剑指 Offer（第 2 版）

剑指 Offer 03. 数组中重复的数字

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

```
class Solution {
    public int findRepeatNumber(int[] nums) {
        int[] tmp = new int[nums.length];
        int ans = -1;

        for(int i = 0 ; i < nums.length; i ++){
            tmp[nums[i]] += 1;
            if(tmp[nums[i]]>1){
                ans = nums[i];
                break;
            }
        }

        return ans;

    }
}
```

剑指 Offer 04. 二维数组中的查找

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。请完成一个高效的函数，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

```
class Solution {
    public boolean findNumberIn2DArray(int[][] matrix, int target) {

        if(matrix == null || matrix.length <= 0 || matrix[0].length <=0) return false;
        int m = matrix[0].length-1;

        for(int i = 0 ; i < matrix.length; i++){
            if(target>matrix[i][m]) continue;
            if(target< matrix[i][m]){
                for(int j = 0; j < m ; j++){
                    if(target == matrix[i][j])
                        return true;
                }
            }

            else return true;

        }

        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

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

        for(int i = 0 ; i <s.length(); i++){

            if(s.charAt(i) == ' '){
                str.append("%20");
            }

            else
                str.append(s.charAt(i));
        }

        return str.toString();
    }
}
```

```java
//官方答案
class Solution {
    public String replaceSpace(String s) {
        StringBuilder res = new StringBuilder();
        for(Character c : s.toCharArray())
        {
            if(c == ' ') res.append("%20");
            else res.append(c);
        }
        return res.toString();
    }
}
```

剑指 Offer 06. 从尾到头打印链表

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

```java
class Solution {
    public int[] reversePrint(ListNode head) {


        Stack<ListNode> stack = new Stack<>();


        while(head != null){
            stack.push(head);
            head = head.next;

        }

        int count = stack.size();
        int[] res = new int[count];
        for(int i = 0 ; i < count; i++){
            res[i] = stack.pop().val;
        }

        return res;
    }
}
```

剑指 Offer 07. 重建二叉树

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

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

class Solution {
    private Map<Integer, Integer> indexMap;

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        int n = preorder.length;
        // 构造哈希映射，帮助我们快速定位根节点
        indexMap = new HashMap<Integer, Integer>();
        for (int i = 0; i < n; i++) {
            indexMap.put(inorder[i], i);
        }
        return myBuildTree(preorder, inorder, 0, n - 1, 0, n - 1);
    }


      public TreeNode myBuildTree(int[] preorder, int[] inorder, int preorder_left,
int preorder_right, int inorder_left, int inorder_right) {
        if (preorder_left > preorder_right) {
            return null;
        }

        // 前序遍历中的第一个节点就是根节点
        int preorder_root = preorder_left;
        // 在中序遍历中定位根节点
        int inorder_root = indexMap.get(preorder[preorder_root]);

        // 先把根节点建立出来
        TreeNode root = new TreeNode(preorder[preorder_root]);
        // 得到左子树中的节点数目
        int size_left_subtree = inorder_root - inorder_left;
        // 递归地构造左子树，并连接到根节点

        root.left = myBuildTree(preorder, inorder, preorder_left + 1, preorder_left +
size_left_subtree, inorder_left, inorder_root - 1);
        // 递归地构造右子树，并连接到根节点
```

```
        root.right = myBuildTree(preorder, inorder, preorder_left + size_left_subtree
+ 1, preorder_right, inorder_root + 1, inorder_right);
        return root;
    }
}
```

```
//迭代
class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if (preorder == null || preorder.length == 0) {
            return null;
        }
        TreeNode root = new TreeNode(preorder[0]);
        Deque<TreeNode> stack = new LinkedList<TreeNode>();
        stack.push(root);
        int inorderIndex = 0;
        for (int i = 1; i < preorder.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;
    }
}
```

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
```

```
class CQueue {
    Deque<Integer> stack1;
    Deque<Integer> stack2;

    public CQueue() {
        stack1 = new LinkedList<Integer>();
        stack2 = new LinkedList<Integer>();
    }

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

    public int deleteHead() {
        // 如果第二个栈为空
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        if (stack2.isEmpty()) {
            return -1;
        } else {
            int deleteItem = stack2.pop();
            return deleteItem;
        }
    }
}
```

剑指 Offer 10- I. 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（即 F(N)）。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
```

```
class Solution {
    public int fib(int n) {
        int a = 0, b = 1, sum;
        for(int i = 0; i < n; i++){
            sum = (a + b) % 1000000007;
            a = b;
            b = sum;
        }
        return a;
    }
}
```

剑指 Offer 10- II. 青蛙跳台阶问题

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
```

```
class Solution {
    public int numWays(int n) {
        int a = 1; int b = 1; int sum = 0;
       for(int i = 0; i < n; i ++){
            sum = (a +b ) %1000000007;
            a = b;
            b = sum;
        }

        return a;
    }
}
```

剑指 Offer 11. 旋转数组的最小数字

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
```

```
class Solution {
    public int minArray(int[] numbers) {
        int low = 0;
        int high = numbers.length - 1;
        while (low < high) {
            int pivot = low + (high - low) / 2;
            if (numbers[pivot] < numbers[high]) {
                high = pivot;
            } else if (numbers[pivot] > numbers[high]) {
                low = pivot + 1;
            } else {
                high -= 1;
            }
        }
        return numbers[low];
    }
}
```

剑指 Offer 12. 矩阵中的路径

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

```
class Solution {
    int[] dx = {-1, 1, 0, 0};
    int[] dy = {0, 0, -1, 1};
```

```
    int m, n;

    public boolean exist(char[][] board, String word) {
        m = board.length;
        n = board[0].length;
        char[] chars = word.toCharArray();
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                if(board[i][j] == chars[0]){
                    board[i][j] = '/';
                    if(dfs(i, j, 1, board, chars)){
                        return true;
                    }
                    board[i][j] = chars[0];
                }
            }
        }
        return false;
    }

    private boolean dfs(int x, int y, int index, char[][] board, char[] chars){
        if(index == chars.length){
            return true;
        }

        for(int i = 0; i < 4; i++){
            int nx = x + dx[i], ny = y + dy[i];
            if(nx >= 0 && nx < m && ny >= 0 && ny < n && board[nx][ny] ==
chars[index]){
                board[nx][ny] = '/';
                if(dfs(nx, ny, index + 1, board, chars)){
                    return true;
                }
                board[nx][ny] = chars[index];
            }
        }
        return false;
    }
}
```

```
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;
        board[i][j] = '\0';
        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] = word[k];
        return res;
    }
}
```

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 {
    public int movingCount(int m, int n, int k) {
        if (k == 0) {
            return 1;
        }
        Queue<int[]> queue = new LinkedList<int[]>();
        // 向右和向下的方向数组
        int[] dx = {0, 1};
        int[] dy = {1, 0};
        boolean[][] vis = new boolean[m][n];
        queue.offer(new int[]{0, 0});
        vis[0][0] = true;
        int ans = 1;
        while (!queue.isEmpty()) {
            int[] cell = queue.poll();
            int x = cell[0], y = cell[1];
            for (int i = 0; i < 2; ++i) {
                int tx = dx[i] + x;
                int ty = dy[i] + y;
                if (tx < 0 || tx >= m || ty < 0 || ty >= n || vis[tx][ty] || get(tx) +
get(ty) > k) {
                    continue;
                }
                queue.offer(new int[]{tx, ty});
                vis[tx][ty] = true;
                ans++;
            }
        }
        return ans;
```

```
    }

    private int get(int x) {
        int res = 0;
        while (x != 0) {
            res += x % 10;
            x /= 10;
        }
        return res;
    }
}
```

