零、模板

0.0 算法思路总结

0.1 快速排序模板

```
    public static void quickSort(int a[], int low, int hight) {
        int i, j, index;
        if (low > hight) {
            return;
        }
        i = low;
        j = hight;
        index = a[i]; // 用子表的第一个记录做基准
        while (i < j) { // 从表的两端交替向中间扫描
            while (i < j && a[j] >= index)
                j--;
            if (i < j)
                a[i++] = a[j];// 用比基准小的记录替换低位记录
            while (i < j && a[i] < index)
                i++;
            if (i < j) // 用比基准大的记录替换高位记录
                a[j--] = a[i];
        }
        a[i] = index;// 将基准数值替换回 a[i]
        quickSort(a, low, i - 1); // 对低子表进行递归排序
        quickSort(a, i + 1, hight); // 对高子表进行递归排序
    }
```

0.2 动态规划模板

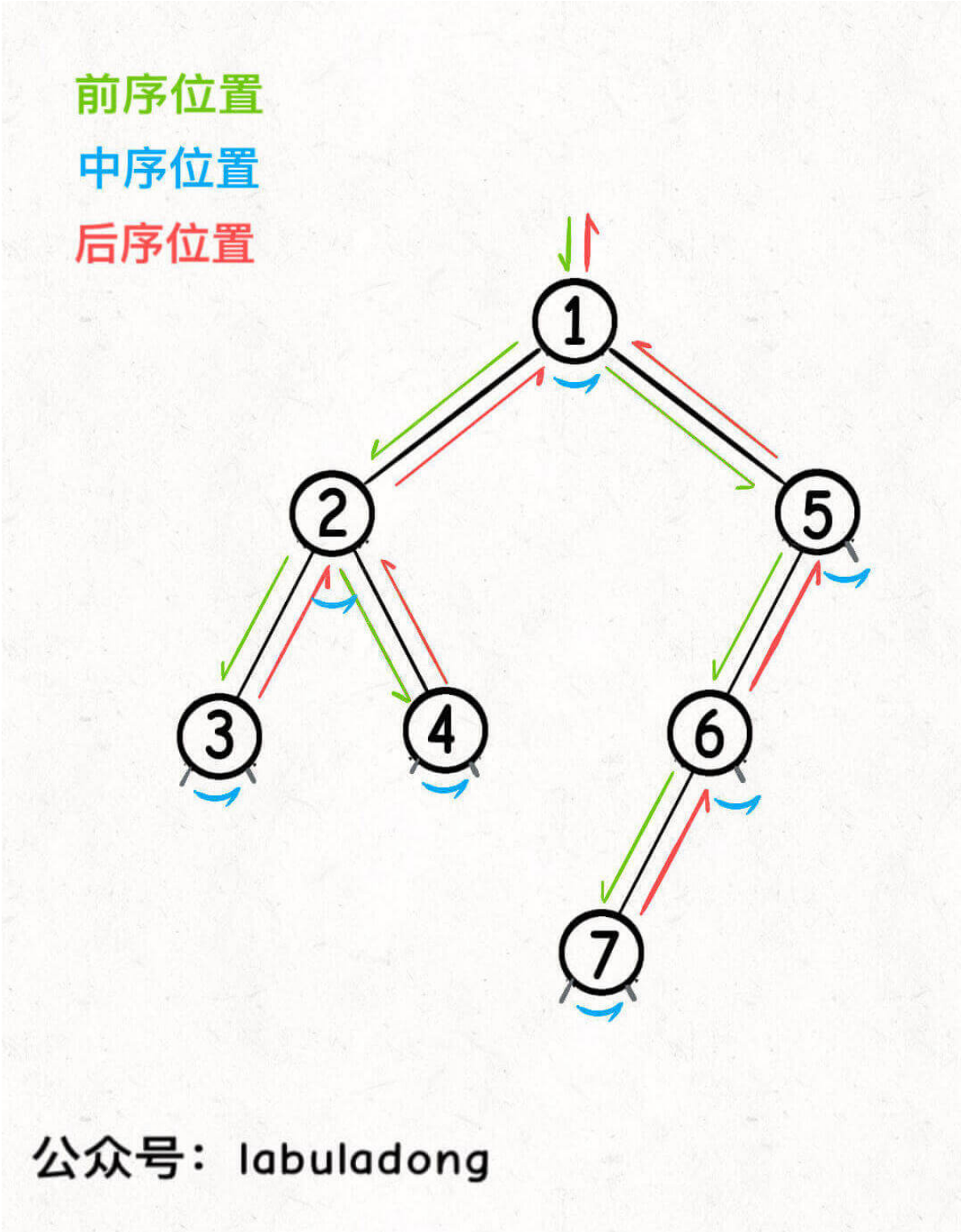
```
# 初始化 base case
dp[0][0][...] = base
# 进行状态转移
for 状态1 in 状态1的所有取值:
    for 状态2 in 状态2的所有取值:
        for ...
            dp[状态1][状态2][...] = 求最值(选择1, 选择2...)
```

0.3 二分查找模板

```
int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;    // 注意

    while(left <= right) {
        int mid = left + (right - left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1;         // 注意
        else if (nums[mid] > target)
            right = mid - 1;        // 注意
    }
    return -1;
}
```

0.4 二叉树遍历模板

```
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序位置
    traverse(root.left);
    // 中序位置
    traverse(root.right);
    // 后序位置
}
```



0.5 回溯算法框架

解决一个回溯问题，实际上就是一个决策树的遍历过程。你只需要思考 3 个问题：

1、路径：也就是已经做出的选择。

2、选择列表：也就是你当前可以做的选择。

3、结束条件：也就是到达决策树底层，无法再做选择的条件。

```
result = []
void backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

其核心就是 for 循环里面的递归，在递归调用之前「做选择」，在递归调用之后「撤销选择」

参考：2.21

0.6 BFS

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路

    q.offer(start); // 将起点加入队列
    visited.add(start);
    int step = 0; // 记录扩散的步数

    while (q not empty) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            /* 划重点：这里判断是否到达终点 */
            if (cur is target)
                return step;
            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj()) {
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
            }
        }
        /* 划重点：更新步数在这里 */
        step++;
    }
}
```

0.7 滑动窗口

```
int left = 0, right = 0;
while (right < s.size()) {
    // 增大窗口
    window.add(s[right]);
    right++;

    while (window needs shrink) {
        // 缩小窗口
        window.remove(s[left]);
        left++;
    }
}
```

0.8 单调栈

```
Deque<Integer> stack = new ArrayDeque<>();  //双端队列代替Stack
for(int i = 0; i < nums.length; i++)
{
    while(!stack.isEmpty() && stack.peek() > nums[i])
    {
        stack.pop();
    }
    stack.push(nums[i]);
}
```

一、剑指offer

1.1 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：
输入：
[2, 3, 1, 0, 2, 5, 3]
输出：2 或 3
```

```
//遍历数组，第一次遇到数字x时，将其交换至索引x处；而当第二次遇到数字x时，一定有 nums[x] = x ，此时即可得到一组重复数字。
class Solution {
    public int findRepeatNumber(int[] nums) {
        int i = 0;
        while(i < nums.length) {
            if(nums[i] == i) {
                i++;
                continue;
```

```
            }
            if(nums[nums[i]] == nums[i]) return nums[i];
            int tmp = nums[i];
            nums[i] = nums[tmp];
            nums[tmp] = tmp;
        }
        return -1;
    }
}
```

1.2 二维数组中查找

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。
```

```
//将矩阵向左旋转45度，可以看作一个二叉搜索树，从右上角或者左下角为根开始搜索；
class Solution {
    public boolean findNumberIn2DArray(int[][] matrix, int target) {
        if(matrix.length == 0 || matrix[0].length == 0)
            return false;
        int i =0 ,j = matrix[0].length-1;
        while(i<matrix.length && j>=0){
            if(target < matrix[i][j]) j--;
            else if(target > matrix[i][j]) i++;
            else return true;
        }
        return false;
    }
}
```

1.3 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:
输入：s = "We are happy."
输出："We%20are%20happy."
```

```java
//Java 语言中，字符串都被设计成「不可变」的类型，即无法直接修改字符串的某一位字符，需要新建一个字符串实现。
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();
    }
}
```

1.4 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）。

解法1：递归

