.charAt(j - 2) == '.')) {
                            f[i][j] |= f[i - 1][j];
                        }
                    }
                }
            }
        }
        return f[n][m];
    }
}
```

剑指Offer 20（表示数值的字符串）

请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100"、"5e2"、"-123"、"3.1416"、"-1E-16"、"0123"都表示数值，但"12e"、"1a3.14"、"1.2.3"、"+-5"及"12e+5.4"都不是。

```
// cv大法
class Solution {
    public boolean isNumber(String s) {
        if(s == null || s.length() == 0){
```

```
            return false;
        }
        //标记是否遇到相应情况
        boolean numSeen = false;
        boolean dotSeen = false;
        boolean eSeen = false;
        char[] str = s.trim().toCharArray();
        for(int i = 0;i < str.length; i++){
            if(str[i] >= '0' && str[i] <= '9'){
                numSeen = true;
            }else if(str[i] == '.'){
                //.之前不能出现.或者e
                if(dotSeen || eSeen){
                    return false;
                }
                dotSeen = true;
            }else if(str[i] == 'e' || str[i] == 'E'){
                //e之前不能出现e，必须出现数
                if(eSeen || !numSeen){
                    return false;
                }
                eSeen = true;
                numSeen = false;//重置numSeen，排除123e或者123e+的情况,确保e之后也出
现数
            }else if(str[i] == '-' || str[i] == '+'){
                //+-出现在0位置或者e/E的后面第一个位置才是合法的
                if(i != 0 && str[i-1] != 'e' && str[i-1] != 'E'){
                    return false;
                }
            }else{//其他不合法字符
                return false;
            }
        }
        return numSeen;
    }
}
```

剑指Offer 21（调整数组顺序使奇数位于偶数前面）

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. `1 <= nums.length <= 50000`
2. `1 <= nums[i] <= 10000`

```java
// 双指针法交换，太慢了
class Solution {
    public int[] exchange(int[] nums) {
        // 双指针法
        int left = 0, right = nums.length - 1;
        int temp;
        while (left < right){
             if (isOdd(nums[left])) {// 前奇
                left++;
            }else if (!isOdd(nums[right])){// 后偶
                right--;
            }else {// 前偶后奇
                temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;
                left++;
                right--;
            }
        }
        return nums;
    }

    private boolean isOdd(int num){
        return num % 2 == 1;
    }
}
```

剑指Offer 22（链表中倒数第 K 个结点）

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.
```

```java
// 典型的双指针问题
class Solution {
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode p1 = head, p2 = head;
        for (int i = 1; i < k; i++) {
            p1 = p1.next;
        }
        while (p1.next != null){
            p1 = p1.next;
            p2 = p2.next;
        }
        return p2;
    }
}
```

剑指Offer 24（反转链表）

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`

```java
// 三指针法，需注意指针初始化为null
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode pre = null;
        ListNode cur = head;
        ListNode next = null;
        while (cur != null){
            next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
}
```

剑指Offer 25（合并有序链表）

输入两个递增排序的链表，合并这两个链表并使新链表中的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`

```
// 链表类问题，设置dummyHead是一个常规操作，主要是为了避免讨论头节点，倒不一定是头节点丢失。
// 这个题如果你不用dummyHead，你就需要去讨论头节点到底是l1还是l2。
// 而如果是删除倒数第n个节点那个题，如果不采用dummyHead，你就需要单独去讨论如果删除的是倒数第n个节点，也就是head被删除的情况。
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummyHead = new ListNode(-1), pre = dummyHead;
        while (l1 != null && l2 != null){
            if (l1.val <= l2.val) {
                pre.next = l1;
                pre = l1;
                l1 = l1.next;
            }else {
                pre.next = l2;
                pre = l2;
                l2 = l2.next;
            }
        }
        if (l1 != null) {
            pre.next = l1;
        }
        if (l2 != null) {
            pre.next = l2;
        }
        return dummyHead.next;
    }
}
```

剑指Offer 26（树的子结构）

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`

```java
// 自己写不出啊
class Solution {
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        if(A == null || B == null) return false;
        return dfs(A, B) || isSubStructure(A.left, B) ||
isSubStructure(A.right, B);
    }
    public boolean dfs(TreeNode A, TreeNode B){
        if(B == null) return true;
        if(A == null) return false;
        return A.val == B.val && dfs(A.left, B.left) && dfs(A.right, B.right);
    }
}
```

剑指Offer 27（树的镜像）

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：

镜像输出：

示例 1：

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

限制：

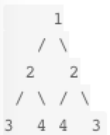
`0 <= 节点个数 <= 1000`

```java
// 递归，注意一下其他的方法，用栈或队列
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if (root == null) return null;
        TreeNode tempLeft = root.left;//后面的操作会改变 left 指针，因此先保存下来
        root.left = mirrorTree(root.right);
        root.right = mirrorTree(tempLeft);
        return root;
    }
}
```

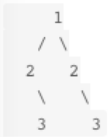
剑指Offer 28（树的对称）

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。



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



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
```

> 做递归思考三步：
>
> 1. 递归的函数要干什么？
>
> - 函数的作用是判断传入的两个树是否镜像。
> - 输入：TreeNode left, TreeNode right
> - 输出：是：true，不是：false
>
> 2. 递归停止的条件是什么？
>
> - 左节点和右节点都为空 -> 倒底了都长得一样 ->true
> - 左节点为空的时候右节点不为空，或反之 -> 长得不一样-> false
> - 左右节点值不相等 -> 长得不一样 -> false
>
> 3. 从某层到下一层的关系是什么？
>
> - 要想两棵树镜像，那么一棵树左边的左边要和二棵树右边的右边镜像，一棵树左边的右边要和二棵树右边的左边镜像
> - 调用递归函数传入左左和右右
> - 调用递归函数传入左右和右左
> - 只有左左和右右镜像且左右和右左镜像的时候，我们才能说这两棵树是镜像的
>
> 4. 调用递归函数，我们想知道它的左右孩子是否镜像，传入的值是root的左孩子和右孩子。这之前记得判个root==null

```java
// 递归
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null) return true;
        return helper(root.left, root.right);
    }

    private boolean helper(TreeNode node1, TreeNode node2){
        if (node1 == null && node2 == null) return true;
        if (node1 == null || node2 == null || node1.val != node2.val) return
false;
        return helper(node1.left, node2.right) && helper(node1.right,
node2.left);
    }
}
```

剑指Offer 29（顺时针打印矩阵）

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`

```java
class Solution {
    // 第一位表示行变化，第二位表示列变化
    // 0代表不变，1代表增加，-1代表减少
    // 四行分别代表：右，下，左，上
    int[][] d = {{0,1}, {1,0}, {0, -1}, {-1, 0}};
    public int[] spiralOrder(int[][] matrix) {
        int n = matrix.length;
        if (n == 0)
            return new int[0];
        int m = matrix[0].length;
        int[] res = new int[n * m];
        boolean[][] vis = new boolean[n][m];
        int r = 0, c = 0;
        int index = 0;
        // 初始化 方向。往左
```

```
        int dir = 0;
        while (index < n * m) {
            res[index++] = matrix[r][c];
            // 标记当前点已经访问过
            vis[r][c] = true;
            int nextR = r + d[dir % 4][0];
            int nextC = c + d[dir % 4][1];
            // 边界判断
            // 最外圈，0 <= nextR < n-1，0 <= nextC <= m-1
            // 其余位置根据vis数据判断，只要访问过就算达到边界
            if (nextR == n || nextR < 0 || nextC == m || nextC < 0 ||
vis[nextR][nextC]) {
                // 达到边界转方向
                dir += 1;
                nextR = r + d[dir % 4][0];
                nextC = c + d[dir % 4][1];
            }
            // 更新当前点
            r = nextR;
            c = nextC;
        }
        return res;
    }
}
```

剑指Offer 30（包含 min 函数的栈）

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:

1. 各函数的调用总次数不超过 20000 次

```
class MinStack {

    // 数据栈和最小值栈
    Stack<Integer> dataStack, minStack;
    public MinStack() {
        dataStack = new Stack<>();
```

```
            minStack = new Stack<>();
        }

        public void push(int x) {
            // 数据栈添加数据
            dataStack.add(x);
            // 若最小值栈为空 或者
            // 加入的值小于等于当前最小值，添加最小值
            if(minStack.empty() || minStack.peek() >= x)
                minStack.add(x);
        }

        public void pop() {
            // 数据栈pop，若数据栈pop的值是最小值则最小值栈也要pop
            if(dataStack.pop().equals(minStack.peek()))
                minStack.pop();
        }