剑指 Offer 14- I. 剪绳子

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

```
class Solution {
    public int cuttingRope(int n) {
        int[] dp = new int[n + 1];
        dp[2] = 1;
        for(int i = 3; i < n + 1; i++){  //动态规划，长度从3开始，逐步判断直到n
            for(int j = 2; j < i; j++){  //先把绳子减去j，剩下（i-j）部分再决定减不减
                dp[i] = Math.max(dp[i], Math.max(j * (i - j), j * dp[i - j]));
            }
        }
        return dp[n];
    }
}
```

剑指 Offer 14- II. 剪绳子 II

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

这题再用动态规划的话惨不忍睹，还是要看贪心
核心思路是：**尽可能把绳子分成长度为3的小段，这样乘积最大**

证明链接：https://leetcode-cn.com/problems/jian-sheng-zi-lcof/solution/mian-shi-ti-14-i-jian-sheng-zi-tan-xin-si-xiang-by/

步骤如下：

1. 如果 n == 2，返回1，如果 n == 3，返回2，两个可以合并成n小于4的时候返回n - 1
2. 如果 n == 4，返回4
3. 如果 n > 4，分成尽可能多的长度为3的小段，每次循环长度n减去3，乘积res乘以3；最后返回时乘以小于等于4的最后一小段；每次乘法操作后记得取余就行
4. 以上2和3可以合并

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

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（以二进制串形式），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'。
```

提示：

- 输入必须是长度为 32 的 二进制串 。

```
//通过最后一位做相与运算的方式
public class Solution {
    public int hammingWeight(int n) {
        int res = 0;
        while(n != 0) {
            res += n & 1;
            n >>>= 1;  //无符号右移
        }
        return res;
    }
}
```

```
//Java语言的特点
public class Solution {
    public int hammingWeight(int n) {
       return Integer.bitCount(n);
    }
}

//其中bitCount函数的实现源代码如下：
public static int bitCount(int i) {
    // HD, Figure 5-2
    i = i - ((i >>> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
    i = (i + (i >>> 4)) & 0x0f0f0f0f;
    i = i + (i >>> 8);
    i = i + (i >>> 16);
    return i & 0x3f;
}
```

剑指 Offer 16. 数值的整数次方

实现 pow(x, n) ，即计算 x 的 n 次幂函数（即，xn）。不得使用库函数，同时不需要考虑大数问题。

示例 1：

```
输入：x = 2.00000, n = 10
输出：1024.00000
```

示例 2：

```
输入：x = 2.10000, n = 3
输出：9.26100
```

示例 3：

```
输入：x = 2.00000, n = -2
输出：0.25000
解释：2-2 = 1/22 = 1/4 = 0.25
```

提示：

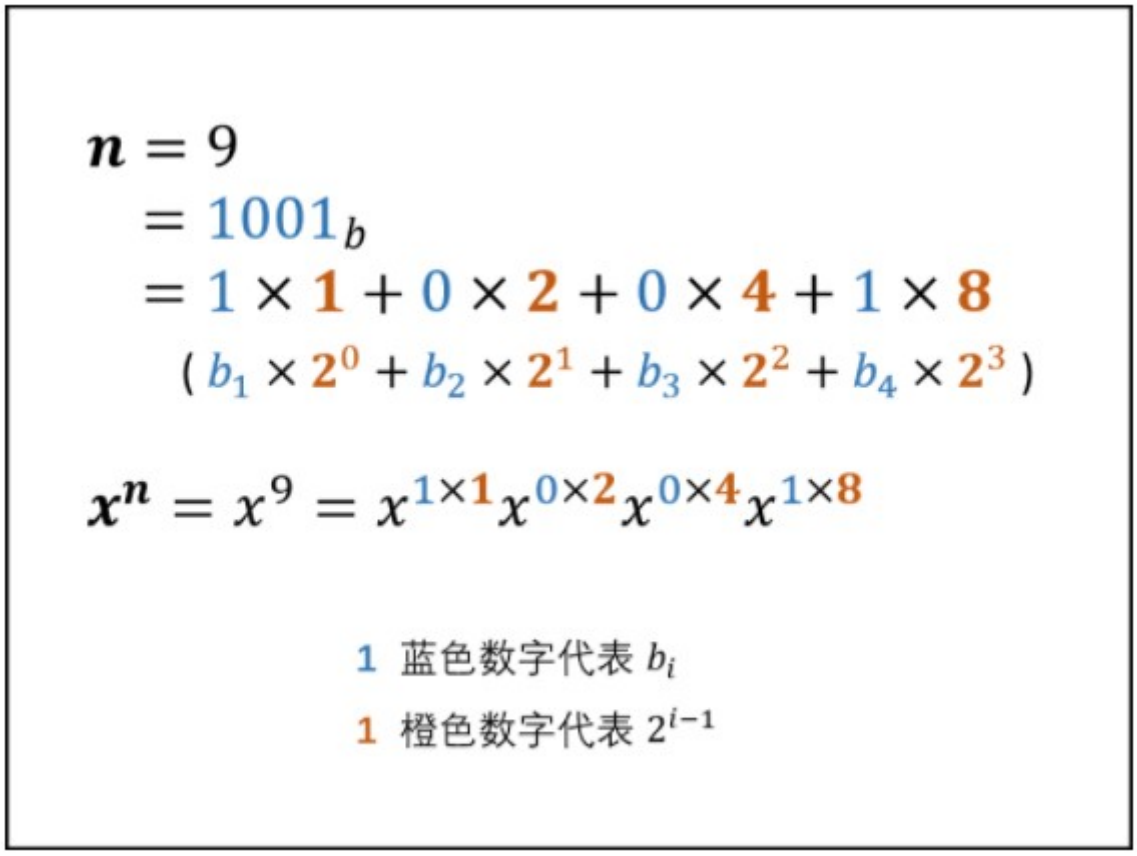
- -100.0 < x < 100.0
- -2^{31} <= n <= $2^{31}-1$
- -10^4 <= x^n <= 10^4

算法流程：

1. 当 x = 0x=0 时：直接返回 00 （避免后续 x = 1 / xx=1/x 操作报错）。
2. 初始化 res = 1res=1；
3. 当 n < 0n<0 时：把问题转化至 n \geq 0n≥0 的范围内，即执行 x = 1/xx=1/x ， n = - nn=−n；
4. 循环计算：当 n = 0n=0 时跳出；
    1. 当 n & 1 = 1n&1=1 时：将当前 xx 乘入 resres （即 res *= xres∗=x ）；
    2. 执行 x = x^2x=x (即 x *= xx∗=x ）；
    3. 执行 nn 右移一位（即 n >>= 1n>>=1）。
5. 返回 resres 。

复杂度分析：

时间复杂度 O($log_2 n$)： 二分的时间复杂度为对数级别。
空间复杂度 O(1)： res, b 等变量占用常数大小额外空间。




```
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;  //在本题中可理解为删除最后一位
           //使用 >>= 运算符和使用下面的语句是等效的：
           //result = result >> expression
           // >>= 运算符把 result 的所有位向右移 expression 指定的位数。result 的符号位被用来填充右移后左边空出的位。从右边移出去的位被丢弃。例如，下面的代码被求值后，temp 的值是 -4：-14 （即二进制的 11110010）右移两位等于 -4 （即二进制的 11111100）。
        }
        return res;
    }
}
```

```
//递归的方式
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;
    }
}
```

剑指 Offer 17. 打印从1到最大的n位数

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

```
class Solution {
    int[] res;
    int count = 0;

    public int[] printNumbers(int n) {
        res = new int[(int)Math.pow(10, n) - 1];
        for(int digit = 1; digit < n + 1; digit++){
            for(char first = '1'; first <= '9'; first++){
                char[] num = new char[digit];
                num[0] = first;
                dfs(1, num, digit);
            }
        }
        return res;
    }

    private void dfs(int index, char[] num, int digit){
        if(index == digit){
```