```java
class Solution {
    ArrayList<Integer> tmp = new ArrayList<Integer>();
    public int[] reversePrint(ListNode head) {
        recur(head);
        int[] res = new int[tmp.size()];
        for(int i=0;i<res.length;i++){
            res[i] = tmp.get(i);
        }
        return res;
    }
    public void recur(ListNode head){
        if(head == null) return;
        recur(head.next);
        tmp.add(head.val);
    }
}
```

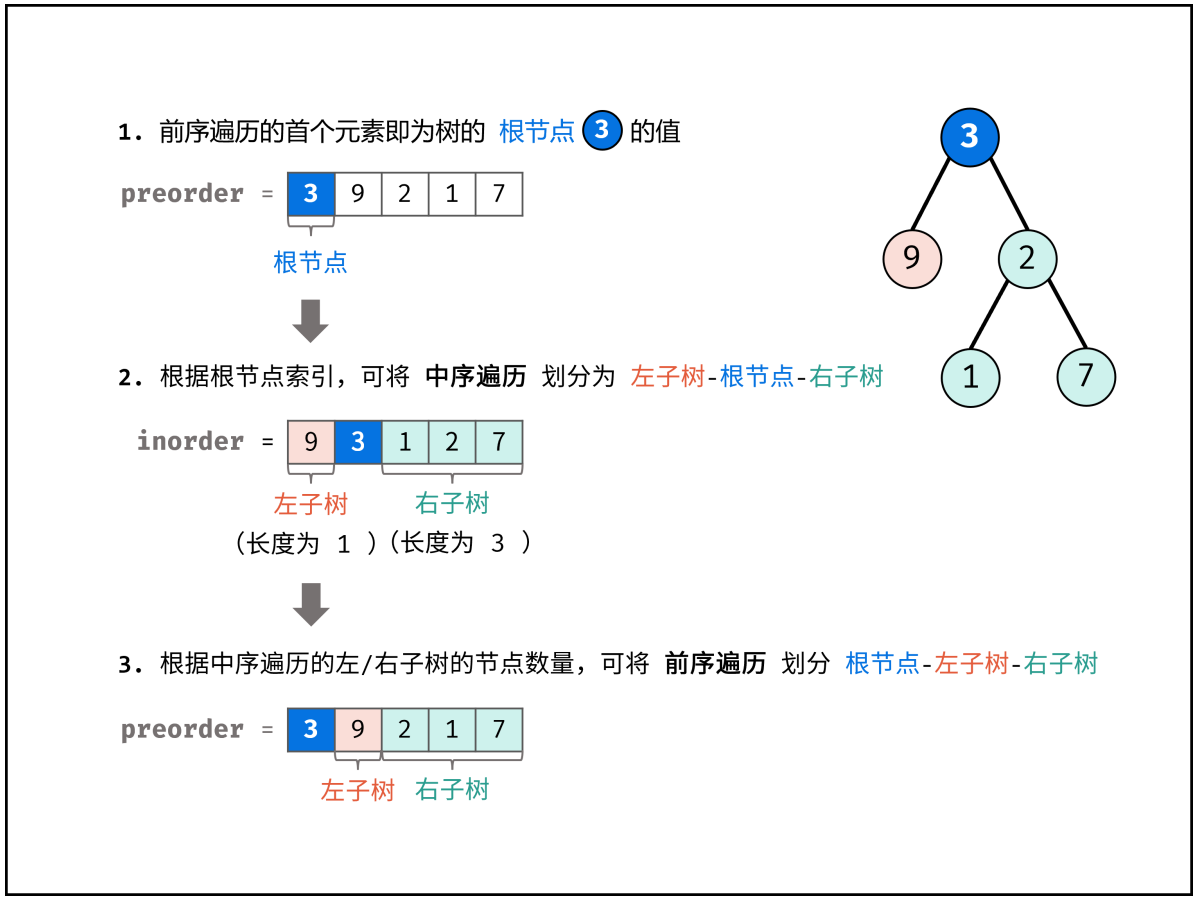
解法2：辅助栈

```java
//遍历链表，将链表值依次入栈，最后创建数组依次出栈接收
class Solution {
    public int[] reversePrint(ListNode head) {
        LinkedList<Integer> stack = new LinkedList<Integer>();
        while(head != null) {
            stack.addLast(head.val);
            head = head.next;
        }
        int[] res = new int[stack.size()];
        for(int i = 0; i < res.length; i++)
            res[i] = stack.removeLast();
    return res;
    }
}
```

1.5 重建二叉树

输入某二叉树的前序遍历和中序遍历的结果，请构建该二叉树并返回其根节点。

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




```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    int[] preorder;
    HashMap<Integer,Integer> dic = new HashMap<>();
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        this.preorder = preorder;
        //将中序遍历的值-索引存入map中，方便递归时获取左子树与右子树的数量及其根的索引
        for(int i=0;i<inorder.length;i++){
            dic.put(inorder[i],i);
        }
        //当前根的的索引，递归树的左边界，递归树的右边界
        return recur(0,0,inorder.length-1);
    }

    TreeNode recur(int pre_root,int in_left,int in_right){
        if(in_left > in_right) return null;
        TreeNode root = new TreeNode(preorder[pre_root]);
        int in_root = dic.get(preorder[pre_root]);  //获取在中序遍历中根节点所在索引
```

```
        //左子树的根的索引为先序中的根节点索引+1
        //递归左子树的左边界为原来的中序in_left
        //递归左子树的右边界为中序中的根节点索引-1
        root.left = recur(pre_root+1,in_left,in_root-1);
        //右子树的根的索引为先序中的 （当前根索引 + 左子树的数量 + 1）
        //递归右子树的左边界为中序中当前根节点索引+1
        //递归右子树的右边界为中序中原来右子树的边界
        root.right = recur(pre_root+(in_root-in_left)+1,in_root+1,in_right);
        return root;
    }
}
```

1.6 双栈实现队列

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)

```
//双栈实现队列，一个栈存储队尾插入元素，一个栈反向第一个栈，用于对头删除操作（反向栈出栈一个元素即可）
class CQueue {
    LinkedList<Integer> stack,rStack;
    public CQueue() {
        stack = new LinkedList<Integer>();
        rStack = new LinkedList<Integer>();
    }

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

    public int deleteHead() {
        if(!rStack.isEmpty()) return rStack.pop();
        if(stack.isEmpty()) return -1;
        while(!stack.isEmpty()){
            rStack.push(stack.pop());
        }
        return rStack.pop();
    }
}
```

1.7 斐波那契数列（动态规划）

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

```java
//动态规划
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;   //精华
    }
}
```

1.8 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。

```java
//动态规划，就是斐波那契数列；
//跳1个台阶有1种方法，跳2个台阶有2种方法（可以看作跳两个1级台阶），跳3个台阶有三种方法（可以看作1+2）即前两个台阶种数相加；
class Solution {
    public int numWays(int n) {
        int a=1,b=1,sum;        //a b sum
        for(int i=0;i<n;i++){
            sum = (a + b) % 1000000007;
            a = b;
            b = sum;
        }
        return a;
    }
}
```

1.9 旋转数组最小数字

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。

给你一个可能存在 重复 元素值的数组 numbers ，它原来是一个升序排列的数组，并按上述情形进行了一次旋转。请返回旋转数组的最小元素。例如，数组 [3,4,5,1,2] 为 [1,2,3,4,5] 的一次旋转，该数组的最小值为1。