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

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

剑指Offer 31（栈的压入、弹出序列）

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：

1. `0 <= pushed.length == popped.length <= 1000`
2. `0 <= pushed[i], popped[i] < 1000`
3. `pushed` 是 `popped` 的排列。

```java
// 根据两个数组模拟栈的出入
class Solution {
    public boolean validateStackSequences(int[] pushed, int[] popped) {
        Stack<Integer> stack = new Stack<>();
        int i = 0;
        for(int num : pushed) {
            stack.push(num); // num 入栈
            // 只有当此时栈顶元素等于此时poped[i]，出栈，并且i++
            while(!stack.isEmpty() && stack.peek() == popped[i]) { // 循环判断与出栈
                stack.pop();
                i++;
            }
        }
        // 模拟成功则栈为空
        return stack.isEmpty();
    }
}
```

剑指Offer 32-1（从上往下打印二叉树）

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:

1. 节点总数 <= 1000

```java
// 就一个层次遍历，不是很懂为什么是medium
class Solution {
    public int[] levelOrder(TreeNode root) {
        if (root == null)
            return new int[]{};
        List<Integer> ansList = new ArrayList<>();
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()){
            TreeNode node = queue.poll();
```

```
            if (node.left != null)
                queue.add(node.left);
            if (node.right != null)
                queue.add(node.right);
            ansList.add(node.val);
        }
        int[] ans = new int[ansList.size()];
        for (int i = 0; i < ansList.size(); i++) {
            ans[i] = ansList.get(i);
        }
        return ans;
    }
}
```

剑指Offer 32-2（把二叉树打印成多行）

剑指 Offer 32 - II. 从上到下打印二叉树 II

难度 简单 59 收藏 分享 切换为英文 接收动态 反馈

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：

1. 节点总数 `<= 1000`

```
// 自己的方法用了两个队列
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> ansList = new ArrayList<>();
        if (root == null)
            return ansList;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()){
            List<Integer> list = new ArrayList<>();
            Queue<TreeNode> tempQueue = new LinkedList<>();
```

```
            while (!queue.isEmpty()){
                tempQueue.add(queue.poll());
            }
            while (!tempQueue.isEmpty()){
                if (tempQueue.peek().left != null)
                    queue.add(tempQueue.peek().left);
                if (tempQueue.peek().right != null)
                    queue.add(tempQueue.peek().right);
                list.add(tempQueue.poll().val);
            }
            ansList.add(new ArrayList<>(list));
        }
        return ansList;
    }
}
```

剑指Offer 32-3（按之字形顺序打印二叉树）

剑指 Offer 32 - III. 从上到下打印二叉树 III

难度 中等 50 收藏 分享 切换为英文 接收动态 反馈

请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。

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

提示：

1. 节点总数 <= 1000

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> ansList = new ArrayList<>();
        if (root == null)
            return ansList;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()){
```

```
            List<Integer> list = new ArrayList<>();
            Queue<TreeNode> tempQueue = new LinkedList<>();
            while (!queue.isEmpty()){
                tempQueue.add(queue.poll());
            }
            while (!tempQueue.isEmpty()){
                if (tempQueue.peek().left != null)
                    queue.add(tempQueue.peek().left);
                if (tempQueue.peek().right != null)
                    queue.add(tempQueue.peek().right);
                list.add(tempQueue.poll().val);
            }
            ansList.add(new ArrayList<>(list));
        }
        // 与上一题的主要区别
        for (int i = 1; i < ansList.size(); i += 2) {
            Collections.reverse(ansList.get(i));
        }
        return ansList;
    }
}
```

剑指Offer 33（二叉搜索树的后序遍历序列）

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：

1. 数组长度 `<= 1000`

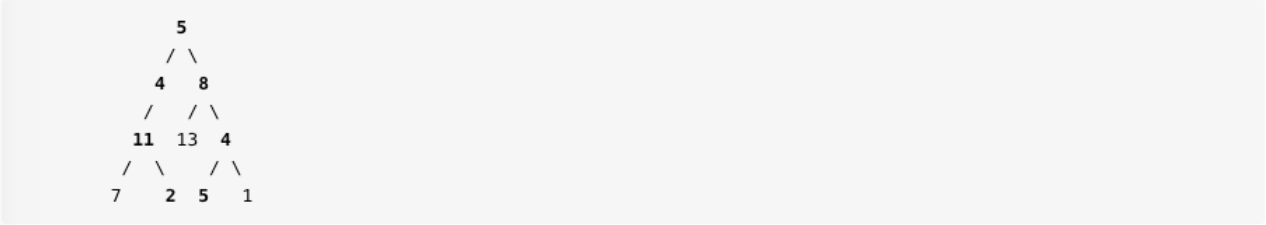
```
// 单调栈
class Solution {
    public boolean verifyPostorder(int[] postorder) {
```

```
        // 单调栈使用，单调递增的单调栈
        Deque<Integer> stack = new LinkedList<>();
        int pervElem = Integer.MAX_VALUE;
        // 逆向遍历，就是翻转的先序遍历
        for (int i = postorder.length - 1;i>=0;i--){
            // 左子树元素必须要小于递增栈被peek访问的元素，否则就不是二叉搜索树
            if (postorder[i] > pervElem){
                return false;
            }
            while (!stack.isEmpty() && postorder[i] < stack.peek()){
                // 数组元素小于单调栈的元素了，表示往左子树走了，记录下上个根节点
                // 找到这个左子树对应的根节点，之前右子树全部弹出，不再记录，因为不可能在往根节点的右子树走了
                pervElem = stack.pop();
            }
            // 这个新元素入栈
            stack.push(postorder[i]);
        }
        return true;
    }
}
// 递归
class Solution {
    public boolean verifyPostorder(int[] postorder) {
        return recur(postorder, 0, postorder.length - 1);
    }
    boolean recur(int[] postorder, int i, int j) {
        if(i >= j) return true;
        int p = i;
        while(postorder[p] < postorder[j]) p++;
        int m = p;
        while(postorder[p] > postorder[j]) p++;
        return p == j && recur(postorder, i, m - 1) && recur(postorder, m, j - 1);
    }
}
```

剑指Offer 34（树的路径和）

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`，



返回:

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

提示:

1. 节点总数 `<= 10000`

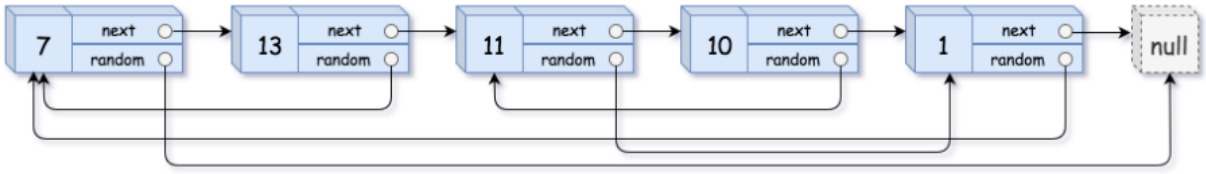
```
class Solution {
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        List<List<Integer>> ans = new ArrayList<>();
        dfs(root, sum, 0, new ArrayList<>(), ans);
        return ans;
    }

    private void dfs(TreeNode root, int sum, int curSum, List<Integer> list,
List<List<Integer>> ans){
        // 节点为空直接返回
        if (root == null)
            return;
        // 将当前节点的值加入到list中
        list.add(root.val);
        // 每往下走一步就要计算走过的路径和
        curSum += root.val;
        // 如果到达叶子节点，就不能往下走了，直接return
        if (root.left == null && root.right == null){
            if (sum == curSum)
                // 此处一定要new
                ans.add(new ArrayList<>(list));
            // 需要将最后加入的节点给移除掉，
            // 因为下一步直接return了，不会再走最后一行的remove了，
            // 所以这里在return之前提前把最后一个结点的值给remove掉。
            list.remove(list.size() - 1);
            return;
        }
        // 如果没到达叶子节点，就继续从他的左右两个子节点往下找
        dfs(root.left, sum, curSum, list, ans);
        dfs(root.right, sum, curSum, list, ans);
```

```
            // 我们要理解递归的本质，当递归往下传递的时候他最后还是会往回走，
            // 我们把这个值使用完之后还要把它给移除，这就是回溯
            list.remove(list.size() - 1);
        }
    }
```

剑指Offer 35（链表拷贝）

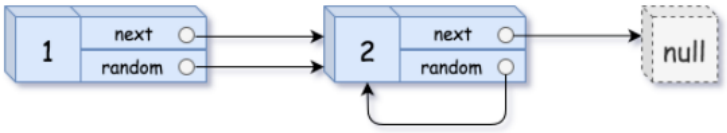
请实现 `copyRandomList` 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 `next` 指针指向下一个节点，还有一个 `random` 指针指向链表中的任意节点或者 `null`。

示例 1：



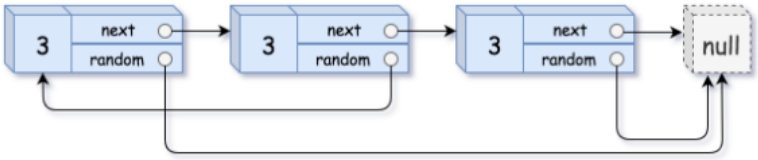