```
            res[count++] = Integer.parseInt(String.valueOf(num));
            return;
        }
        for(char i = '0'; i <= '9'; i++){
            num[index] = i;
            dfs(index + 1, num, digit);
        }
    }
}
```

剑指 Offer 18. 删除链表的节点

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

```
class Solution {
    public ListNode deleteNode(ListNode head, int val) {
        if (head == null) return null;
        if (head.val == val) return head.next;
        ListNode cur = head;
        while (cur.next != null && cur.next.val != val)
            cur = cur.next;
        if (cur.next != null)
            cur.next = cur.next.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*'*'。

```java
class Solution {
    public boolean isMatch(String s, String p) {
        int m = s.length();
        int n = p.length();
        boolean f[][] = new boolean[m + 1][n + 1];
        f[0][0] = true;
        //f[0][0]代表s和p均为空字符串，f[1][1]代表s和p的第一个字符（即在s和p中下标为0的字符）
        for(int i = 0; i <= m ; ++i) {
            for(int j = 1; j <= n; ++j) {
                if(p.charAt(j - 1) == '*') {//p的第j个字符为*
                    if(matches(s, p, i, j - 1)) {//匹配s的第i个字符和p的第j-1个字符
                        f[i][j] = f[i - 1][j] || f[i][j - 2];
                        //p中*前面的字符在s中出现多次或者在s中只出现1次
                    }
                    else {
                        f[i][j] = f[i][j - 2];//p中*前面的在s中字符出现0次
                    }
```

```
                }
                else {//p的第j个字符不为*
                    if(matches(s, p, i, j)) {//匹配s的第i个字符和p的第j个字符
                        f[i][j] = f[i - 1][j - 1];//匹配成功，状态转移；匹配不成功，默认是
false
                    }
                }
            }
        }
        return f[m][n];
    }

    private boolean matches(String s, String p, int i, int j) {//注意在字符串中的下标变换
        if(i == 0) { //代表是空字符，直接返回
            return false;
        }
        if(p.charAt(j - 1) == '.') {
            return true;
        }
        return s.charAt(i - 1) == p.charAt(j - 1);
    }
}
```

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。

```
class Solution {
    public boolean isNumber(String s) {
        int n = s.length();
        int index = 0;
        boolean hasNum = false, hasE = false;
        boolean hasSign = false, hasDot = false;
        while(index < n && s.charAt(index) == ' ')
            index++;
        while(index < n){
            while(index < n && s.charAt(index) >= '0' && s.charAt(index) <= '9'){
                index++;
                hasNum = true;
            }
            if(index == n){
                break;
            }
            char c = s.charAt(index);
            if(c == 'e' || c == 'E'){
                if(hasE || !hasNum){
                    return false;
                }
                hasE = true;
                hasNum = false; hasSign = false; hasDot = false;
            }else if(c == '+' || c == '-'){
                if(hasSign || hasNum || hasDot){
                    return false;
                }
                hasSign = true;
            }else if(c == '.'){
```

```
                if(hasDot || hasE){
                    return false;
                }
                hasDot = true;
            }else if(c == ' '){
                break;
            }else{
                return false;
            }
            index++;
        }
        while(index < n && s.charAt(index) == ' ')
            index++;
        return hasNum && index == n;
    }
}
```

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：

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

```
class Solution {
    public int[] exchange(int[] nums) {
        int left = 0, right = nums.length - 1;
        while(left <= right){
            while(left <= right && nums[left] % 2 == 1)
                left++;
            while(left <= right && nums[right] % 2 == 0)
                right--;
            if(left > right)
                break;
            int tmp = nums[left];
            nums[left] = nums[right];
            nums[right] = tmp;
        }
        return nums;
    }
}
```

剑指 Offer 22. 链表中倒数第k个节点

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.
```

```
class Solution {
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode slow = head, fast = head;
        for(int i = 0; i < k; i++){
            fast = fast.next;
        }
        while(fast != null){
            fast = fast.next;
            slow = slow.next;
        }
        return slow;
    }
}
```

剑指 Offer 24. 反转链表

反转一个单链表。

示例:

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

进阶:
你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

```
class Solution {
    public ListNode reverseList(ListNode head) {
         //单链表为空或只有一个节点，直接返回原单链表
        if (head == null || head.next == null){
            return head;
        }
        //前一个节点指针
        ListNode preNode = null;
        //当前节点指针
        ListNode curNode = head;
        //下一个节点指针
        ListNode nextNode = null;
        while (curNode != null){
            nextNode = curNode.next;//nextNode 指向下一个节点,保存当前节点后面的链表。
            curNode.next=preNode;//将当前节点next域指向前一个节点   null<-1<-2<-3<-4
            preNode = curNode;//preNode 指针向后移动。preNode指向当前节点。
            curNode = nextNode;//curNode指针向后移动。下一个节点变成当前节点
        }
        return preNode;
    }
}
```

剑指 Offer 25. 合并两个排序的链表

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

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1 == null) {
            return l2;
        }
        if(l2 == null) {
            return l1;
        }

        if(l1.val < l2.val) {
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        } else {
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
}
```

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

```
class Solution {
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        return (A != null && B != null) && (recur(A, B) || isSubStructure(A.left, B)
|| isSubStructure(A.right, B));
    }
    boolean recur(TreeNode A, TreeNode B) {
        if(B == null) return true;
        if(A == null || A.val != B.val) return false;
        return recur(A.left, B.left) && recur(A.right, B.right);
    }
}
```

```
class Solution {
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        if(B == null) return false;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(A);
        while(!queue.isEmpty()){
            TreeNode node = queue.poll();
            if(node.val == B.val){
                if(helper(node, B)){
                    return true;
                }
            }
            if(node.left != null){
                queue.offer(node.left);
            }
            if(node.right != null){
                queue.offer(node.right);
            }
        }
        return false;
    }

    private boolean helper(TreeNode nodeA, TreeNode nodeB){
        Queue<TreeNode> queueA = new LinkedList<>();
        Queue<TreeNode> queueB = new LinkedList<>();
        queueA.offer(nodeA);
        queueB.offer(nodeB);

        while(!queueB.isEmpty()){
            nodeA = queueA.poll();
            nodeB = queueB.poll();
            if(nodeA == null || nodeA.val != nodeB.val){
                return false;
            }
            if(nodeB.left != null){
                queueA.offer(nodeA.left);
                queueB.offer(nodeB.left);
```

```
            }
            if(nodeB.right != null){
                queueA.offer(nodeA.right);
                queueB.offer(nodeB.right);
            }
        }
        return true;
    }

}
```

```
class Solution {
    private TreeNode B;
    public boolean isSubStructure(TreeNode A, TreeNode B) {
        if(B == null) return false;
        this.B = B;
        return dfs(A);
    }

    private boolean dfs(TreeNode nodeA){
        if(nodeA == null)
            return false;
        if(nodeA.val == B.val)
            if(helper(nodeA, B))
                return true;
        return dfs(nodeA.left) || dfs(nodeA.right);
    }

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

剑指 Offer 27. 二叉树的镜像

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：

```
      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

```
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if(root == null) return null;
        Stack<TreeNode> stack = new Stack<>() {{ add(root); }};
        while(!stack.isEmpty()) {
            TreeNode node = stack.pop();
            if(node.left != null) stack.add(node.left);
            if(node.right != null) stack.add(node.right);
            TreeNode tmp = node.left;
            node.left = node.right;
            node.right = tmp;
        }
        return root;
    }
}
```

```
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if(root == null) return null;
        TreeNode tmp = root.left;
        root.left = mirrorTree(root.right);
        root.right = mirrorTree(tmp);
        return root;
    }
}
```