```
示例 ：
输入：[3,4,5,1,2]
输出：1
```

```java
//旋转数组，二分法；
//先使用二分查找，注意升序时，将mid与右边界比较才能确定零界点位置；
class Solution {
    public int minArray(int[] numbers) {
        int i=0,j=numbers.length-1,mid;
```

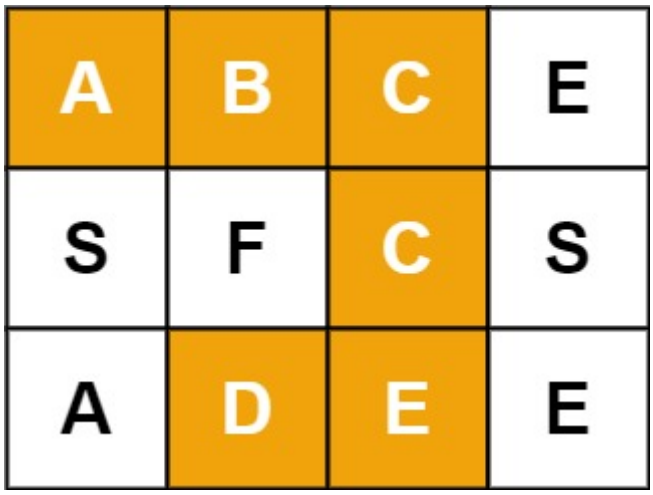
```
        while(i < j){
            mid = (i + j) / 2;
            if(numbers[mid] < numbers[j]) j = mid;           //零界点肯定在mid左边
            else if(numbers[mid] > numbers[j]) i = mid + 1; //零界点在mid右边
            //如果mid == j，那么numbers[i]~numbers[mid]或者numbers[mid]~numbers[j]
数字相同，这时直接遍历查找
            else{
                for(int t=i;t<j;t++){
                    if(numbers[t] > numbers[t+1]) return numbers[t+1];
                }
                return numbers[i];
            }
        }
        return numbers[i];
    }
}
```

1.10 矩阵中的路径（DFS）

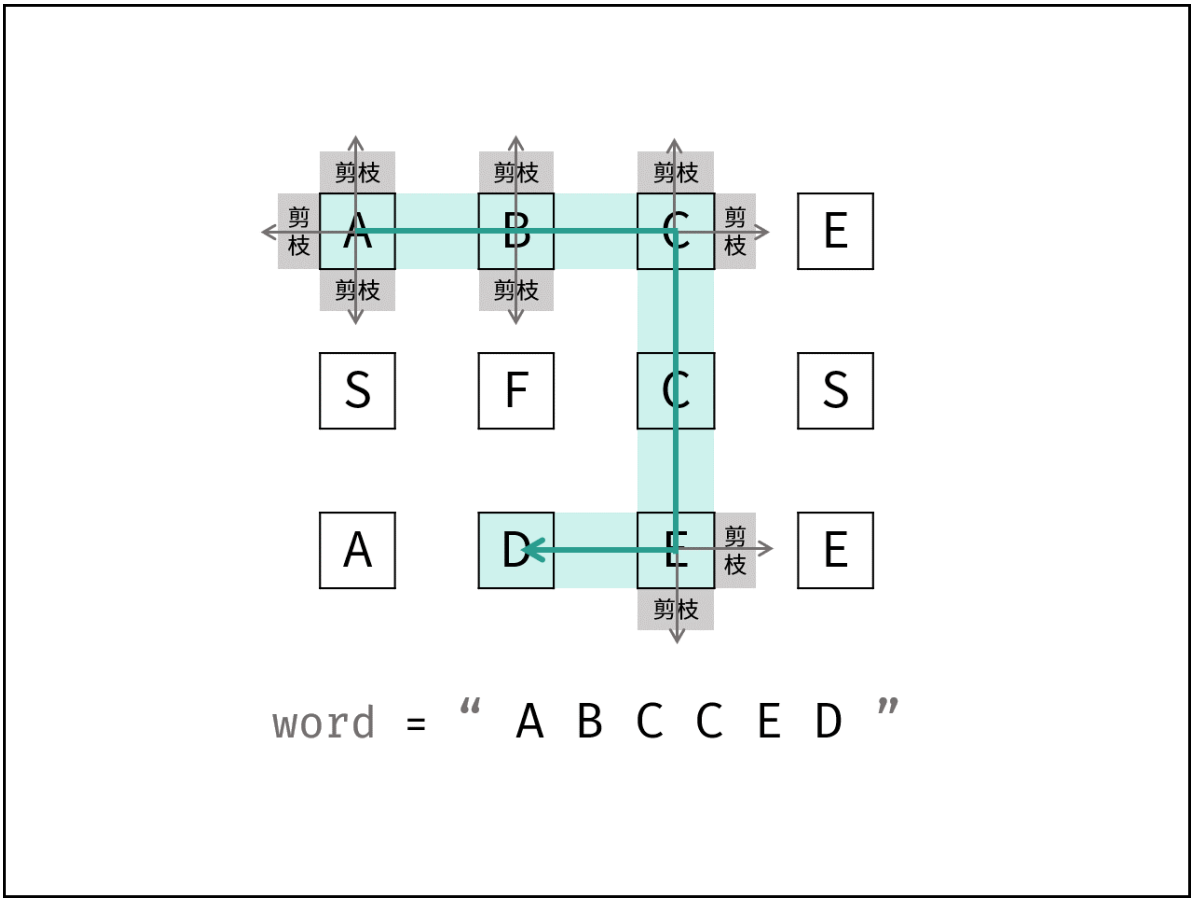
给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true ；否则，返回 false 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

例如，在下面的 3×4 的矩阵中包含单词 "ABCCED"（单词中的字母已标出）。




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




```
//图的DFS+剪枝
//利用递归的思想，首先遍历图查找与字符串第一个字符匹配的位置，再依次朝4个方向搜索（其中越界、已访问过等的方向需要剪枝）；
class Solution {
    public boolean exist(char[][] board, String word) {
        char[] words = word.toCharArray();
        for(int i = 0; i < board.length; i++) {// 遍历图
            for(int j = 0; j < board[0].length; j++) {
                if(dfs(board, words, i, j, 0)) return true;// 如果找到了，就返回true
            }
        }
        return false;
    }

    boolean dfs(char[][] board, char[] word, int i, int j, int k) {
        // i:行，j:列， k是传入字符串当前索引
        //先剪枝，再匹配字符串第一个字符位置
        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;    //字符串匹配成功
        board[i][j] = '\0';                      //标记访问过的字符，防止重复访问
        //已经找到了第一个字符对应位置，索引字符串索引k+1，再朝4个方向DFS
        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];// 还原找过的元素，因为之后可能还会访问到（不同路径）
        return res;
    }
}
```

1.11 机器人运动范围

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：
输入：m = 2, n = 3, k = 1
输出：3

提示：
1 <= n,m <= 100
0 <= k <= 20
```

```
//DFS
//仅需向右和向下搜索即可访问所有可达解（等腰三角形）
class Solution {
    public int movingCount(int m, int n, int k) {
        boolean[][] visited = new boolean[m][n]; //标识单元格已被访问
        return dfs(m,n,k,0,0,visited);
    }

    //i:行坐标,j:纵坐标
    int dfs(int m,int n,int k,int i,int j,boolean[][] visited){
        if(i >= m || j >= n || k < sum(i) + sum(j) || visited[i][j]) return 0;
        visited[i][j] = true;
        return 1 + dfs(m,n,k,i+1,j,visited) + dfs(m,n,k,i,j+1,visited);//本身自己
的1格加上朝右和朝下搜索格数;
    }

    //计算位数和
    int sum(int x){
        int t = x % 10;
        x /= 10;
        return t + x;
    }
}
```

1.12 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：
输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36

2 <= n <= 58
```

```java
//动态规划（dp）
class Solution {
    public int cuttingRope(int n) {
        //dp[i]存储长度为i的绳子剪成m端后长度的最大乘积
        int dp[] = new int[n+1];
        dp[2] = 1;  //长度为2时已经知道最大乘积为1
        for(int i = 3;i <= n;i++){  //dp[2]已知，从dp[3]开始求
            //首先对绳子剪长度为j的一段
            for(int j = 2;j < i;j++){
                //比较:dp[i]:当前长度不剪
                //j*(i-j):只剪了j长度的一段
                //j*dp[i-j]:剪了j长度一段并且j可能继续剪
                dp[i] = Math.max(dp[i],Math.max(j*(i-j),j*dp[i-j]));
            }
        }
        //现在已经求出每个长度i对应的最大乘积，返回dp[n]
        return dp[n];
    }
}
```

```java
//贪心算法
//数学推导证明，当n大于4时，尽可能多的剪出长度为3的段，求出的乘积最大；
class Solution {
    public int cuttingRope(int n) {
        if(n == 2)return 1;
        if(n == 3)return 2;
        if(n == 4)return 4;
        int res = 1;
        while(n > 4){
            res *= 3;
            n -= 3;
        }
        return res*n;
    }
}
```

1.13 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。

2 <= n <= 1000`