```
输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
```

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 。
>
> 太菜了，没明白题意。

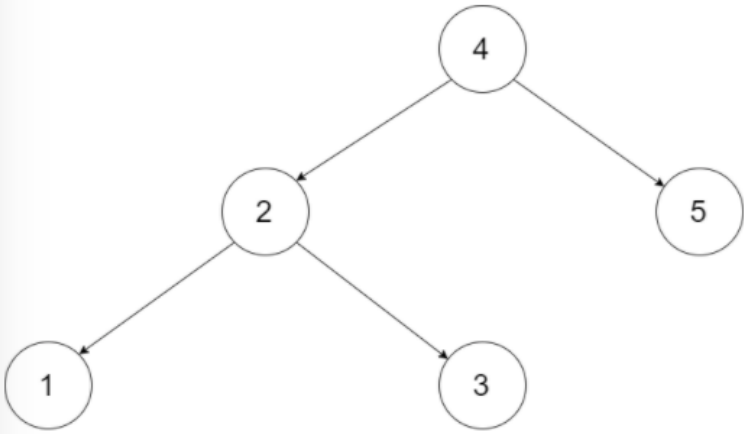
> 浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存。但深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内
>
> 存，修改新对象不会改到原对象。

```
class Solution {
    public Node copyRandomList(Node head) {
        if (head == null) {
            return head;
        }
        //map中存的是(原节点，拷贝节点)的一个映射
        Map<Node, Node> map = new HashMap<>();
        for (Node cur = head; cur != null; cur = cur.next) {
            map.put(cur, new Node(cur.val));
        }
        //将拷贝的新的节点组织成一个链表
        for (Node cur = head; cur != null; cur = cur.next) {
            map.get(cur).next = map.get(cur.next);
            map.get(cur).random = map.get(cur.random);
        }

        return map.get(head);
    }
}
```

剑指Offer 36（二叉搜索树与双向链表）

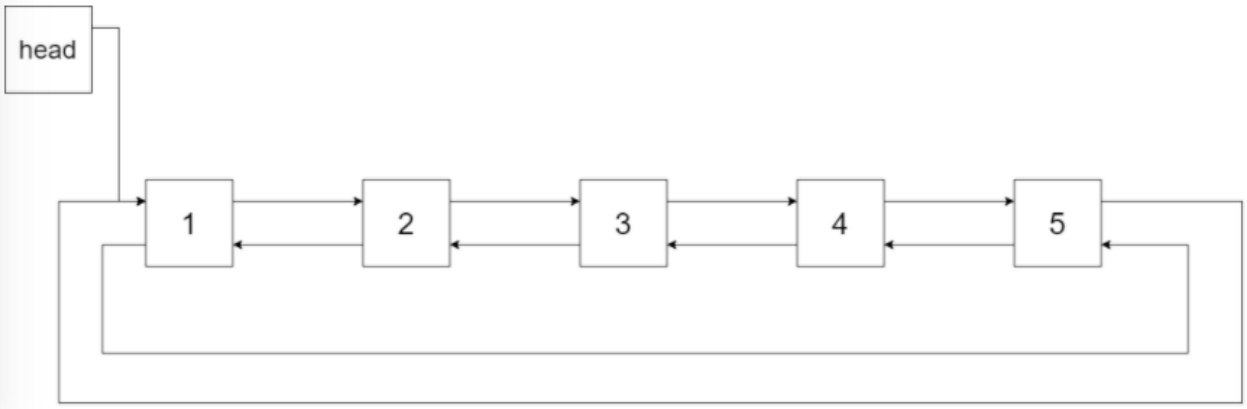
输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。

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。

```
/*
// 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 {
    // 1. 中序，递归
    Node pre;
    Node head;
    public Node treeToDoublyList(Node root) {
        // 边界值
        if(root == null) return null;
        inOrder(root);

        // 题目要求头尾连接
        head.left = pre;
        pre.right = head;
        // 返回头节点
        return head;
    }
    void inOrder(Node cur) {
        // 递归结束条件
        if(cur == null)
            return;
        inOrder(cur.left);
        // 如果pre为空，就说明是第一个节点，头结点，然后用head保存头结点，用于之后的返回
        if (pre == null)
            head = cur;
        // 如果不为空，那就说明是中间的节点。并且pre保存的是上一个节点，
        // 让上一个节点的右指针指向当前节点
        else if (pre != null)
            pre.right = cur;
        // 再让当前节点的左指针指向父节点，也就连成了双向链表
        cur.left = pre;
        // 保存当前节点，用于下层递归创建
        pre = cur;
        inOrder(cur.right);
    }
}/*
// 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) {
```

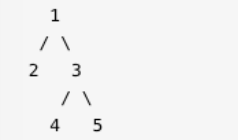
```
        val = _val;
        left = _left;
        right = _right;
    }
};
*/
class Solution {
    // 1. 中序，递归
    Node pre;
    Node head;
    public Node treeToDoublyList(Node root) {
        // 边界值
        if(root == null) return null;
        inOrder(root);

        // 题目要求头尾连接
        head.left = pre;
        pre.right = head;
        // 返回头节点
        return head;
    }
    void inOrder(Node cur) {
        // 递归结束条件
        if(cur == null)
            return;
        inOrder(cur.left);
        // 如果pre为空，就说明是第一个节点，头结点，然后用head保存头结点，用于之后的返回
        if (pre == null)
            head = cur;
        // 如果不为空，那就说明是中间的节点。并且pre保存的是上一个节点，
        // 让上一个节点的右指针指向当前节点
        else if (pre != null)
            pre.right = cur;
        // 再让当前节点的左指针指向父节点，也就连成了双向链表
        cur.left = pre;
        // 保存当前节点，用于下层递归创建
        pre = cur;
        inOrder(cur.right);
    }
}
```

剑指Offer 37（序列化二叉树）

请实现两个函数，分别用来序列化和反序列化二叉树。

示例:

你可以将以下二叉树：



序列化为 "[1,2,3,null,null,4,5]"

```
// 暂时放弃，但感觉可以写
public class Codec {
    public String serialize(TreeNode root) {
        if(root == null) return "[]";
        StringBuilder res = new StringBuilder("[");
        Queue<TreeNode> queue = new LinkedList<TreeNode>() {{ 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<TreeNode>() {{ 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;
    }
}
```

剑指Offer 38（不重复全排列）

输入一个字符串，打印出该字符串中字符的所有排列。

你可以以任意顺序返回这个字符串数组，但里面不能有重复元素。

示例:

```
输入：s = "abc"
输出：["abc","acb","bac","bca","cab","cba"]
```

限制:

`1 <= s 的长度 <= 8`

```
class Solution {
    List<String> res = new ArrayList<>();
    char[] c;
    public String[] permutation(String s) {
        c = s.toCharArray();
        dfs(0);
        return res.toArray(new String[res.size()]);
    }
    void dfs(int x) {
        if(x == c.length - 1) {
            res.add(String.valueOf(c)); // 添加排列方案
            return;
        }
        HashSet<Character> set = new HashSet<>();
        for(int i = x; i < c.length; i++) {
            if(set.contains(c[i])) continue; // 重复，因此剪枝
            set.add(c[i]);
            swap(i, x); // 交换，将 c[i] 固定在第 x 位
            dfs(x + 1); // 开启固定第 x + 1 位字符
            swap(i, x); // 恢复交换
        }
    }
    void swap(int a, int b) {
        char tmp = c[a];
        c[a] = c[b];
        c[b] = tmp;
    }
}
```

剑指Offer 39（数组中出现次数超过一半的数字）

数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1：

```
输入：[1, 2, 3, 2, 2, 2, 5, 4, 2]
输出：2
```

限制：

`1 <= 数组长度 <= 50000`

```java
class Solution {
    public int majorityElement(int[] nums) {
        // 需要的数字出现次数多于一半 那么排序后必定在中间
        Arrays.sort(nums);
        return nums[nums.length / 2];
    }
}
```

剑指Offer 40（最小的 K 个数）

输入整数数组 `arr` ，找出其中最小的 `k` 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

示例 1：

```
输入：arr = [3,2,1], k = 2
输出：[1,2] 或者 [2,1]
```

示例 2：

```
输入：arr = [0,1,2,1], k = 1
输出：[0]
```

限制：

- `0 <= k <= arr.length <= 10000`
- `0 <= arr[i] <= 10000`