剑指 Offer 28. 对称的二叉树

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

```
class Solution {
    public boolean isSymmetric(TreeNode root) {
        return root == null ? true : recur(root.left, root.right);
    }
    boolean recur(TreeNode L, TreeNode R) {
        if(L == null && R == null) return true;
        if(L == null || R == null || L.val != R.val) return false;
        return recur(L.left, R.right) && recur(L.right, R.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

```
class Solution {
    public int[] spiralOrder(int[][] matrix) {
        if(matrix.length == 0) return new int[0];
        int l = 0, r = matrix[0].length - 1, t = 0, b = matrix.length - 1, x = 0;
        int[] res = new int[(r + 1) * (b + 1)];
        while(true) {
            for(int i = l; i <= r; i++) res[x++] = matrix[t][i]; // left to right.
            if(++t > b) break;
            for(int i = t; i <= b; i++) res[x++] = matrix[i][r]; // top to bottom.
            if(l > --r) break;
            for(int i = r; i >= l; i--) res[x++] = matrix[b][i]; // right to left.
            if(t > --b) break;
            for(int i = b; i >= t; i--) res[x++] = matrix[i][l]; // bottom to top.
            if(++l > r) break;
        }
        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:

各函数的调用总次数不超过 20000 次

```
class MinStack {
    Stack<Integer> A, B;
    public MinStack() {
        A = new Stack<>();
        B = new Stack<>();
    }
    public void push(int x) {
        A.add(x);
        if(B.empty() || B.peek() >= x)
            B.add(x);
    }
    public void pop() {
        if(A.pop().equals(B.peek()))
            B.pop();
    }
    public int top() {
        return A.peek();
    }
    public int min() {
        return B.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：

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

```
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
            while(!stack.isEmpty() && stack.peek() == popped[i]) { // 循环判断与出栈
                stack.pop();
                i++;
            }
        }
        return stack.isEmpty();
    }
}
```

剑指 Offer 32 - I. 从上到下打印二叉树

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：

节点总数 <= 1000

```
class Solution {
    public int[] levelOrder(TreeNode root) {
        if(root == null) return new int[0];
        Queue<TreeNode> queue = new LinkedList<>(){{ add(root); }};
        ArrayList<Integer> ans = new ArrayList<>();
        while(!queue.isEmpty()) {
            TreeNode node = queue.poll();
            ans.add(node.val);
            if(node.left != null) queue.add(node.left);
            if(node.right != null) queue.add(node.right);
        }
        int[] res = new int[ans.size()];
        for(int i = 0; i < ans.size(); i++)
```

```
            res[i] = ans.get(i);
        return res;
    }
}
```

剑指 Offer 32 - II. 从上到下打印二叉树 II

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

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        if(root != null) queue.add(root);
        while(!queue.isEmpty()) {
            List<Integer> tmp = new ArrayList<>();
            for(int i = queue.size(); i > 0; i--) { //这里一定要从高到低，不然队列的元素数量变化就会出错
                TreeNode node = queue.poll();
                tmp.add(node.val);
                if(node.left != null) queue.add(node.left);
                if(node.right != null) queue.add(node.right);
            }
            res.add(tmp);
        }
        return res;
    }
}
```

剑指 Offer 32 - III. 从上到下打印二叉树 III

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：

节点总数 <= 1000

```java
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        if(root != null) queue.add(root);
        while(!queue.isEmpty()) {
            LinkedList<Integer> tmp = new LinkedList<>();
            for(int i = queue.size(); i > 0; i--) {
                TreeNode node = queue.poll();
                if(res.size() % 2 == 0) tmp.addLast(node.val); // 偶数层 -> 队列头部
                else tmp.addFirst(node.val); // 奇数层 -> 队列尾部
                if(node.left != null) queue.add(node.left);
                if(node.right != null) queue.add(node.right);
            }
            res.add(tmp);
        }
        return res;
    }
}
```

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:

数组长度 <= 1000

```
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);
    }
}
```

```
class Solution {
    public boolean verifyPostorder(int[] postorder) {
        Stack<Integer> stack = new Stack<>();
        int root = Integer.MAX_VALUE;
        for(int i = postorder.length - 1; i >= 0; i--) {
            if(postorder[i] > root) return false;
            while(!stack.isEmpty() && stack.peek() > postorder[i])
                root = stack.pop();
            stack.add(postorder[i]);
        }
        return true;
    }
}
```

剑指 Offer 34. 二叉树中和为某一值的路径

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 target = 22，

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

返回:

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

提示：

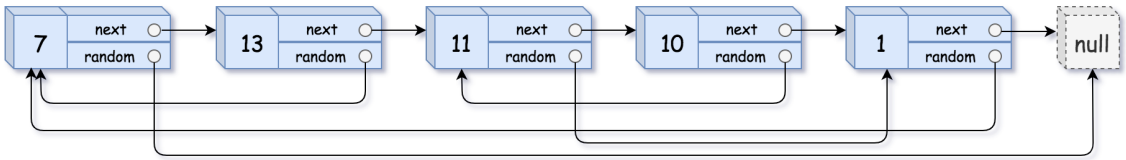
节点总数 <= 10000

```
class Solution {
    LinkedList<List<Integer>> res = new LinkedList<>();
    LinkedList<Integer> path = new LinkedList<>();
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        recur(root, sum);
        return res;
    }
    void recur(TreeNode root, int tar) {
        if(root == null) return;
        path.add(root.val);
        tar -= root.val;
        if(tar == 0 && root.left == null && root.right == null)
            res.add(new LinkedList(path));
        recur(root.left, tar);
        recur(root.right, tar);
        path.removeLast();
    }
}
```

剑指 Offer 35. 复杂链表的复制

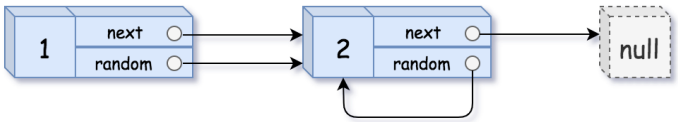
请实现 copyRandomList 函数，复制一个复杂链表。在复杂链表中，每个节点除了有一个 next 指针指向下一个节点，还有一个 random 指针指向链表中的任意节点或者 null。

示例 1：



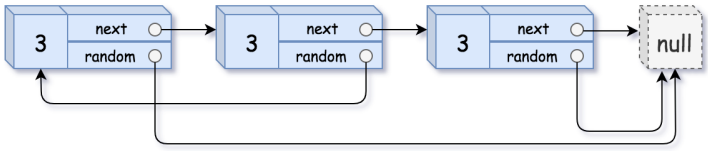

```
输入：head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
输出：[[7,null],[13,0],[11,4],[10,2],[1,0]]
```

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 。

```java
class Solution {
    public Node copyRandomList(Node head) {
        if(head == null) return null;
        Node cur = head;
        Map<Node, Node> map = new HashMap<>();
        // 3. 复制各节点，并建立 “原节点 -> 新节点” 的 Map 映射
        while(cur != null) {
            map.put(cur, new Node(cur.val));
            cur = cur.next;
        }
        cur = head;
        // 4. 构建新链表的 next 和 random 指向
        while(cur != null) {
            map.get(cur).next = map.get(cur.next);
            map.get(cur).random = map.get(cur.random);
            cur = cur.next;
        }
        // 5. 返回新链表的头节点
        return map.get(head);
    }
}
```