```java
//只能使用贪心算法
//因为n较大，所以直接使用贪心算法，大数求余法：循环求余法；
class Solution {
    public int cuttingRope(int n) {
        if(n == 2) return 1;
        if(n == 3) return 2;
        if(n == 4) return 4;
        long res = 1;    //必须定义为long，否则会溢出；
```

```
        while(n > 4){
            res = res * 3 % 1000000007;
            n -= 3;
        }
        return (int)(res * n % 1000000007);
    }
}
```

1.14 二进制中1的个数

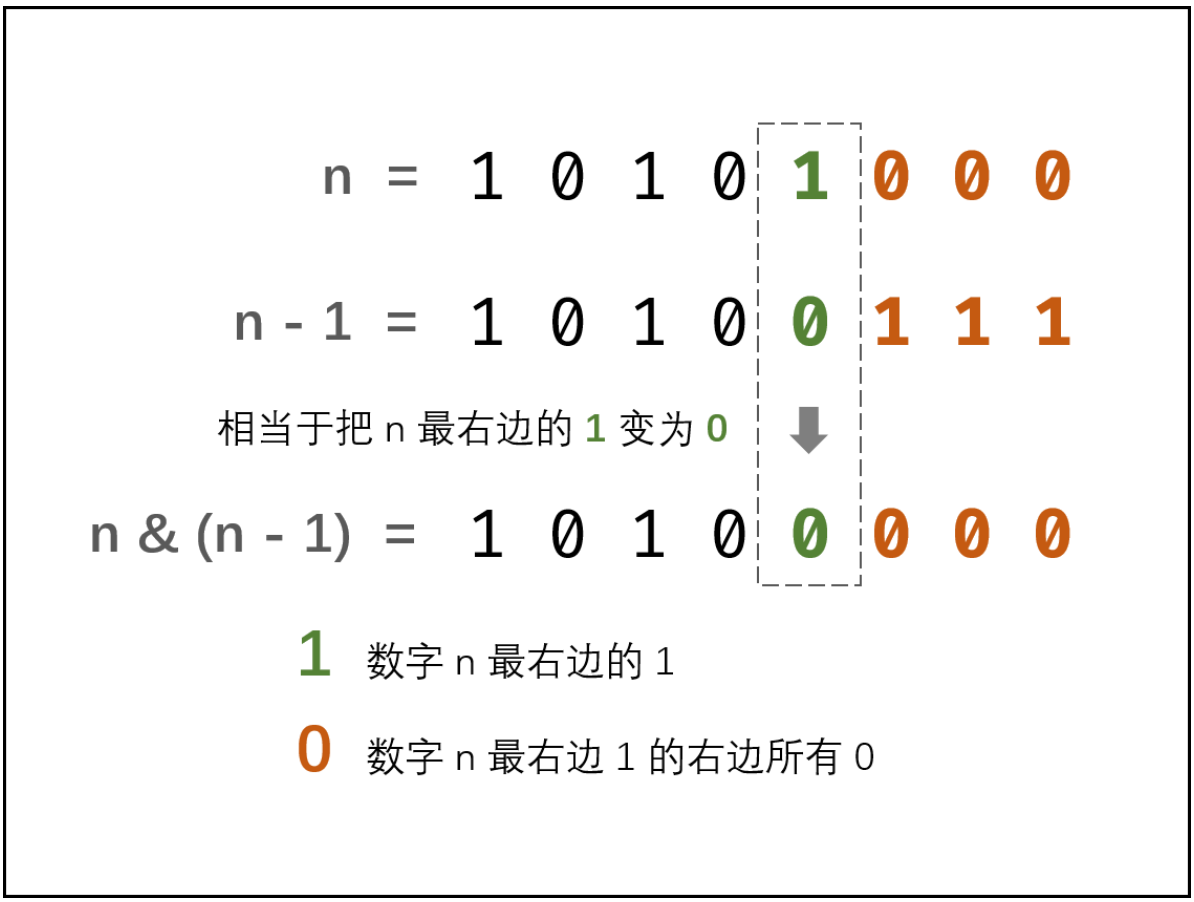
编写一个函数，输入是一个无符号整数（以二进制串的形式），返回其二进制表达式中数字位数为 '1' 的个数（也被称为 汉明重量).）。

提示：

请注意，在某些语言（如 Java）中，没有无符号整数类型。在这种情况下，输入和输出都将被指定为有符号整数类型，并且不应影响您的实现，因为无论整数是有符号的还是无符号的，其内部的二进制表示形式都是相同的。
在 Java 中，编译器使用 二进制补码 记法来表示有符号整数。因此，在上面的 示例 3 中，输入表示有符号整数 -3。

```
示例：
输入：n = 11 (控制台输入 00000000000000000000000000001011)
输出：3
解释：输入的二进制串 00000000000000000000000000001011 中，共有三位为 '1'。
```

```
//第一种，逐位判断
//将n与1相与，每次向右移一位；
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int res = 0;
        while(n != 0){  //条件判断为 == 0；
            res += n & 1;
            n = n >>> 1;//(Java无符号移位>>>)
        }
        return res;
    }
}
```




```
//第二种，n&(n-1)
//n&(n - 1)其预算结果恰为把n的二进制位中的最低位的1变为0之后的结果。
public class Solution {
    // you need to treat n as an unsigned value
    public int hammingWeight(int n) {
        int res = 0;
        while(n != 0){
            res++;
            n &= n - 1;
        }
        return res;
    }
}
```

1.15 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。

```
//二分思想快速求幂法
//将x^n分为x^n/2 * x^n/2 循环相乘，判断奇数时再乘一个x；
//关键点：x =* x;而非res *= x;
class Solution {
    public double myPow(double x, int n) {
        if(n == 0) return 1.0;  //n为0时直接返回1.0
        double res = 1.0;
        long t = n;             //防止溢出
        if(t < 0){              //n<0时转换为正数计算
            t = -t;
            x = 1 / x;
```

```
        }
        while(t > 0){
            if((t & 1) == 1){    //<=> t%2,奇数时，需要再乘一个x
                res *= x;
            }
            x *= x;              //<=>x^2
            t >>= 1;             //<=>t向下整除2
        }
        return res;
    }
}
```

1.16 打印1到最大的n位数（考虑大数）

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。

```
//1.为了避免数字开头出现0，先把首位first固定，first取值范围为1~9
//2.用digit表示要生成的数字的位数，本题要从1位数一直生成到n位数，对每种数字的位数都生成一下首位，所以有个双重for循环
//3.生成首位之后进入递归生成剩下的digit - 1位数，从0~9中取值
//4.递归的中止条件为已经生成了digit位的数字，即index == digit，将此时的数num转为int加到结果res中
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);
        }
    }
}
```

1.17 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。

```
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

```
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.val != val && cur.next != null)  //遍历链表，知道找到val
            cur = cur.next;
        cur.next = cur.next.next;                       //单向链表，只需要改动尾指针
        return head;
    }
}
```

1.18 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: falses 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
//DP，有点难
//逐个匹配字符，分为空正则和非空正则，再分为当前字符为*或者不为*，为*时分为看*和不看*
class Solution {
    public boolean isMatch(String A, String B) {
        int n = A.length();
        int m = B.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 (B.charAt(j - 1) != '*') {   //注意下标位置
                        if (i > 0 && (A.charAt(i - 1) == B.charAt(j - 1) ||
B.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 && (A.charAt(i - 1) == B.charAt(j -
2) || B.charAt(j - 2) == '.')) {
                            f[i][j] |= f[i - 1][j];
                        }
                    }
                }
            }
        }
        return f[n][m];
    }
}
```

1.19 表示数值的字符串（有限状态机）

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）。

数值（按顺序）可以分成以下几个部分：

若干空格
一个 小数 或者 整数
（可选）一个 'e' 或 'E' ，后面跟着一个 整数
若干空格
小数（按顺序）可以分成以下几个部分：

（可选）一个符号字符（'+' 或 '-'）
下述格式之一：
至少一位数字，后面跟着一个点 '.'
至少一位数字，后面跟着一个点 '.' ，后面再跟着至少一位数字
一个点 '.' ，后面跟着至少一位数字
整数（按顺序）可以分成以下几个部分：

（可选）一个符号字符（'+' 或 '-'）
至少一位数字
部分数值列举如下：

["+100", "5e2", "-123", "3.1416", "-1E-16", "0123"]
部分非数值列举如下：

["12e", "1a3.14", "1.2.3", "+-5", "12e+5.4"]