```java
// Arrays.sort()
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        Arrays.sort(arr);
        int[] ans = new int[k];
        for (int i = 0; i < k; i++) {
            ans[i] = arr[i];
        }
        return ans;
    }
}
```

剑指Offer 41（数据流中的中位数）

如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 (2 + 3) / 2 = 2.5

设计一个支持以下两种操作的数据结构：

- void addNum(int num) - 从数据流中添加一个整数到数据结构中。
- double findMedian() - 返回目前所有元素的中位数。

示例 1：

```
输入：
["MedianFinder","addNum","addNum","findMedian","addNum","findMedian"]
[[],[1],[2],[],[3],[]]
输出：[null,null,null,1.50000,null,2.00000]
```

示例 2：

```
输入：
["MedianFinder","addNum","findMedian","addNum","findMedian"]
[[],[2],[],[3],[]]
输出：[null,null,2.00000,null,2.50000]
```

限制：

- 最多会对 `addNum、findMedian` 进行 `50000` 次调用。

```java
class MedianFinder {
    Queue<Integer> A, B;
    public MedianFinder() {
        A = new PriorityQueue<>(); // 小顶堆，保存较大的一半
        B = new PriorityQueue<>((x, y) -> (y - x)); // 大顶堆，保存较小的一半
    }
    public void addNum(int num) {
        if(A.size() != B.size()) {
            A.add(num);
            B.add(A.poll());
        } else {
```

```
            B.add(num);
            A.add(B.poll());
        }
    }
    public double findMedian() {
        return A.size() != B.size() ? A.peek() : (A.peek() + B.peek()) / 2.0;
    }
}
```

剑指Offer 42（连续子数组的最大和）

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`

```
class Solution {
    public int maxSubArray(int[] nums) {
        // 动态规划
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        int max = dp[0];
        for(int i = 1; i < nums.length; i++){
            dp[i] = Math.max(dp[i-1] + nums[i], nums[i]);
            max = Math.max(max, dp[i]);
        }
        return max;
    }
}
```

剑指Offer 43（1 ～n 整数中 1 出现的次数）

给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。

示例:

```
输入: 13
输出: 6
解释: 数字 1 出现在以下数字中: 1, 10, 11, 12, 13 。
```

```java
// 数学方法
class Solution {
    public int countDigitOne(int n) {
        int num = n;
        long i = 1;
        int s = 0;
        while(num > 0) {
            if(num % 10 == 0) // 不包含1 -9
                s += (num / 10) * i; // 修正值是 0
            if(num % 10 == 1) // 包含 1 - 9的一部分
                s += (num / 10) * i + (n % i) + 1; // 修正值是(n % i) + 1
            if(num % 10 > 1) // 包含1 - 9
                s += (num / 10) * i + i; // i
            num = num / 10; // 比如109/10 = 10,  可以按10位的处理，因为i增量了10倍
            i = i * 10; // 每次1的个数是增加10倍
        }
        return s;
    }
}
// 或者用数位DP，得去了解
```

剑指Offer 44（数字序列中的某一位数字）

数字以0123456789101112131415...的格式序列化到一个字符序列中。在这个序列中，第5位（从下标0开始计数）是5，第13位是1，第19位是4，等等。

请写一个函数，求任意第n位对应的数字。

示例 1：

```
输入：n = 3
输出：3
```

示例 2：

```
输入：n = 11
输出：0
```

限制：

- `0 <= n < 2^31`

```java
// 找规律
class Solution {
    public int findNthDigit(int n) {
        int digit = 1;
        long start = 1;
        long count = 9;
        while (n > count) { // 1.
            n -= count;
            digit += 1;
```

```
            start *= 10;
            count = digit * start * 9;
        }
        long num = start + (n - 1) / digit; // 2.
        return Long.toString(num).charAt((n - 1) % digit) - '0'; // 3.
    }
}
```

剑指Offer 45（把数组排成最小的数）

输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

示例 1:

```
输入: [10,2]
输出: "102"
```

示例 2:

```
输入: [3,30,34,5,9]
输出: "3033459"
```

提示:

- `0 < nums.length <= 100`

说明:

- 输出结果可能非常大，所以你需要返回一个字符串而不是整数
- 拼接起来的数字可能会有前导 0，最后结果不需要去掉前导 0

```
class Solution {
    public String minNumber(int[] nums) {
        String[] strs = new String[nums.length];
        // 整型数组 -> 字符串数组
        for(int i = 0; i < nums.length; i++)
            strs[i] = String.valueOf(nums[i]);
        // 将lambda表达式定义的比较器传入 Array.sort() 方法
        Arrays.sort(strs, (x, y) -> (x + y).compareTo(y + x));
        // 将字符串数组连接为数组，速度快于 String.join("",strs);
        StringBuilder res = new StringBuilder();
        for(String s : strs)
            res.append(s);
        return res.toString();
    }
}
```

剑指Offer 46（把数字翻译成字符串）

给定一个数字，我们按照如下规则把它翻译为字符串：0 翻译成 “a”，1 翻译成 “b”，……，11 翻译成 “l”，……，25 翻译成 “z”。一个数字可能有多个翻译。请编程实现一个函数，用来计算一个数字有多少种不同的翻译方法。

示例 1：

```
输入：12258
输出：5
解释：12258有5种不同的翻译，分别是"bccfi", "bwfi", "bczi", "mcfi"和"mzi"
```

提示：

- $0 <= num < 2^{31}$

```
// 熟记compareTo函数的使用
class Solution {
    public int translateNum(int num) {
        String s = String.valueOf(num);
        int a = 1, b = 1;
        for(int i = s.length() - 2; i > -1; i--) {
            String tmp = s.substring(i, i + 2);
            int c = tmp.compareTo("10") >= 0 && tmp.compareTo("25") <= 0 ? a +
b : a;
            b = a;
            a = c;
        }
        return a;
    }
}
```

剑指Offer 47（礼物的最大价值）

在一个 m*n 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于 0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向右或者向下移动一格、直到到达棋盘的右下角。给定一个棋盘及其上面的礼物的价值，请计算你最多能拿到多少价值的礼物？

示例 1：

```
输入：
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
输出：12
解释：路径 1→3→5→2→1 可以拿到最多价值的礼物
```

提示：

- $0 < grid.length <= 200$
- $0 < grid[0].length <= 200$

```
// 没用滚动数组
```

```java
class Solution {
    public int maxValue(int[][] grid) {
        if (grid == null || grid.length == 0)
            return 0;
        int m = grid.length, n = grid[0].length;
        int[][] dp = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                dp[i][j] = Math.max(grid[i - 1][j - 1] + dp[i - 1][j], grid[i -
1][j - 1] + dp[i][j - 1]);
            }
        }
        return dp[m][n];
    }
}

// 用了滚动数组优化空间复杂度
class Solution {
    public int maxValue(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        int[] dp = new int[n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                dp[j] = Math.max(dp[j], dp[j - 1]) + grid[i - 1][j - 1];
            }
        }
        return dp[n];
    }
}

// DFS超时
class Solution {
    int m = 0;  // 行数
    int n = 0;  // 列数
    public int maxValue(int[][] grid) {
        m = grid.length;
        if (m == 0) {   // 空的grid
            return 0;
        }
        n = grid[0].length;
        return DFS(grid, 0, 0);
    }

    public int DFS(int[][] grid, int row, int column) {
        int goRight = grid[row][column];
        int goDown = grid[row][column];
        if (row + 1 < m && column + 1 < n) {            // 右和下都可以走
            goRight += DFS(grid, row, column + 1);
            goDown += DFS(grid, row + 1, column);
```

```
        } else if (row + 1 < m && column + 1 >= n) {    // 只能走右
            goRight += DFS(grid, row + 1, column);
        } else if (row + 1 >= m && column + 1 < n) {    // 只能走下
            goDown += DFS(grid, row, column + 1);
        }
        return Math.max(goRight, goDown);
    }
}

// 暴力递归超时
class Solution {
    public int maxValue(int[][] grid) {
        if (grid == null || grid.length == 0)
            return 0;
        int m = grid.length, n = grid[0].length;
        return helper(grid, m - 1, n - 1);
    }

    private int helper(int[][] grid, int m, int n){
        if (m == 0 && n == 0)
            return grid[m][n];
        if (n == 0)
            return grid[m][n] + helper(grid, m - 1, 0);
        if (m == 0)
            return grid[m][n] + helper(grid, 0, n - 1);
        return grid[m][n] + Math.max(helper(grid, m - 1, n), helper(grid, m, n
- 1));
    }
}
```

剑指Offer 48（最长不含重复字符的子字符串）

请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。

示例 1:

```
输入: "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
```

示例 2:

```
输入: "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
```

示例 3:

```
输入: "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
     请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
```

提示：

- `s.length <= 40000`