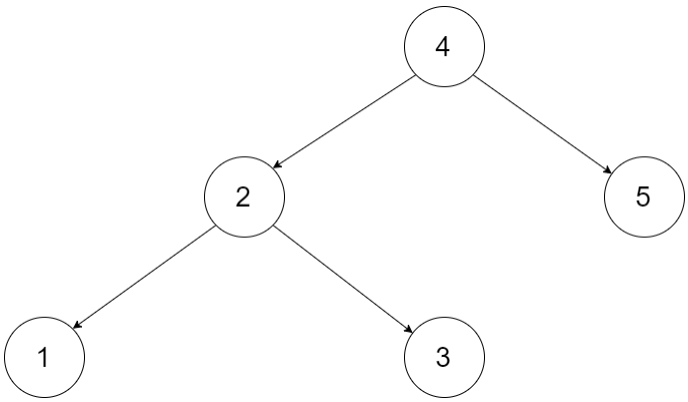
```java
class Solution {
    public Node copyRandomList(Node head) {
        if(head == null) return null;
        Node cur = head;
        // 1. 复制各节点，并构建拼接链表
        while(cur != null) {
            Node tmp = new Node(cur.val);
            tmp.next = cur.next;
            cur.next = tmp;
            cur = tmp.next;
        }
        // 2. 构建各新节点的 random 指向
        cur = head;
        while(cur != null) {
```

```
            if(cur.random != null)
                cur.next.random = cur.random.next;
            cur = cur.next.next;
        }
        // 3. 拆分两链表
        cur = head.next;
        Node pre = head, res = head.next;
        while(cur.next != null) {
            pre.next = pre.next.next;
            cur.next = cur.next.next;
            pre = pre.next;
            cur = cur.next;
        }
        pre.next = null; // 单独处理原链表尾节点
        return res;      // 返回新链表头节点
    }
}
```

剑指 Offer 36. 二叉搜索树与双向链表

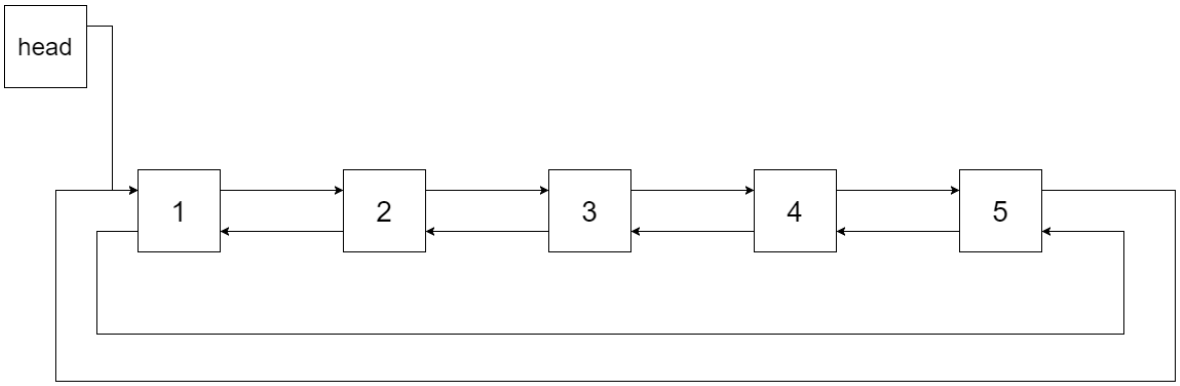
输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的循环双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。

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。

```
class Solution {
    Node pre, head;
    public Node treeToDoublyList(Node root) {
```

```
        if(root == null) return null;
        dfs(root);
        head.left = pre;
        pre.right = head;
        return head;
    }
    void dfs(Node cur) {
        if(cur == null) return;
        dfs(cur.left);
        if(pre != null) pre.right = cur;
        else head = cur;
        cur.left = pre;
        pre = cur;
        dfs(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：

```
    1
   / \
  2   3
     / \
    4   5

序列化为 "[1,2,3,null,null,4,5]"
```

```
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;
    }
}
```

剑指 Offer 38. 字符串的排列

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 LinkedList<>();
    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

```
class Solution {
    public int majorityElement(int[] nums) {

        int key = nums[0];
        int count = 1 ;

        for(int i = 1 ; i < nums.length; i++){
            if(key != nums[i] && count > 0){
                count--;
            }
            else if(key != nums[i] && count == 0){
                key = nums[i];
            }

            else{
                count++;
            }

        }

        return key;
    }
}
```

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

```
//堆
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        int[] vec = new int[k];
        if (k == 0) { // 排除 0 的情况
            return vec;
        }
        PriorityQueue<Integer> queue = new PriorityQueue<Integer>(new
Comparator<Integer>() {
            public int compare(Integer num1, Integer num2) {
                return num2 - num1;
            }
        });
        for (int i = 0; i < k; ++i) {
            queue.offer(arr[i]);
        }
        for (int i = k; i < arr.length; ++i) {
            if (queue.peek() > arr[i]) {
                queue.poll();
                queue.offer(arr[i]);
            }
        }
        for (int i = 0; i < k; ++i) {
            vec[i] = queue.poll();
        }
        return vec;
    }
}
```

```
//快排的变形
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        if (k == 0) {
            return new int[0];
        } else if (arr.length <= k) {
            return arr;
        }

        // 原地不断划分数组
        partitionArray(arr, 0, arr.length - 1, k);

        // 数组的前 k 个数此时就是最小的 k 个数，将其存入结果
        int[] res = new int[k];
        for (int i = 0; i < k; i++) {
            res[i] = arr[i];
        }
        return res;
    }

    void partitionArray(int[] arr, int lo, int hi, int k) {
        // 做一次 partition 操作
        int m = partition(arr, lo, hi);
        // 此时数组前 m 个数，就是最小的 m 个数
        if (k == m) {
            // 正好找到最小的 k(m) 个数
            return;
```

```
        } else if (k < m) {
            // 最小的 k 个数一定在前 m 个数中，递归划分
            partitionArray(arr, lo, m - 1, k);
        } else {
            // 在右侧数组中寻找最小的 k-m 个数
            partitionArray(arr, m + 1, hi, k);
        }
    }

    public int partition(int[] list, int first, int last) {
        int pivot = list[first];
        int low = first + 1;
        int high = last;

        while (high > low) {

            while (low <= high && list[low] <= pivot)
                low++;

            while (low <= high && list[high] > pivot)
                high--;

            if (high > low) {
                int temp = list[high];
                list[high] = list[low];
                list[low] = temp;
            }
        }

        while (high > first && list[high] >= pivot)
            high--;

        if (pivot > list[high]) {
            list[first] = list[high];
            list[high] = pivot;
            return high;
        } else {
            return first;
        }
    }

}
```

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。

```
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) {
        int max = nums[0];
        int former = 0;//用于记录dp[i-1]的值，对于dp[0]而言，其前面的dp[-1]=0
```

```
        int cur = nums[0];//用于记录dp[i]的值
        for(int num:nums){
            cur = num;
            if(former>0) cur +=former;
            if(cur>max) max = cur;
            former=cur;
        }
        return max;
    }
}
```

剑指 Offer 43. 1 ~ n 整数中 1 出现的次数

输入一个整数 n ， 求1 ~ n这n个整数的十进制表示中1出现的次数。

例如，输入12，1 ~ 12这些整数中包含1 的数字有1、10、11和12，1一共出现了5次。

示例 1：

```
输入：n = 12
输出：5
```

示例 2：

```
输入：n = 13
输出：6
```

限制：

- 1 <= n < 2^31

解题思路

f(n))函数的意思是1 ~ n这n个整数的十进制表示中1出现的次数，将n拆分为两部分，最高一位的数字high和其他位的数字last，分别判断情况后将结果相加，看例子更加简单。

例子如n=1234，high=1, pow=1000, last=234

可以将数字范围分成两部分1 - 999和1000 - 1234

- 1 - 999这个范围1的个数是f(pow-1)
- 1000~1234这个范围1的个数需要分为两部分：
  - 千位是1的个数：千位为1的个数刚好就是234+1(last+1)，注意，这儿只看千位，不看其他位
  - 其他位是1的个数：即是234中出现1的个数，为f(last)

所以全部加起来是f(pow-1) + last + 1 + f(last);