```
示例 1：
输入：s = "0"
输出：true

示例 2：
输入：s = "e"
输出：false

示例 3：
输入：s = "."
输出：false

示例 4：
输入：s = "    .1  "
输出：true
```

提示：

1 <= s.length <= 20
s 仅含英文字母（大写和小写），数字（0-9），加号 '+' ，减号 '-' ，空格 ' ' 或者点 '.' 。

```
//有限状态机（编译原理）
//小数表示可省去0，-0.4 = -.4，0.4 = .4；2.、3. = 2、3，小数点前有数，后面可以不跟数代表原数
//注意e8即10的8次幂（8次方），也可以是e-7，但题目要求必须跟整数
```

```java
//题目规定是数值前后可有空格，中间不能有，这个情况要考虑清楚。s：符号、d：数字
class Solution {
    public boolean isNumber(String s) {
        Map[] states = {
            //0：规定0是初值，字符串表示数值，有4种起始状态，开头空格、符号、数字、前面没有数的小数点
            //其中 开头空格 还是指向states[0]，上一位是 开头空格，下一位可以是 空格、符号、数字、前面没有数的小数点
            new HashMap<>() {{ put(' ', 0); put('s', 1); put('d', 2); put('.', 4); }},
            //1：上一位是符号，符号位后面可以是 数字、前面没有数的小数点
            new HashMap<>() {{ put('d', 2); put('.', 4); }},
            //2：上一位是数字，数字的下一位可以是 数字、前面有数的小数点、e、结尾空格
            new HashMap<>() {{ put('d', 2); put('.', 3); put('e', 5); put(' ', 8); }},
            //3：上一位是前面有数的小数点，下一位可以是 数字、e（8.e2 = 8e2，和2的情况一样）、结尾空格
            new HashMap<>() {{ put('d', 3); put('e', 5); put(' ', 8); }},
            //4：上一位是前面没有数的小数点，下一位只能是 数字（符号肯定不行，e得前面有数才行）
            new HashMap<>() {{ put('d', 3); }},
            //5：上一位是e，下一位可以是 符号、数字
            new HashMap<>() {{ put('s', 6); put('d', 7); }},
            //6：：上一位是e后面的符号，下一位只能是 数字
            new HashMap<>() {{ put('d', 7); }},
            //7：上一位是e后面的数字，下一位可以是 数字、结尾空格
            new HashMap<>() {{ put('d', 7); put(' ', 8); }},
            //8：上一位是结尾空格，下一位只能是 结尾空格
            new HashMap<>() {{ put(' ', 8); }}
        };
        int p = 0;
        char t;
        //遍历字符串，每个字符匹配对应属性并用t标记，非法字符标记？
        for(char c : s.toCharArray()) {
            if(c >= '0' && c <= '9') t = 'd';
            else if(c == '+' || c == '-') t = 's';
            else if(c == 'e' || c == 'E') t = 'e';
            else if(c == '.' || c == ' ') t = c;
            else t = '?';
            //当前字符标记和任何一种当前规定格式都不匹配，直接返回false
            if(!states[p].containsKey(t)) return false;
            //更新当前字符的规定格式，进入下一个规定的Map数组
            p = (int)states[p].get(t);
        }
        //2（正、负整数）、3（正、负小数）、7（科学计数法）、8（前三种形式的结尾加上空格）
        //只有这四种才是正确的结尾
        return p == 2 || p == 3 || p == 7 || p == 8;
    }
}
```

1.20 调整数组奇偶数

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，使得所有奇数在数组的前半部分，所有偶数在数组的后半部分。

```
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。
```

```
//双向指针，前后寻找奇偶数，然后交换
class Solution {
    public int[] exchange(int[] nums) {
        int i=0,j=nums.length-1,t;
        while(i < j){
            while(i < j && (nums[i] & 1) == 1) i++;
            while(i < j && (nums[j] & 1) == 0) j--;
            t = nums[i];
            nums[i] = nums[j];
            nums[j] = t;
        }
        return nums;
    }
}
```

1.21 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.
```

```
//双指针
//前指针（f）和后指针（l）相距k，当f到达链表尾部时，l即为倒数第k个节点的位置；
class Solution {
    public ListNode getKthFromEnd(ListNode head, int k) {
        ListNode f=head,l=head;
        if(head == null) return null;
        for(int i=0;i<k;i++)
            f = f.next;
        while(f != null){
            f = f.next;
            l = l.next;
        }
        return l;
    }
}
```

1.22 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
```

```
//双指针迭代
//遍历每次将当前节点（cur）的next指向前一个节点（pre），需要用一个tmp保存当前节点的下一个节点；
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode pre = null,cur = head,tmp;
        while(cur != null){
            tmp = cur.next;
            cur.next = pre;
            pre = cur;
            cur = tmp;
        }
        return pre;
    }
}
```

```
//递归
//先递归到最后一个null节点，返回上一个节点（尾节点），回溯时改变next指向上一个节点；
//pre：前一个节点，即在cur左边一个节点
class Solution {
    public ListNode reverseList(ListNode head) {
        return recur(head,null);
    }

    ListNode recur(ListNode cur,ListNode pre){
        if(cur == null) return pre;
        ListNode res = recur(cur.next,cur);
        cur.next = pre;
        return res;
    }
}
```

1.23 合并两个排序链表

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
```

```
//建立一个新的头节点，循环比较两个链表节点值，插入新链表中
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
```

```
        ListNode nHead = new ListNode(0),cur = nHead;
        while(l1 != null && l2 != null){
            if(l1.val < l2.val){
                cur.next = l1;
                l1 = l1.next;
            }else{
                cur.next = l2;
                l2 = l2.next;
            }
            cur = cur.next;
        }
        cur.next = l1 != null ? l1 : l2;     //判断最后一个节点
        return nHead.next;                   //新建的头节点为null
    }
}
```

1.24 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。

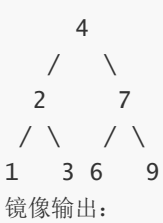
```
示例 1:
输入：A = [1,2,3], B = [3,1]
输出：false

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

```
//先序遍历树A中每个节点（isSubStructure），判断以该节点为根节点是否包含树B（recur）
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;                                    //B遍历完证明匹配成
功
        //当A和B节点值不同时直接返回false，A遍历完且B未遍历完证明匹配失败，位置不能变，否则空
指针
        if(A == null || A.val != B.val) return false;
        return recur(A.left,B.left) && recur(A.right,B.right);
    }
}
```

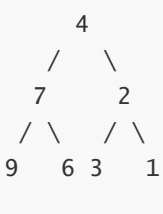
1.25 二叉树的镜像

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：



镜像输出：



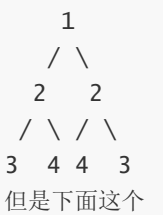
示例 1：
输入：root = [4,2,7,1,3,6,9]
输出：[4,7,2,9,6,3,1]

```
//递归
class Solution {
    public TreeNode mirrorTree(TreeNode root) {
        if(root == null) return null;        //根节点越过子节点
        TreeNode tmp = root.left;            //暂存左子节点
        root.left = mirrorTree(root.right); //递归右子节点
        root.right = mirrorTree(tmp);        //递归左子节点
        return root;
    }
}
```

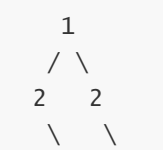
1.26 对称的二叉树

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。



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



```
   3    3


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

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

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

1.27 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:
输入: matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出: [1,2,3,6,9,8,7,4,5]
```