```java
// 哈希表解法，滑动窗口
class Solution {
    public int lengthOfLongestSubstring(String s) {
        if (s.length() == 0) return 0;
        Map<Character, Integer> map = new HashMap<>();
        int ans = 0;
        for (int end = 0, start = 0; end < s.length(); end++) {
            // 如果当前出现重复字母，窗口左指针右移到重复字母上次出现的右一位
            if (map.containsKey(s.charAt(end))) {
                start = Math.max(map.get(s.charAt(end))+1, start);
            }
            // 不断更新的窗口长度与原来所求结果的最大值
            ans = Math.max(ans, end - start + 1);
            // 将出现字母与最新出现的位置存入HashMap
            map.put(s.charAt(end), end);
        }
        return ans;
    }
}
```

剑指Offer 49（丑数）

我们把只包含质因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 n 个丑数。

示例:

```
输入: n = 10
输出: 12
解释: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 是前 10 个丑数。
```

说明:

1. `1` 是丑数。
2. `n` 不超过1690。

```
class Solution {
    public int nthUglyNumber(int n) {
        int p2 = 0,p3 = 0,p5 = 0;
        int[] dp = new int[n];
        dp[0] = 1;
        for (int i = 1; i < n; i++) {
            dp[i] = Math.min(dp[p2] * 2, Math.min(dp[p3] * 3, dp[p5] * 5));
            if(dp[i] == dp[p2] * 2) p2++;
            if(dp[i] == dp[p3] * 3) p3++;
            if(dp[i] == dp[p5] * 5) p5++;
        }
        return dp[n - 1];
    }
    // 一个十分巧妙的动态规划问题
    // 1.我们将前面求得的丑数记录下来，后面的丑数就是前面的丑数*2，*3，*5
    // 2.但是问题来了，我怎么确定已知前面k-1个丑数，我怎么确定第k个丑数呢
    // 3.采取用三个指针的方法，p2,p3,p5
    // 4.index2指向的数字下一次永远*2，p3指向的数字下一次永远*3，p5指向的数字永远*5
    // 5.我们从2*p2 3*p3 5*p5选取最小的一个数字，作为第k个丑数
    // 6.如果第K个丑数==2*p2，也就是说前面0-p2个丑数*2不可能产生比第K个丑数更大的丑数了，
所以p2++
    // 7.p3,p5同理
    // 8.返回第n个丑数
}
```

剑指Offer 50（第一个只出现一次的字符位置）

在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。 s 只包含小写字母。

示例:

```
s = "abaccdeff"
返回 "b"

s = ""
返回 " "
```

限制:

`0 <= s 的长度 <= 50000`

```java
class Solution {
    public char firstUniqChar(String s) {
        Map<Character, Integer> map = new HashMap<>();
        for (Character c : s.toCharArray()) {
            //getOrDefault意思就是当Map集合中有这个key时，就使用这个key值，如果没有就使用默认值defaultValue
            //此处存入的是每个元素出现的次数
            map.put(c, map.getOrDefault(c, 0) + 1);
        }
        // 按照字符出现顺序，查找出现近一次的字符。用foreach 顺序不对
        for (int i = 0; i < s.length(); i++) {
            if (map.get(s.charAt(i))==1)
                return s.charAt(i);
        }
        return ' ';
    }
}
```

剑指Offer 51（数组中的逆序对）

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

示例 1:

```
输入: [7,5,6,4]
输出: 5
```

限制:

`0 <= 数组长度 <= 50000`

```java
// 暴力超时
class Solution {
    public int reversePairs(int[] nums) {
        int cnt = 0;
        int len = nums.length;
        for (int i = 0; i < len - 1; i++) {
            for (int j = i + 1; j < len; j++) {
                if (nums[i] > nums[j]) {
                    cnt++;
                }
            }
        }
        return cnt;
    }
}

// 归并排序
```

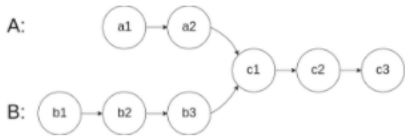
```java
class Solution {

    public int reversePairs(int[] nums) {
        int len = nums.length;

        if (len < 2) {
            return 0;
        }

        int[] copy = new int[len];
        for (int i = 0; i < len; i++) {
            copy[i] = nums[i];
        }

        int[] temp = new int[len];
        return reversePairs(copy, 0, len - 1, temp);
    }

    /**
     * nums[left..right] 计算逆序对个数并且排序
     *
     * @param nums
     * @param left
     * @param right
     * @param temp
     * @return
     */
    private int reversePairs(int[] nums, int left, int right, int[] temp) {
        if (left == right) {
            return 0;
        }

        int mid = left + (right - left) / 2;
        int leftPairs = reversePairs(nums, left, mid, temp);
        int rightPairs = reversePairs(nums, mid + 1, right, temp);

        // 如果整个数组已经有序，则无需合并，注意这里使用小于等于
        if (nums[mid] <= nums[mid + 1]) {
            return leftPairs + rightPairs;
        }

        int crossPairs = mergeAndCount(nums, left, mid, right, temp);
        return leftPairs + rightPairs + crossPairs;
    }
    /**
     * nums[left..mid] 有序，nums[mid + 1..right] 有序
     *
     * @param nums
     * @param left
```

```
     * @param mid
     * @param right
     * @param temp
     * @return
     */
    private int mergeAndCount(int[] nums, int left, int mid, int right, int[]
temp) {
        for (int i = left; i <= right; i++) {
            temp[i] = nums[i];
        }

        int i = left;
        int j = mid + 1;

        int count = 0;
        for (int k = left; k <= right; k++) {
            // 有下标访问，得先判断是否越界
            if (i == mid + 1) {
                nums[k] = temp[j];
                j++;
            } else if (j == right + 1) {
                nums[k] = temp[i];
                i++;
            } else if (temp[i] <= temp[j]) {
                // 注意：这里是 <= ，写成 < 就不对，请思考原因
                nums[k] = temp[i];
                i++;
            } else {
                nums[k] = temp[j];
                j++;

                // 在 j 指向的元素归并回去的时候，计算逆序对的个数，只多了这一行代码
                count += (mid - i + 1);
            }
        }
        return count;
    }
}
```

剑指Offer 52（相交链表）

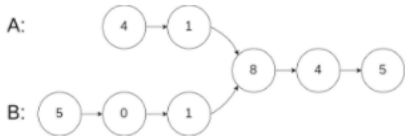
输入两个链表，找出它们的第一个公共节点。

如下面的两个链表：



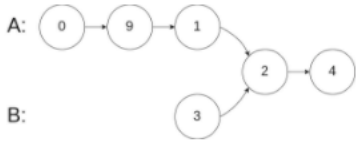
在节点 c1 开始相交。

示例 1：



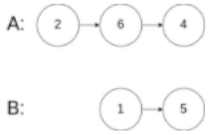

```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
```

示例 2：




```
输入：intersectVal = 2, listA = [0,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
输出：Reference of the node with value = 2
输入解释：相交节点的值为 2 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [0,9,1,2,4]，链表 B 为 [3,2,4]。在 A 中，相交节点前有 3 个节点；在 B 中，相交节点前有 1 个节点。
```

示例 3：




```
输入：intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
输出：null
输入解释：从各自的表头开始算起，链表 A 为 [2,6,4]，链表 B 为 [1,5]。由于这两个链表不相交，所以 intersectVal 必须为 0，而 skipA 和 skipB 可以是任意值。
解释：这两个链表不相交，因此返回 null。
```

```java
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) {
            return null;
        }
        ListNode pA = headA, pB = headB;
        //判断两个指针所指节点的值是否相同，若不相同则执行循环语句
        //总体思路是指针各自遍历一遍链表，遍历完成后，然后遍历另一条链表，当指针指向同一元素时，表明此处为相交节点
        //若无相交节点，则遍历完两条链表，pA与pB都为null，此时 pA == pB，跳出循环
        while(pA != pB) {
            //若pA为空，则pA指向headB，否则指向下一点
            pA = pA == null ? headB : pA.next;
            //若pB为空，则pB指向headA，否则指向下一点
            pB = pB == null ? headA : pB.next;
        }
        return pA;
    }
}
```

剑指Offer 53 - 1（有序数组同一元素的个数）

统计一个数字在排序数组中出现的次数。

示例 1:

```
输入: nums = [5,7,7,8,8,10], target = 8
输出: 2
```

示例 2:

```
输入: nums = [5,7,7,8,8,10], target = 6
输出: 0
```

限制:

`0 <= 数组长度 <= 50000`

```java
class Solution {
    public int search(int[] nums, int target) {
        // 非空判断
        if(nums == null || nums.length == 0) {
            return 0;
        }
        int ans = 0;
        // 二分查找
        int low = 0, high = nums.length - 1;
        int mid = (low + high) / 2;
        while(low <= high) {
            if(nums[mid] == target) {
                ans = count(nums, mid, target);
                break;
            }
            if(nums[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
            mid = (low + high) / 2;
        }
        return ans;
    }