```
class Solution {
    public int countDigitOne(int n) {
        int digit = 1, res = 0;
        int high = n / 10, cur = n % 10, low = 0;
        while(high != 0 || cur != 0) {
            if(cur == 0) res += high * digit;
            else if(cur == 1) res += high * digit + low + 1;
            else res += (high + 1) * digit;
            low += cur * digit;
            cur = high % 10;
            high /= 10;
            digit *= 10;
        }
```

```
        return res;
    }
}
```

```
//递归
class Solution {
    public int countDigitOne(int n) {
        return f(n);
    }

    private int f(int n ) {
        if (n <= 0)
            return 0;
        String s = String.valueOf(n);
        int high = s.charAt(0) - '0';
        int pow = (int) Math.pow(10, s.length()-1);
        int last = n - high*pow;
        if (high == 1) {
            return f(pow-1) + last + 1 + f(last);
        } else {
            return pow + high*f(pow-1) + f(last);
        }
    }
}
```

剑指 Offer 44. 数字序列中某一位的数字

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

```
class Solution {
    public int findNthDigit(int n) {

        int digital = 1; //几位数
        long start = 1; //n位数是从哪个数开始的(2位数是从10开始的)
        long count = 9; //n位数总共有多少个数字(2位数有2*90个数字)

        while(n > count){ //确定第n位对应是几位数
            n -= count;
            digital += 1;
            start *= 10;
            count = 9 * start * digital;
```

```
        }

        //确定是哪个数
        long sum = start + (n - 1)/digital; //因为start是从1开始的，所以要用n-1整除

        //确定在这个数的哪一位
        return Long.toString(sum).charAt((n - 1) % digital) - '0';
    }
}
```

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) {
        //把int数组转换成字符串的形式，用字符串进行比较
        String[] str = new String[nums.length];
        for(int i = 0 ; i < str.length; i++){
            str[i] = String.valueOf(nums[i]);
        }

        //字符串排序
        quickSort(str,0,str.length-1);

        //拼接排序好的字符串
        StringBuilder res = new StringBuilder();
        for(int i = 0 ; i < str.length; i++){
            res.append(str[i]);
        }

        return res.toString();
    }

    //用快排
    public void quickSort(String[] strs, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(strs, low, high);
            quickSort(strs, low, pivotIndex - 1);
```

```
            quickSort(strs, pivotIndex + 1, high);
        }
    }

    public int partition(String[] strs, int l, int r) {
        //数组的第一个数为基准元素
        String pivot = strs[l];
        int low = l + 1;
        int high = r;
        while (low < high) {

            while (low <= high && (strs[high] + pivot).compareTo(pivot + strs[high])
>= 0)
                high--;
            while (low <= high && (strs[low] + pivot).compareTo(pivot + strs[low]) <=
0)
                low++;

            if(low < high){
                String tmp = strs[high];
                strs[high] = strs[low];
                strs[low] = tmp;
            }
        }

        while(high > l && (strs[high] + pivot).compareTo(pivot + strs[high]) >= 0)
            high--;
        if((strs[high] + pivot).compareTo(pivot + strs[high]) < 0){
            strs[l] = strs[high];
            strs[high] = pivot;
            return high;
        }
        else{
            return l;
        }
    }

}
```

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 < 231

```
class Solution {
    public int translateNum(int num) {
        //相当于青蛙跳台阶的变式，如果num的一个长度为2的子串i在10-25之间，
```

```
        //表示有两种表示成字母的方法（分开表示或合在一起表示）

       String str = String.valueOf(num);

        int a = 1, b = 1; //a表示k-2, b 表示k-1
        for(int i = 2; i <= str.length(); i++){
            String tmp = str.substring(i-2,i); //不包括下标为i的子串
            int sum = (tmp.compareTo("10") >= 0 &&  tmp.compareTo("25")<= 0) ? a+b :
b;
            a = b;
            b = sum;
        }

        return b;
    }
}
```

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

```
class Solution {

    public int maxValue(int[][] grid) {
        int[][] dp = new int[grid.length + 1 ][grid[0].length + 1];

        for(int i = 1 ; i < dp.length; i++ ){
            for(int j = 1 ; j < dp[0].length; j++){
                if(i == 1 && j == 1)  dp[i][j] = grid[0][0];
                else dp[i][j] =  Math.max(dp[i-1][j], dp[i][j-1]) + grid[i-1][j-1];
            }
        }

        return dp[grid.length][grid[0].length];

    }


}
```

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

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        HashMap<Character,Integer> map = new HashMap<>();

        int res = 0, left = 0;
        for(int i = 0 ; i < s.length(); i++){
            if(map.containsKey(s.charAt(i))){
               left = Math.max(left, map.get(s.charAt(i)) + 1 );    //防止出现"abba"的情
形


            }

            map.put(s.charAt(i),i);
            res = Math.max(res, i - left + 1);
        }

        return  res;
    }
}
```

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 是丑数。

- n 不超过1690。

```
class Solution {
    public int nthUglyNumber(int n) {
        //新丑数在原丑数的基础上*2/3/5得到

        int[] arr = new int[n]; //存放前n个丑数

        int[] pos = new int[3]; // 对应2,3,5
        arr[0] = 1;
        for(int i = 1; i < n; i++){
            int a = arr[pos[0]] * 2;
            int b = arr[pos[1]] * 3;
            int c = arr[pos[2]] * 5;

            int minNum = Math.min(Math.min(a,b),c);

            if(a == minNum) pos[0]++; //6 = 2*3,a和c同时要更新
            if(b == minNum) pos[1]++;
            if(c == minNum) pos[2]++;

            arr[i] = minNum;
        }

        return arr[n-1];
    }
}
```

剑指 Offer 50. 第一个只出现一次的字符

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

```
class Solution {
    public char firstUniqChar(String s) {
        HashMap<Character, Integer> dic = new HashMap<>();
        char[] sc = s.toCharArray();
        for(char c : sc){
            if(!dic.containsKey(c))
                dic.put(c, 1);
            else{
                int value = dic.get(c);
                dic.put(c,++value);
            }

        }

        for(char c : sc)
```

```
            if(dic.get(c) == 1) return c;
        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

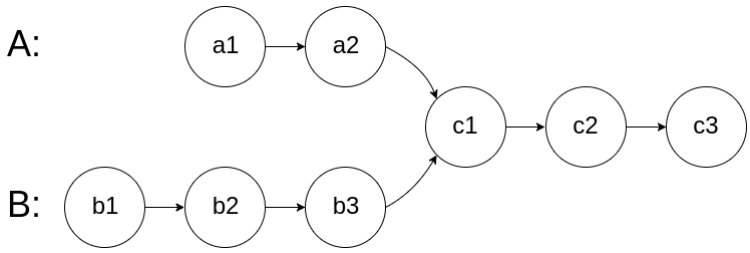
```
class Solution {
    public int reversePairs(int[] nums) {
        if(nums.length < 2) return 0;
        int[] tmp = new int[nums.length];

        return mergeSort(nums,0,nums.length-1,tmp);
    }

    public int mergeSort(int[] nums,int left, int right, int[] tmp){
        if(left >= right) return 0 ;
        int mid = left + (right - left)/2;

        int leftNum =  mergeSort(nums,left,mid,tmp);
        int rightNum = mergeSort(nums,mid+1,right,tmp);

        if(nums[mid]<= nums[mid+1]) return leftNum + rightNum;
        int crossNum = mergeSortCross(nums,left,mid,right,tmp);

        return leftNum + rightNum + crossNum;
    }

    public int mergeSortCross(int[] nums,int left,int mid, int right, int[] tmp){
        for(int i = left; i<= right; i++){
            tmp[i] = nums[i];
        }

        int i = left;
        int j = mid + 1;
        int count = 0;

        for(int k = left; k <= right ; k++){ //同时做排序和逆序的统计
            if(i > mid){
                nums[k] = tmp[j]; //这里顺便排好序，防止重复计数
                j++;
            }else if(j > right){
                nums[k] = tmp[i];
                i++;
            }else if(tmp[i]<= tmp[j]){
                nums[k] = tmp[i];
```

```
                i++;
            }else{
                nums[k] = tmp[j];
                j++;
                count += mid - i + 1; //如果i逆序了，那么i,mid都应该是逆序的，因为数组是递增的
            }
        }

        return count;

    }
}
```