```
//模拟法
//t,r,b,l四个方位边界下标，注意细节
class Solution {
    public int[] spiralOrder(int[][] matrix) {
        if(matrix.length == 0) return new int[0];
        int t = 0,r = matrix[0].length - 1,b = matrix.length - 1,l = 0,ind = 0;
        int[] res = new int[(r+1) * (b+1)];
        while(true){
            for(int i=l;i<=r;i++) res[ind++] = matrix[t][i];        //left ->
right
            if(++t > b) break;
            for(int i=t;i<=b;i++) res[ind++] = matrix[i][r];        //top ->
bottom
            if(--r < l) break;
            for(int i=r;i>=l;i--) res[ind++] = matrix[b][i];        //right ->
left
            if(--b < t) break;
            for(int i=b;i>=t;i--) res[ind++] = matrix[i][l];        //bottom ->
top
            if(++l > r) break;
        }
        return res;
    }
}
```

1.28 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)。

```
//辅助栈
//A用于存储，B始终存储push进来比栈顶小（或等于）的元素；
class MinStack {
    Stack<Integer> A,B;
    /** initialize your data structure here. */
    public MinStack() {
        A = new Stack<>();
        B = new Stack<>();
    }

    public void push(int x) {
        A.push(x);
        if(B.empty() || B.peek() >= x)
            B.push(x);
    }

    public void pop() {
        if(B.peek().equals(A.pop()))    //记住不能使用==
            B.pop();
    }

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

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

1.29 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：
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
```

```
//用栈模拟
//遍历pushed数组，依次压栈，循环判断 “栈顶元素 == 弹出序列的当前元素” 是否成立，将符合弹出序列顺序的栈顶元素全部弹出。
class Solution {
```

```
    public boolean validateStackSequences(int[] pushed, int[] popped) {
        Stack<Integer> stack = new Stack<>();
        int i = 0;
        for(int x : pushed){
            stack.push(x);
            while(!stack.isEmpty() && stack.peek() == popped[i]){
                stack.pop();
                i++;
            }
        }
        return stack.isEmpty();     //如果栈元素全部弹出则为真
    }
}
```

1.30 从上到下打印二叉树

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]
]
```

```
//广度优先（BFS）
//建立一个辅助队列，每层节点加入队列，再循环出队加入值并加入子节点；
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<>();
            //精髓，不能写成for(int i=0;i<queue.size();i++),因为每次循环queue长度在变
            for(int i=queue.size();i>0;i--){
                TreeNode tNode = queue.poll();
                tmp.add(tNode.val);
                if(tNode.left != null) queue.add(tNode.left);
                if(tNode.right != null) queue.add(tNode.right);
            }
            res.add(tmp);
        }
        return res;
    }
}
```

1.31 从上到下打印二叉树II

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]
]
```

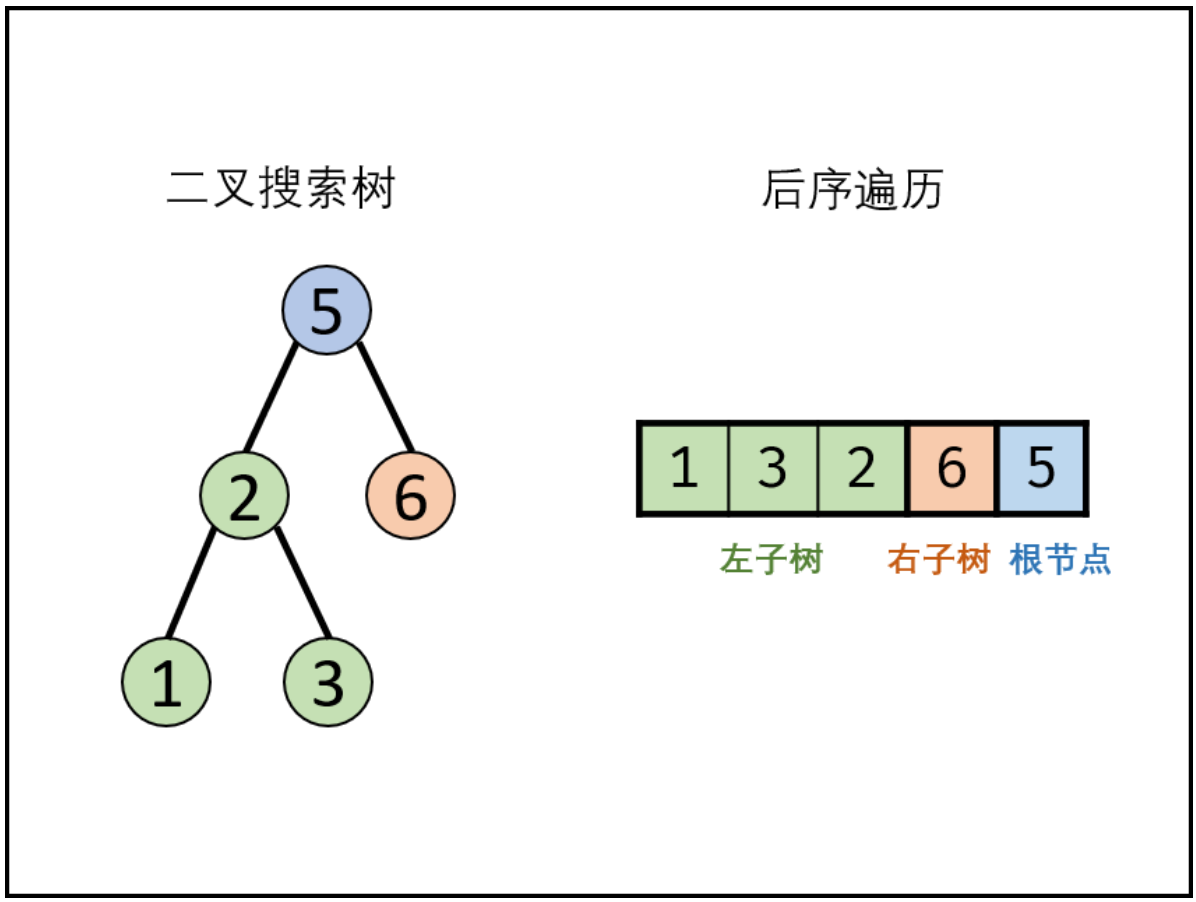
```
//方法同Ⅰ，tmp改为LinkedList，方便头尾插入，使用res.size() % 2判断奇偶层；
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        if(root != null) queue.add(root);
        while(!queue.isEmpty()){
            LinkedList<Integer> tmp = new LinkedList<>();
            for(int i = queue.size();i > 0 ;i--){
                TreeNode tNode = queue.poll();
                if(res.size() % 2 == 0){    //偶，插入尾
                    tmp.addLast(tNode.val);
                }else{                      //奇，插入头
                    tmp.addFirst(tNode.val);
                }
                if(tNode.left != null) queue.add(tNode.left);
                if(tNode.right != null) queue.add(tNode.right);
            }
            res.add(tmp);
        }
        return res;
    }
}
```

1.32 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
```




```
/*
划分左右子树： 遍历后序遍历的[left, right]区间元素，寻找 第一个大于根节点 的节点，索引记为mid
此时，可划分出左子树区间 [left,mid-1] 、右子树区间 [mid, right- ]、根节点索引right。
判断是否为二叉搜索树：
左子树区间 [left, mid- 1]内的所有节点都应 < postorder[right]。
右子树区间 [mid, right-1]内的所有节点都应 > postorder[right] 。
t == right:判断此树是否正确：
*/

class Solution {
    public boolean verifyPostorder(int[] postorder) {
        return recur(postorder,0,postorder.length-1);
    }

    boolean recur(int[] postorder,int left,int right){
        if(left >= right) return true;
        int t = left;
```

```
        while(postorder[t] < postorder[right]) t++;
        int mid = t;
        while(postorder[t] > postorder[right]) t++;
        return t == right && recur(postorder,left,mid-1) &&
recur(postorder,mid,right-1);
    }
}
```

1.33 二叉树中和为某一值的路径

给你二叉树的根节点 root 和一个整数目标和 targetSum ， 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。

叶子节点 是指没有子节点的节点。

```
//DFS
//先序遍历，记录从根节点到当前节点的路径，比较target值
class Solution {
    LinkedList<Integer> path = new LinkedList<>();
    LinkedList<List<Integer>> res = new LinkedList<>();
    public List<List<Integer>> pathSum(TreeNode root, int target) {
        recur(root,target);
        return res;
    }

    void recur(TreeNode root,int target){
        if(root == null) return;
        path.add(root.val);
        target -= root.val;
        if(target == 0 && root.left == null && root.right == null)
            res.add(new LinkedList(path));  //需要new一个新的path，否则path加入res中
也会随时变化
        recur(root.left,target);
        recur(root.right,target);
        path.removeLast();          //从path移除当前节点
    }
}
```