    // 向两边扩张计算出个数
    private int count(int[] nums, int cur, int target){
        int l = cur - 1, h = cur + 1;
        int ans = 1;
        while(l >= 0 && nums[l] == target) {
            l--;
```

```
            ans++;
        }
        while(h < nums.length && nums[h] == target) {
            h++;
            ans++;
        }
        return ans;
    }
}
```

剑指Offer 53 - 2（0～n-1中缺失的数字）

一个长度为n-1的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中，请找出这个数字。

示例 1:

```
输入: [0,1,3]
输出: 2
```

示例 2:

```
输入: [0,1,2,3,4,5,6,7,9]
输出: 8
```

限制:

`1 <= 数组长度 <= 10000`

```
class Solution {
    public int missingNumber(int[] nums) {
        // 二分法，最后返回low或者high都可
        int low = 0, high = nums.length - 1;
        while(low <= high) {
            int mid = (low + high) / 2;
            if(nums[mid] == mid)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return low;
    }
}

// 理论的和减去实际的和即为缺少的数字
class Solution {
    public int missingNumber(int[] nums) {
        //计算出0-n的和  n*(n+1)/2
        int sum = nums.length * (nums.length+1)/2;
        return  sum - Arrays.stream(nums).sum() ;
```

```
    }
}
```

剑指Offer 54（二叉搜索树的倒数第k大节点）

给定一棵二叉搜索树，请找出其中第k大的节点。

示例 1:

```
输入: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
输出: 4
```

示例 2:

```
输入: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
输出: 4
```

限制:

1 ≤ k ≤ 二叉搜索树元素个数

```java
class Solution {
    // 先中序遍历，存入list，再取倒数第k项
    public int kthLargest(TreeNode root, int k) {
        List<Integer> treeList = new ArrayList<>();
        helper(root, treeList);
        return treeList.get(treeList.size() - k);
    }

    private void helper(TreeNode root, List<Integer> treeList){
        if (root == null) return;
        if (root.left != null) helper(root.left, treeList);
        treeList.add(root.val);
        if (root.right != null) helper(root.right, treeList);
    }
}

// 做了点小变化，反中序遍历，直接得到逆序列
class Solution {
     public int kthLargest(TreeNode root, int k) {
        List<Integer> treeList = new ArrayList<>();
```

```
        helper(root, treeList);
        return treeList.get(k - 1);
    }

    private void helper(TreeNode root, List<Integer> treeList){
        if (root == null) return;
        if (root.right != null) helper(root.right, treeList);
        treeList.add(root.val);
        if (root.left != null) helper(root.left, treeList);
    }
}

// 思路同上，但直接return答案，不再用list
class Solution {
    private int ans = 0, cnt = 0;
    public int kthLargest(TreeNode root, int k) {
        helper(root, k);
        return ans;
    }

    private void helper(TreeNode root,  int k){
        if (root == null) return;
        if (root.right != null) helper(root.right, k);
        if (++cnt == k){
            ans = root.val;
        }
        if (root.left != null) helper(root.left, k);
    }
}
```

剑指Offer 55-1（树的深度）

给定一棵二叉搜索树，请找出其中第k大的节点。

示例 1：

```
输入：root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
输出：4
```

示例 2：

```
输入：root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
输出：4
```

限制：

1 ≤ k ≤ 二叉搜索树元素个数

```java
// 递归法
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null)
            return 0;
        return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
    }
}
//层次遍历，AC速度还不如递归。但是面试可能考
class Solution {
    public int maxDepth(TreeNode root) {
        if(root == null) return 0;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        int depth = 0;
        while(!queue.isEmpty()){
            // 需要先记录当前层的长度，因为queue.size()会变
            int curSize = queue.size();
            for(int i = 0; i < curSize; i++){
                TreeNode temp = queue.poll();
                if(temp.left != null) queue.add(temp.left);
                if(temp.right != null) queue.add(temp.right);
            }
            depth++;
        }
        return depth;
    }
```

```
}
```

剑指Offer 55-2（树的平衡）

输入一棵二叉树的根节点，判断该树是不是平衡二叉树。如果某二叉树中任意节点的左右子树的深度相差不超过1，那么它就是一棵平衡二叉树。

示例 1:

给定二叉树 `[3,9,20,null,null,15,7]`

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

返回 `true` 。

示例 2:

给定二叉树 `[1,2,2,3,3,null,null,4,4]`

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

返回 `false` 。

限制:

- `1 <= 树的结点个数 <= 10000`

```java
// 结合55-1的经典递归
class Solution {
    public boolean isBalanced(TreeNode root) {
        if(root==null) return true;
        if(Math.abs(maxDepth(root.left) - maxDepth(root.right)) <= 1){
            return isBalanced(root.left) && isBalanced(root.right);
        }
        return false;
    }

    public int maxDepth(TreeNode root) {
        if (root == null)
            return 0;
        return Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
    }
}
```

剑指Offer 56-1（数组中数字出现的次数）

一个整型数组 `nums` 里除两个数字之外，其他数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是O(n)，空间复杂度是O(1)。

示例 1：

```
输入：nums = [4,1,4,6]
输出：[1,6] 或 [6,1]
```

示例 2：

```
输入：nums = [1,2,10,4,1,4,3,3]
输出：[2,10] 或 [10,2]
```

限制：

- `2 <= nums.length <= 10000`

```java
// 哈希表 思路简单，可考虑位运算
class Solution {
    public int[] singleNumbers(int[] nums) {
        int[] ans = new int[2];
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            map.put(nums[i],map.getOrDefault(nums[i],0) + 1);
        }
        int i = 0;
        for (int num : map.keySet()) {
            if (map.get(num) == 1) {
                ans[i] = num;
                i++;
            }
        }
        return ans;
    }
}
```

剑指Offer 56-2（数组中数字出现的次数）

在一个数组 `nums` 中除一个数字只出现一次之外，其他数字都出现了三次。请找出那个只出现一次的数字。

示例 1：

```
输入：nums = [3,4,3,3]
输出：4
```

示例 2：

```
输入：nums = [9,1,7,9,7,9,7]
输出：1
```

限制：

- `1 <= nums.length <= 10000`
- `1 <= nums[i] < 2^31`

```java
// 哈希表 思路简单，可考虑位运算
class Solution {
    public int singleNumber(int[] nums) {
        int ans = 0;
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            map.put(nums[i],map.getOrDefault(nums[i],0) + 1);
        }
        for (int num : map.keySet()) {
            if (map.get(num) == 1) {
                ans = num;
                break;
            }
        }
        return ans;
    }
}
```

剑指Offer 57-1（和为 S 的两个数字）

输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得它们的和正好是s。如果有多对数字的和等于s，则输出任意一对即可。

示例 1：

```
输入：nums = [2,7,11,15], target = 9
输出：[2,7] 或者 [7,2]
```

示例 2：

```
输入：nums = [10,26,30,31,47,60], target = 40
输出：[10,30] 或者 [30,10]
```

限制：

- `1 <= nums.length <= 10^5`
- `1 <= nums[i] <= 10^6`

```java
// 本题是有序的，可优化。可用双指针
class Solution {
    public int[] twoSum(int[] nums, int target) {
        // 用 HashMap 存储数组元素和索引的映射，
        // 在访问到 nums[i] 时，判断 HashMap 中是否存在 target - nums[i]，
        HashMap<Integer, Integer> indexForNum = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            if (indexForNum.containsKey(target - nums[i])) {
                return new int[]{target - nums[i], nums[i]};
            } else {
                indexForNum.put(nums[i], i);
            }
        }
        return null;
    }
}
```

剑指Offer 57-2（和为 S 的连续正数序列）

输入一个正整数 `target` ，输出所有和为 `target` 的连续正整数序列（至少含有两个数）。

序列内的数字由小到大排列，不同序列按照首个数字从小到大排列。

示例 1：

```
输入：target = 9
输出：[[2,3,4],[4,5]]
```

示例 2：

```
输入：target = 15
输出：[[1,2,3,4,5],[4,5,6],[7,8]]
```

限制：

- `1 <= target <= 10^5`

```java
class Solution {
    public int[][] findContinuousSequence(int target) {
        List<int[]> list = new ArrayList<>();

        // 脑子里要有一个区间的概念，这里的区间是(1, 2, 3, ..., target - 1)
        // 套滑动窗口模板，l是窗口左边界，r是窗口右边界，窗口中的值一定是连续值。
        // 当窗口中数字和小于target时，r右移；大于target时，l右移；等于target时就获得了一个解
        for (int l = 1, r = 1, sum = 0; r <= target/2 + 1; r++) {
            sum += r;
            // 左指针右移
            while (sum > target) {
                sum -= l;
```