剑指 Offer 52. 两个链表的第一个公共节点

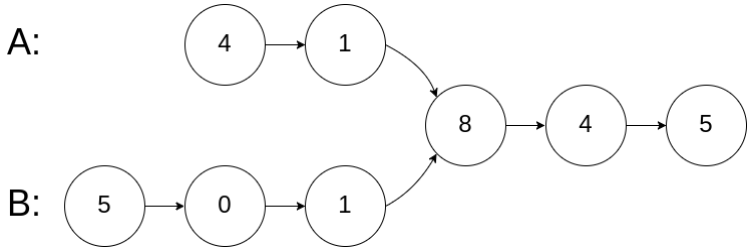
输入两个链表，找出它们的第一个公共节点。

如下面的两个链表：



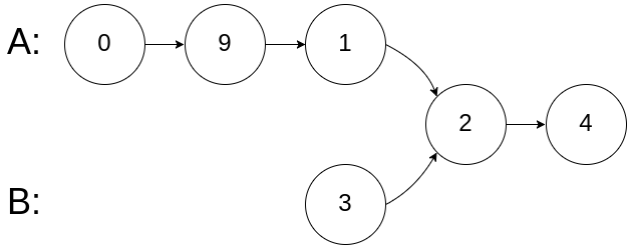
在节点 c1 开始相交。

示例 1：



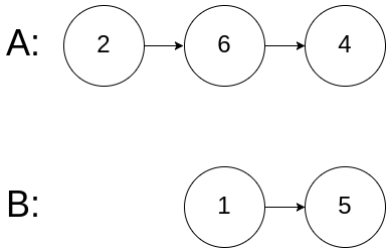

```
输入：intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5], skipA = 2, skipB = 3
输出：Reference of the node with value = 8
输入解释：相交节点的值为 8 （注意，如果两个列表相交则不能为 0）。从各自的表头开始算起，链表 A 为 [4,1,8,4,5]，链表 B 为 [5,0,1,8,4,5]。在 A 中，相交节点前有 2 个节点；在 B 中，相交节点前有 3 个节点。
```

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。
```

注意：

- 如果两个链表没有交点，返回 null.
- 在返回结果后，两个链表仍须保持原有的结构。
- 可假定整个链表结构中没有循环。
- 程序尽量满足 O(n) 时间复杂度，且仅用 O(1) 内存。

```java
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        ListNode tmpA = headA;
        ListNode tmpB = headB;

        //假如两个链表没有公共节点，就把NULL当作它们的公共节点，
        //所以两个判断是tmpA != tmpB
        while(tmpA != tmpB){
            if(tmpA == null)
                tmpA = headB;
            else
                tmpA = tmpA.next;
            if(tmpB == null)
                tmpB = headA;
            else
                tmpB = tmpB.next;
        }


        return tmpA;

    }
}
```

剑指 Offer 53 - I. 在排序数组中查找数字 I

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

```
class Solution {
    public int search(int[] nums, int target) {

        int start = binSearch(nums,target);
        int end = binSearch(nums,target+1);
        return end - start + (end <= nums.length-1 && nums[end] == target ? 1 : 0 );
        //end <= nums.length-1是为了判断当只有一个元素的时候nums[end]会发生数组下标越界
    }

    public int binSearch(int[] nums,int target){
        int l = 0, r = nums.length;
        while (l < r) {
            int mid = l + (r - l) / 2;
            if (nums[mid] < target) {
                l = mid + 1;
            } else {
                r = mid;
            }
        }
        return l;
    }

}
```

剑指 Offer 53 - II. 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) {
        int left = 0, right = nums.length;

        if(nums == null || nums.length == 0 )
            return 0;

        int mid = -1;
        while(left < right){
```

```
            mid = left + (right - left)/2 ;

            if(nums[mid] != mid )
                right = mid;
            else
                left = mid + 1;
        }

        if(mid == right-1)  //假如是[0,1,2],应该返回3
           mid++;

        return mid;
    }
}
```

剑指 Offer 54. 二叉搜索树的第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

```
class Solution {
    int tmp,res;
    public int kthLargest(TreeNode root, int k) {
        //中序遍历二叉搜索树是顺序的，只要按照右，根，左遍历就是逆序了
        tmp = k;
        inorder(root);
        return res;
    }

    public void inorder(TreeNode root){
        if(root == null || tmp == 0) return ;

        inorder(root.right);
        if(--tmp == 0) res = root.val;
```

```
        inorder(root.left);
    }
}
```

剑指 Offer 55 - I. 二叉树的深度

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

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

剑指 Offer 55 - II. 平衡二叉树

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：

- 0 <= 树的结点个数 <= 10000

```
class Solution {

    public boolean isBalanced(TreeNode root) {

        if(root == null) return true;

        int l = deep(root.left);
        int r = deep(root.right);

        return Math.abs( l - r ) <= 1 && isBalanced(root.left) &&
isBalanced(root.right);

    }

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

剑指 Offer 56 - I. 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

```
class Solution {
    public int[] singleNumbers(int[] nums) {
        //任何数异或自己都等于0，0异或任何数都等于这个数本身
        int sum = 0, m = 1 ;

        for(int x: nums)
           sum ^= x;
        //异或之后sum里至少有一位是为一的，找到这个位，并且用它分组
        while((sum & m) == 0)
            m <<= 1;    // 循环左移，计算 m

        int a = 0, b = 0;
        for(int num: nums) {              // 遍历 nums 分组
            if((num & m) != 0) a ^= num;  // 当 num & m != 0
```

```
            else b ^= num;                  // 当 num & m == 0
        }
        return new int[] {a, b};            // 返回出现一次的数字

    }
}
```

剑指 Offer 56 - II. 数组中数字出现的次数 II

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

```
class Solution {
    public int singleNumber(int[] nums) {
        int[] counts = new int[32];
        for(int num : nums) {
            for(int j = 0; j < 32; j++) {  //将每个数的每一位二进制位都记录在数组中
                counts[j] += num & 1;
                num >>>= 1;
            }
        }
        int res = 0, m = 3;
        for(int i = 0; i < 32; i++) {  //将数组中的数字对3取余，然后采用左移将原来的数还原
            res <<= 1;
            res += counts[31 - i] % m;
        }
        return res;
    }
}
```

剑指 Offer 57. 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

```
class Solution {
    public int[] twoSum(int[] nums, int target) {

        if(nums == null || nums.length == 0)
            return nums;


        int left = 0;
        int right = nums.length-1;
        int tmp;
        int[] res = new int[2];

        while(left< right){

            tmp = nums[left] + nums[right];

            if(tmp < target){
                left++;
            }else if(tmp > target){
                right--;
            }else{
                res[0] = nums[left];
                res[1] = nums[right];
                return res;
            }
        }

        return res;
    }
}
```

剑指 Offer 57 - II. 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