1.34 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]]
```

```
//本题难点在于新建链表时random指向节点无法确定
//使用一个HashMap存储节点，构建random时直接去HashMap中查找；
class Solution {
    public Node copyRandomList(Node head) {
        if(head == null) return null;
```

```
        Node cur = head;
        Map<Node, Node> map = new HashMap<>();
        // 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;
        // 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;
        }
        // 返回新链表的头节点
        return map.get(head);
    }
}
```

1.35 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。

```
//dfs
//根据二叉搜索树的中序遍历为递增序列，在中序遍历时更改左右指针。
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) head = cur; //已经到达头节点
        else pre.right = cur;       //建立指针
        cur.left = pre;             //可以与上面语句交换或者合并
        pre = cur;                  //记录前驱节点
        dfs(cur.right);
    }
}
```

1.36 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。

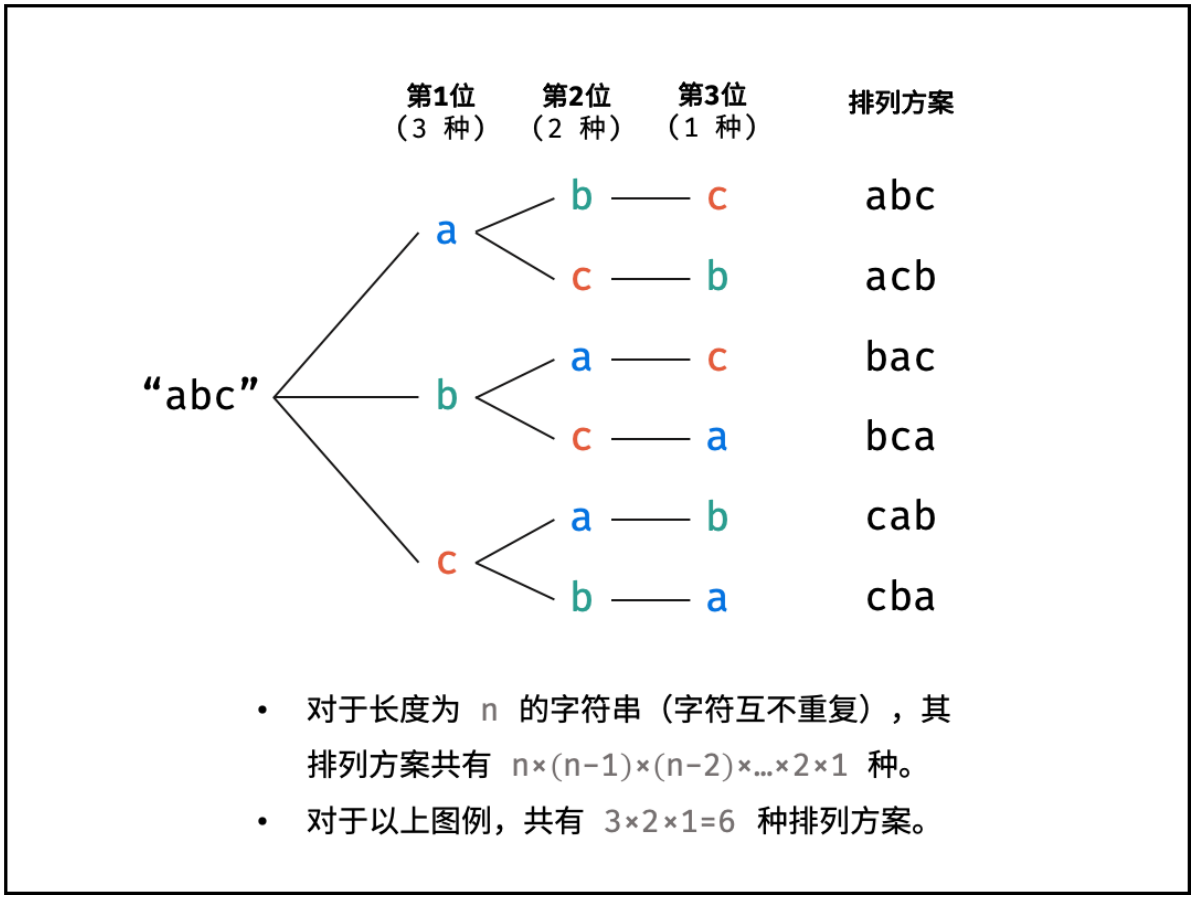
你需要设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

```
//BFS
//DFS、前序、中序、后序都不能反序列化，使用层次遍历，并且保存null节点即可
public class Codec {

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        if(root == null) return "[]";
        Queue<TreeNode> queue = new LinkedList<>();
        StringBuilder res = new StringBuilder("[");
        queue.add(root);
        while(!queue.isEmpty()){
            TreeNode tNode = queue.poll();
            if(tNode != null){
                res.append(tNode.val + ",");
                queue.add(tNode.left);
                queue.add(tNode.right);
            }else{
                res.append("null,");
            }
        }
        res.deleteCharAt(res.length() - 1); //去掉最后一个逗号
        res.append("]");
        return res.toString();
    }

    // Decodes your encoded data to tree.
    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;
    }
}
```

1.37 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"]
```



- 对于长度为 n 的字符串（字符互不重复），其排列方案共有 n×(n−1)×(n−2)×…×2×1 种。
- 对于以上图例，共有 3×2×1=6 种排列方案。

```
//回溯法
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
        }
```

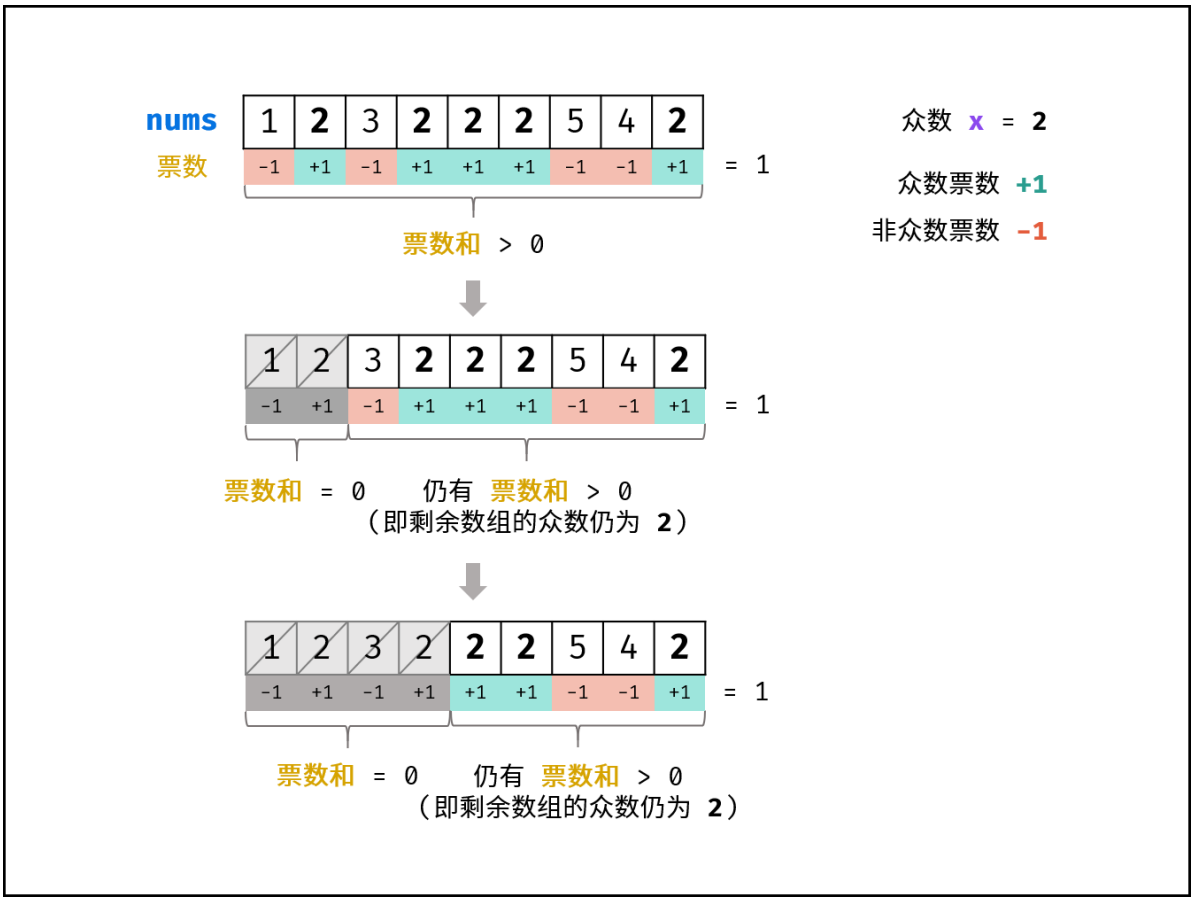
```
    }
    void swap(int a, int b) {
        char tmp = c[a];
        c[a] = c[b];
        c[b] = tmp;
    }
}
```