```
                l++;
            }
            if (sum == target) {
                int[] temp = new int[r - l + 1];
                for (int i = 0; i < temp.length; i++) {
                    temp[i] = l + i;
                }
                list.add(temp);
            }
        }

        int[][] res = new int[list.size()][];
        for (int i = 0; i < res.length; i++) {
            res[i] = list.get(i);
        }
        return res;
    }
}
```

剑指Offer 58-1（翻转单词顺序）

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

示例 1：

```
输入: "the sky is blue"
输出: "blue is sky the"
```

示例 2：

```
输入: "  hello world!  "
输出: "world! hello"
解释: 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。
```

示例 3：

```
输入: "a good   example"
输出: "example good a"
解释: 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。
```

说明：

- 无空格字符构成一个单词。
- 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。
- 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。

```java
class Solution {
    public String reverseWords(String s) {
        String[] strings = s.trim().split(" "); // 删除首尾空格，分割字符串
        StringBuilder res = new StringBuilder();
        for(int i = strings.length - 1; i >= 0; i--) { // 倒序遍历单词列表
            if(strings[i].equals("")) continue; // 遇到空单词则跳过
            res.append(strings[i] + " "); // 将单词拼接至 StringBuilder
        }
        return res.toString().trim(); // 转化为字符串，删除尾部空格，并返回
    }
}
```

剑指Offer 58-2（左旋转字符串）

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。比如，输入字符串"abcdefg"和数字2，该函数将返回左旋转两位得到的结果"cdefgab"。

示例 1：

```
输入: s = "abcdefg", k = 2
输出: "cdefgab"
```

示例 2：

```
输入: s = "lrloseumgh", k = 6
输出: "umghlrlose"
```

限制：

- `1 <= k < s.length <= 10000`

```java
class Solution {
    public String reverseLeftWords(String s, int n) {
        StringBuilder ans = new StringBuilder();
        ans.append(s.substring(n, s.length()));
        ans.append(s.substring(0, n));
        return ans.toString();
    }
}
```

剑指Offer 59-1（滑动窗口的最大值）

给定一个数组 `nums` 和滑动窗口的大小 `k`，请找出所有滑动窗口里的最大值。

示例:

```
输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
输出: [3,3,5,5,6,7]
解释:

  滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

提示:

你可以假设 *k* 总是有效的，在输入数组不为空的情况下，1 ≤ k ≤ 输入数组的大小。

```java
// 自己的太慢了
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums == null || nums.length == 0) {
            return new int[0];
        }
        int len = nums.length;
        int[] ans = new int[len - k + 1];
        for (int l = 0, r = k - 1; l < len - k + 1 ; l++) {
            ans[l] = getMax(nums, l, r);
            r++;
        }
        return ans;
    }

    private int getMax(int[] nums, int l, int r){
        int max = Integer.MIN_VALUE;
        for (int i = l; i <= r; i++) {
            max = Math.max(nums[i], max);
        }
        return max;
    }
}

// LC上需要在线性时间复杂度内解决此题
```

剑指Offer 59-2（队列的最大值）

请定义一个队列并实现函数 `max_value` 得到队列里的最大值，要求函数`max_value`、`push_back` 和 `pop_front` 的**均摊**时间复杂度都是O(1)。

若队列为空，`pop_front` 和 `max_value` 需要返回 -1

示例 1：

```
输入：
["MaxQueue","push_back","push_back","max_value","pop_front","max_value"]
[[],[1],[2],[],[],[]]
输出：[null,null,null,2,1,2]
```

示例 2：

```
输入：
["MaxQueue","pop_front","max_value"]
[[],[],[]]
输出：[null,-1,-1]
```

限制：

- `1 <= push_back,pop_front,max_value的总操作数 <= 10000`
- `1 <= value <= 10^5`

```java
// 定义数组解决
// 定义开始和末尾的指针
// max_value无法做到O(1)
class MaxQueue {

    int[] q = new int[20000];
    int begin = 0, end = 0;

    public MaxQueue() {

    }

    public int max_value() {
        int ans = -1;
        for (int i = begin; i != end; ++i) {
            ans = Math.max(ans, q[i]);
        }
        return ans;
    }

    public void push_back(int value) {
        q[end++] = value;
    }

    public int pop_front() {
        if (begin == end) {
            return -1;
        }
        return q[begin++];
    }
```

```
}
```

剑指Offer 60（n 个骰子的点数）

把n个骰子扔在地上，所有骰子朝上一面的点数之和为s。输入n，打印出s的所有可能的值出现的概率。

你需要用一个浮点数数组返回答案，其中第 i 个元素代表这 n 个骰子所能掷出的点数集合中第 i 小的那个的概率。

示例 1:

```
输入: 1
输出: [0.16667,0.16667,0.16667,0.16667,0.16667,0.16667]
```

示例 2:

```
输入: 2
输出: [0.02778,0.05556,0.08333,0.11111,0.13889,0.16667,0.13889,0.11111,0.08333,0.05556,0.02778]
```

限制:

`1 <= n <= 11`

```java
// 搞不懂
class Solution {
    public double[] twoSum(int n) {
        int[][] dp = new int[n + 1][6 * n + 1];
        //边界条件
        for (int s = 1; s <= 6; s++)
            dp[1][s] = 1;
        for (int i = 2; i <= n; i++) {
            for (int s = i;s <= 6*i;s++) {
                //求dp[i][s]
                for (int d=1; d <= 6; d++) {
                    if(s - d < i - 1)
                      break;//为0了
                    dp[i][s] += dp[i - 1][s - d];
                }
            }
        }
        double total = Math.pow((double)6, (double)n);
        double[] ans = new double[5 * n + 1];
        for (int i = n; i <= 6 * n; i++) {
            ans[i - n] = ((double)dp[n][i]) / total;
        }
        return ans;
    }
}
```

剑指Offer 61（扑克牌顺子）

从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2～10为数字本身，A为1，J为11，Q为12，K为13，而大、小王为 0 ，可以看成任意数字。A 不能视为 14。

示例 1:

```
输入: [1,2,3,4,5]
输出: True
```

示例 2:

```
输入: [0,0,1,2,5]
输出: True
```

限制:

数组长度为 5

数组的数取值为 [0, 13] .

```
class Solution {
    public boolean isStraight(int[] nums) {
        int joker = 0;
        Arrays.sort(nums); // 数组排序
        for(int i = 0; i < 4; i++) {
            if(nums[i] == 0) joker++; // 统计大小王数量
            else if(nums[i] == nums[i + 1]) return false; // 若有重复，提前返回
false
        }
        return nums[4] - nums[joker] < 5; // 最大牌 - 最小牌 < 5 则可构成顺子
    }
}
```

剑指Offer 62（圆圈中最后剩下的数）

0,1,,n-1这n个数字排成一个圆圈，从数字0开始，每次从这个圆圈里删除第m个数字。求出这个圆圈里剩下的最后一个数字。

例如，0、1、2、3、4这5个数字组成一个圆圈，从数字0开始每次删除第3个数字，则删除的前4个数字依次是2、0、4、1，因此最后剩下的数字是3。

示例 1:

```
输入: n = 5, m = 3
输出: 3
```

示例 2:

```
输入: n = 10, m = 17
输出: 2
```

限制:

- `1 <= n <= 10^5`
- `1 <= m <= 10^6`

```java
// 模拟做法，但是很慢
class Solution {
    public int lastRemaining(int n, int m) {
        ArrayList<Integer> list = new ArrayList<>(n);
        for (int i = 0; i < n; i++) {
            list.add(i);
        }
        int idx = 0;
        while (n > 1) {
            idx = (idx + m - 1) % n;
            list.remove(idx);
            // 每取一个数字，剩余的数都会减一，需要的模也减一
            n--;
        }
        return list.get(0);
    }
}
// 数学方法
class Solution {
    public int lastRemaining(int n, int m) {
        int ans = 0;
        // 最后一轮剩下2个人，所以从2开始反推
        for (int i = 2; i <= n; i++) {
            ans = (ans + m) % i;
        }
        return ans;
    }
}
```

剑指Offer 63（股票的最大利润）

假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票一次可能获得的最大利润是多少？

示例 1:

```
输入: [7,1,5,3,6,4]
输出: 5
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
     注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格。
```

示例 2:

```
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。
```

限制：

`0 <= 数组长度 <= 10^5`

```java
class Solution {
    public int maxProfit(int[] prices) {
        int ans = 0;
        //寻找最小的值，并在它之后找一个最大的值，取差值即为答案
        int minPrice = Integer.MAX_VALUE;
        for (int i = 0; i < prices.length; i++) {
            if (prices[i] < minPrice)
                minPrice = prices[i];
            else
                ans = Math.max(ans, prices[i] - minPrice);
        }
        return ans;
    }
}
```