```
class Solution {
    public int[][] findContinuousSequence(int target) {
        int i = 1, j = 2, s = 3;
```

```
        List<int[]> res = new ArrayList<>();

        while(i < j) {
            if(s == target) {
                int[] ans = new int[j - i + 1];
                for(int k = i; k <= j; k++)
                    ans[k - i] = k;
                res.add(ans);
            }
            if(s >= target) {
                s -= i;
                i++;
            } else {
                j++;
                s += j;
            }
        }
        return res.toArray(new int[0][]);
    }
}
```

剑指 Offer 58 - I. 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[] strs = s.trim().split(" "); // 删除首尾空格，分割字符串
        StringBuilder res = new StringBuilder();
        for(int i = strs.length - 1; i >= 0; i--) { // 倒序遍历单词列表
            if(strs[i].equals("")) continue; // 遇到空单词则跳过
            res.append(strs[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 - II. 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 res = new StringBuilder();
        for(int i = n; i < n + s.length(); i++)
            res.append(s.charAt(i % s.length()));
        return res.toString();
    }
}
```

剑指 Offer 59 - I. 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。

```
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums == null || k < 1) return new int[0];

        Deque<Integer> q = new LinkedList<>();  //队列里存放的是nums的下标
        int[] res = new int[nums.length - k + 1];

        int index = 0;  //res的索引值

        for(int i = 0 ; i < nums.length; i ++){
            if(q.size() > 0 && i - q.peekFirst() >= k)
                q.pollFirst();
            while(q.size() > 0 && nums[i] > nums[q.peekLast()])
                q.pollLast();

            q.add(i);

            if(i >= k - 1)
                res[index++] = nums[q.peekFirst()];
        }

        return res;

    }
}
```

剑指 Offer 59 - II. 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

```
class MaxQueue {

    Queue<Integer> queue;
    Deque<Integer> deque; //记录最大值

    public MaxQueue() {
        queue = new LinkedList<>();
        deque = new LinkedList<>();
    }

    public int max_value() {
        return deque.size() > 0 ? deque.peekFirst() : -1;
    }

    public void push_back(int value) {
        queue.add(value);
        while(deque.size() > 0 &&  value > deque.peekLast())
            deque.pollLast();
        deque.add(value);
    }

    public int pop_front() {
        int value = queue.size() > 0 ? queue.poll() : -1 ;
        if(deque.size() > 0 && value == deque.peekFirst())
            deque.pollFirst();
        return value;
    }
}
```

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:
[0.02778,0.05556,0.08333,0.11111,0.13889,0.16667,0.13889,0.11111,0.08333,0.05556,0.027
78]
```

限制:

- 1 <= n <= 11

```
class Solution {
    public double[] dicesProbability(int n) {
        double[] res = new double[1 + n * 5];
        int[][] dp = new int[n + 1 ][6 * n + 1];
```

```
        double all = Math.pow(6,n);
        dp[0][0] = 1;
        for(int i = 1 ; i <= n; i++){
            for(int j = 1; j <= 6*n; j++){
                for(int k = 1; k <= Math.min(j,6); k++)
                    dp[i][j] += dp[i - 1][j - k];
            }
        }

        for(int i = n ; i <= 6 * n ; i++){
            res[i - n ] = dp[n][i] / all;
        }

        return res;
    }
}
```

剑指 Offer 61. 扑克牌中的顺子

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 minNum = 13;
        int maxNum = nums[0];

        HashSet<Integer> set = new HashSet<>();

        for(int i = 0 ; i < nums.length; i++){
            if(nums[i] == 0 ) continue;
            if(minNum > nums[i]) minNum = nums[i];
            if(maxNum < nums[i]) maxNum = nums[i];
            if(!set.add(nums[i])) return false;
        }

        if(maxNum - minNum < 5) return true;
        else return false;
    }

}
```

剑指 Offer 62. 圆圈中最后剩下的数字

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（删除后从下一个数字开始计数）。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

```
class Solution {
    public int lastRemaining(int n, int m) {
        if(n == 1 ) return 0;
        return (lastRemaining(n -1, m) + m) % n;
    }
}
```

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

```
class Solution {
    public int maxProfit(int[] prices) {

        //动态规划
        if(prices == null || prices.length == 0) return 0;

       int maxValue = 0,minValue = prices[0];
```

```
        for(int i = 1 ; i < prices.length; i++){
            maxValue = Math.max(maxValue,prices[i] - minValue);
            minValue = Math.min(minValue,prices[i]);
        }

        return maxValue;

    }
}
```

剑指 Offer 64. 求1+2+...+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

```
class Solution {
    int res = 0;
    public int sumNums(int n) {
        boolean x = n > 1 && sumNums(n-1) > 0;
        res += n;
        return res;

    }
}
```

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
class Solution {
    public int add(int a, int b) {
        while(b != 0) { // 当进位为 0 时跳出
            int c = (a & b) << 1;  // c = 进位
            a ^= b; // a = 非进位和
            b = c; // b = 进位
        }
        return a;
    }
}
```

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[i] 的值是数组 A 中除了下标 i 以外的元素的积, 即 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
class Solution {
    public int[] constructArr(int[] a) {
        //在i的时候避开*a[i],先乘左边，再去乘右边
        int[] res = new int[a.length];

        for(int i = 0 , p = 1; i < a.length ; i ++){
            res[i] = p;
            p *= a[i];
        }

        for(int i = a.length -1 ,p = 1; i >=0 ; i--){
            res[i] *= p;
            p *= a[i];
        }

        return res;
    }
}
```

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 [−231，231 − 1]。如果数值超过这个范围，请返回 INT_MAX (231 − 1) 或 INT_MIN (−231) 。

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 (-231) 。
```

```java
class Solution {
    public int strToInt(String str) {
        char[] arr = str.trim().toCharArray();

        int alertNum = Integer.MAX_VALUE / 10 ; //214748364,防止超出最大整数
        if(arr.length == 0 ) return 0;

        int res = 0,sign = 1;

        int index = 1;
        if(arr[0] == '-') sign = -1;
        else if(arr[0] != '+') index = 0;

        for(int i = index ; i < arr.length; i++){
            if(arr[i] < '0' || arr[i] > '9') break;

            //如果res == 214748364并且arr[i]>'7',这个数就会超过最大整数
```

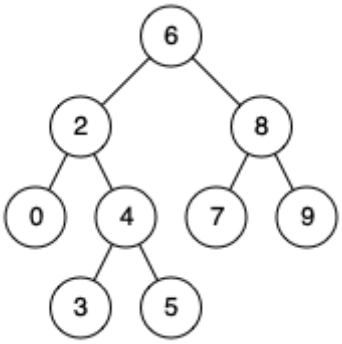
```
            if(res > alertNum || (res == alertNum && arr[i] > '7'))
                return sign == 1 ? Integer.MAX_VALUE:Integer.MIN_VALUE;
            res = res * 10 + (arr[i] - '0');
        }

        return sign* res;

    }
}
```

剑指 Offer 68 - I. 二叉搜索树的最近公共祖先

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。

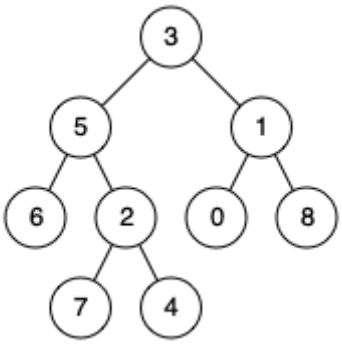
```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root.val < p.val && root.val < q.val)
            return lowestCommonAncestor(root.right, p, q);
        if(root.val > p.val && root.val > q.val)
            return lowestCommonAncestor(root.left, p, q);
        return root;
    }
}
```

剑指 Offer 68 - II. 二叉树的最近公共祖先

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