剑指Offer 64（求 1+2+3+...+n）

求 `1+2+...+n` ，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。

示例 1：

```
输入: n = 3
输出: 6
```

示例 2：

```
输入: n = 9
输出: 45
```

限制：

- `1 <= n <= 10000`

```java
class Solution {
    //Java 中，为构成语句，需加一个辅助布尔量 xx ，否则会报错；
    //Java 中，开启递归函数需改写为 sumNums(n - 1) > 0 ，此整体作为一个布尔量输出，否则会报错；
    //初始化变量 resres 记录结果。（ Java 可使用第二栏的简洁写法，不用借助变量 resres ）。
    public int sumNums(int n) {
        // 当 n = 1 时 n > 1 不成立 ，此时 “短路” ，终止后续递归
        boolean x = n > 1 && (n += sumNums(n - 1)) > 0;
        return n;
    }
}
```

剑指Offer 65（不用加减乘除做加法）

写一个函数，求两个整数之和，要求在函数体内不得使用 “+”、“-”、“*”、“/” 四则运算符号。

示例:

```
输入: a = 1, b = 1
输出: 2
```

提示:

- `a`, `b` 均可能是负数或 0
- 结果不会溢出 32 位整数

```java
// 位运算真不会，得看
class Solution {
    public int add(int a, int b) {
        return b == 0 ? a : add(a ^ b, (a & b) << 1);
    }
}
```

剑指Offer 66（构建乘积数组）

给定一个数组 `A[0,1,…,n-1]`，请构建一个数组 `B[0,1,…,n-1]`，其中 `B` 中的元素 `B[i]=A[0]×A[1]×…×A[i-1]×A[i+1]×…×A[n-1]`。不能使用除法。

示例:

```
输入: [1,2,3,4,5]
输出: [120,60,40,30,24]
```

提示:

- 所有元素乘积之和不会溢出 32 位整数
- `a.length <= 100000`

```java
// 双指针累乘
class Solution {
    public int[] constructArr(int[] a) {
        int n = a.length;
        int left = 1, right = 1;      //left：从左边累乘，right：从右边累乘
        int[] ans = new int[n];
        Arrays.fill(ans, 1);
        for(int i = 0; i < n; ++i){
            //最终每个元素其左右乘积进行相乘得出结果
            ans[i] *= left;        //乘以其左边的乘积
            left *= a[i];
            ans[n - 1 - i] *= right;   //乘以其右边的乘积
            right *= a[n - 1 - i];
        }
```

```
        return ans;
    }
}
// 本质上是动态规划
class Solution {
    public int[] constructArr(int[] a) {
        int n = a.length;
        int[] B = new int[n];
        for (int i = 0, product = 1; i < n; product *= a[i], i++)       /* 从左往右累乘 */
            B[i] = product;
        for (int i = n - 1, product = 1; i >= 0; product *= a[i], i--)  /* 从右往左累乘 */
            B[i] *= product;
        return B;
    }
}
```

剑指Offer 67（把字符串转换成整数）

写一个函数 StrToInt，实现把字符串转换成整数这个功能。不能使用 atoi 或者其他类似的库函数。

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。

当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。

注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

说明：

假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为 $[-2^{31}, 2^{31} - 1]$。如果数值超过这个范围，请返回 INT_MAX ($2^{31} - 1$) 或 INT_MIN (-2^{31}) 。

示例 1:

```
输入: "42"
输出: 42
```

示例 2:

```
输入: "   -42"
输出: -42
解释: 第一个非空白字符为 '-', 它是一个负号。
     我们尽可能将负号与后面所有连续出现的数字组合起来，最后得到 -42 。
```

示例 3:

```
输入: "4193 with words"
输出: 4193
解释: 转换截止于数字 '3' ，因为它的下一个字符不为数字。
```

示例 4:

```
输入: "words and 987"
输出: 0
解释: 第一个非空字符是 'w', 但它不是数字或正、负号。
     因此无法执行有效的转换。
```

示例 5:

```
输入: "-91283472332"
输出: -2147483648
解释: 数字 "-91283472332" 超过 32 位有符号整数范围。
     因此返回 INT_MIN ($-2^{31}$) 。
```

```java
class Solution {
    public int strToInt(String str) {
        char[] c = str.trim().toCharArray();
        if(c.length == 0)
            return 0;
        // limit = 214748364
        int res = 0, limit = Integer.MAX_VALUE / 10;
        int i = 1, sign = 1;
        // 若首字符为'-', 标记符号位为-1
        if(c[0] == '-')
            sign = -1;
        // 其余情况下，只要不是'+', 就都是非法，标记为0
        else if(c[0] != '+')
            i = 0;
        // 遍历剩余字符数组的每一位。
        for(int j = i; j < c.length; j++) {
            // 若char数组中含有非法字符，break
            if(c[j] < '0' || c[j] > '9')
                break;
```

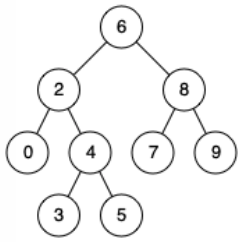
```
            // 结果需要小于2147483647，退一位计算比较方便
            if(res > limit || res == limit && c[j] > '7')
                return sign == 1 ? Integer.MAX_VALUE : Integer.MIN_VALUE;
            res = res * 10 + (c[j] - '0');
        }
        return sign * res;
    }
}
```

剑指Offer 68-1（二叉搜索树最近公共父节点）

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉搜索树: root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
输出: 6
解释: 节点 2 和节点 8 的最近公共祖先是 6。
```

示例 2:

```
输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
输出: 2
解释: 节点 2 和节点 4 的最近公共祖先是 2, 因为根据定义最近公共祖先节点可以为节点本身。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉搜索树中。

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        //当p和q节点等于root节点，直接返回root
        if (p.val == root.val || q.val == root.val){
            return root;
        }
        //递归求解，利用二叉搜索树性质，切记
        if (p.val > root.val && q.val > root.val) {//p和q节点都大于root，则说明p和q
为root的右子节点
            return lowestCommonAncestor(root.right, p, q);
```

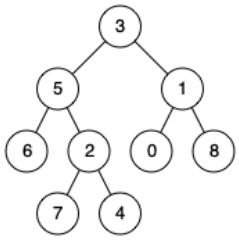
```
        }else if (p.val < root.val && q.val < root.val) {//p和q节点都小于root，则说明p和q为root的左子节点
            return lowestCommonAncestor(root.left, p, q);
        }else{//其他情况均为root节点为最大父节点
            return root;
        }
    }
}
```

剑指Offer 68-2（二叉树最近公共父节点）

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p、q，最近公共祖先表示为一个结点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

例如，给定如下二叉树: root = [3,5,1,6,2,0,8,null,null,7,4]



示例 1:

```
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
输出: 3
解释: 节点 5 和节点 1 的最近公共祖先是节点 3。
```

示例 2:

```
输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
输出: 5
解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。
```

说明:

- 所有节点的值都是唯一的。
- p、q 为不同节点且均存在于给定的二叉树中。

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        /** 经典递归
         注意p,q必然存在树内，且所有节点的值唯一!!!
         递归思想，对以root为根的(子)树进行查找p和q，如果root == null || p || q 直接返回root
         表示对于当前树的查找已经完毕，否则对左右子树进行查找，根据左右子树的返回值判断：
         1. 左右子树的返回值都不为null，由于值唯一左右子树的返回值就是p和q，此时root为LCA
         2. 如果左右子树返回值只有一个不为null，说明只有p和q存在与左或右子树中，最先找到的那个节点为LCA
```

```
         3. 左右子树返回值均为null, p和q均不在树中, 返回null
         **/
        if (root == null || root == p || root == q) return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if (left != null && right != null) {
            return root;
        } else if (left != null) {
            return left;
        } else if (right != null) {
            return right;
        }
        return null;
    }
}
// 大佬题解
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        if(root == null) return null; // 如果树为空，直接返回null
        if(root == p || root == q) return root; // 如果 p和q中有等于 root的，那么它
们的最近公共祖先即为root（一个节点也可以是它自己的祖先）
        TreeNode left = lowestCommonAncestor(root.left, p, q); // 递归遍历左子树，
只要在左子树中找到了p或q，则先找到谁就返回谁
        TreeNode right = lowestCommonAncestor(root.right, p, q); // 递归遍历右子
树，只要在右子树中找到了p或q，则先找到谁就返回谁
        if(left == null) return right; // 如果在左子树中 p和 q都找不到，则 p和 q一定
都在右子树中，右子树中先遍历到的那个就是最近公共祖先（一个节点也可以是它自己的祖先）
        else if(right == null) return left; // 否则，如果 left不为空，在左子树中有找
到节点（p或q），这时候要再判断一下右子树中的情况，如果在右子树中，p和q都找不到，则 p和q一定都
在左子树中，左子树中先遍历到的那个就是最近公共祖先（一个节点也可以是它自己的祖先）
        else return root; //否则，当 left和 right均不为空时，说明 p、q节点分别在 root
异侧，最近公共祖先即为 root
    }
}
```