1.38 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
```




```
//摩尔投票法

class Solution {
    public int majorityElement(int[] nums) {
        int votes = 0,x = 0;
        for(int e:nums){
            if(votes == 0) x = e;
            if(e == x) votes++;
            else votes--;
        }
        return x;
```

```
    }
}
//如果给出数组不一定存在多数元素，需要再遍历数组，记录x的次数与数组长度一半比较。
```

1.39 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]
```

```
//基于快排的思想
//考虑在每次哨兵划分后，判断基准数在数组中的索引是否等于k ，若true则直接返回此时数组的前k个数字即可。
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        if(k >= arr.length) return arr;
        return quickSort(arr,k,0,arr.length-1);
    }

    int[] quickSort(int[] arr,int k,int l,int r){
        int i = l,j = r;
        while(i < j){
            while(i < j && arr[j] >= arr[l]) j--;
            while(i < j && arr[i] <= arr[l]) i++;
            swap(arr,i,j);
        }
        swap(arr,i,l);
        if (i > k) return quickSort(arr, k, l, i - 1);
        if (i < k) return quickSort(arr, k, i + 1, r);
        return Arrays.copyOf(arr, k);
    }

    void swap(int[] arr,int i,int j){
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}
```

1.40 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

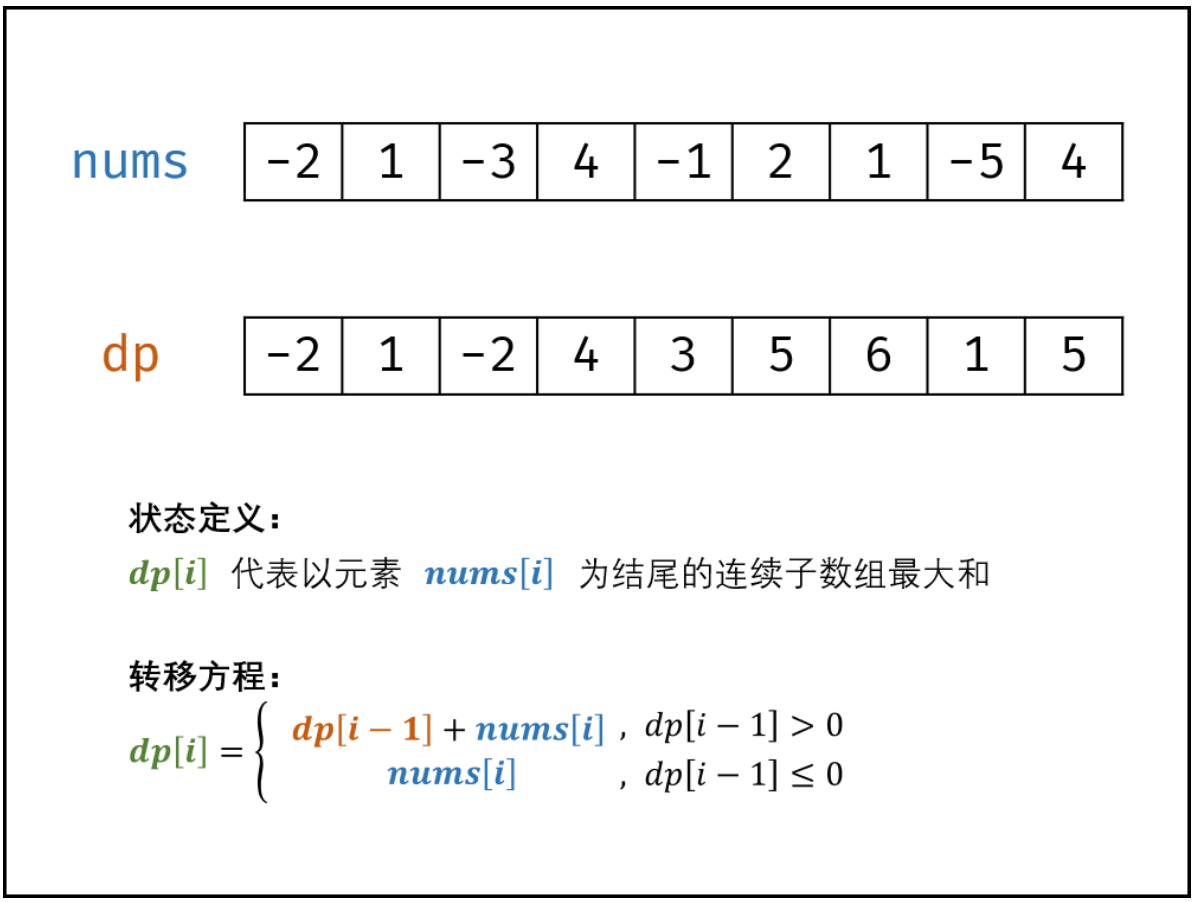
```
//堆
//设置大顶堆和小顶堆，每次依次加入堆中，数据流为奇数时，中位数为小顶堆堆顶，否则两个堆顶平均数。
class MedianFinder {
    Queue<Integer> A,B;
    /** initialize your data structure here. */
    public MedianFinder() {
        A = new PriorityQueue<>();                    // 小顶堆，保存较大的一半
        B = new PriorityQueue<>((x, y) -> (y - x)); // 大顶堆，保存较小的一半
    }

    public void addNum(int num) {
        if(A.size() != B.size()){
            A.add(num);              //循环加入，保证大顶堆中数小于小顶堆中数
            B.add(A.poll());
        }else{
            B.add(num);
            A.add(B.poll());
        }
    }

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

1.41 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。
```




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

1.42 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
```

$digit = 10$　　即求“十位”的 1 的个数

将数字 n 划分为三部分：

$n =$ 2 3 0 4

high　cur　low

出现 1 的数字范围：$0010 \sim 2219$

只看高低位：$000 \sim 229$

易得 1 出现次数为：$229 - 0 + 1 = \mathbf{230}$

结论：

当此位 $cur = 0$ 时，此位 1 的个数的计算公式为：

$high \times digit$

即 $23 \times 10 = \mathbf{230}$

```
//想象成数字密码锁，固定一个数字转动
//cur == 0  :   high×digit
//cur == 1  :   high×digit+low+1
//cur == 2~9:   (high+1)×digit
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;
    }
}
```

1.43 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
```

解：

1.将 101112 \cdots101112⋯ 中的每一位称为 数位 ， 记为 nn ；
2.将 10, 11, 12, \cdots10,11,12,⋯ 称为 数字 ， 记为 numnum ；
3.数字 1010 是一个两位数，称此数字的 位数 为 22 ， 记为 digitdigit ；
4.每 digitdigit 位数的起始数字（即：1, 10, 100, \cdots1,10,100,⋯） ， 记为 startstart

数字范围	位数	数字数量	数位数量
1~9	1	9	9
10~99	2	90	180
100~999	3	900	2700
...	...	...	...
$start$~end	$digit$	$9 \times start$	$9 \times start \times digit$

位数递推公式 $digit = digit + 1$

起始数字递推公式 $start = start \times 10$

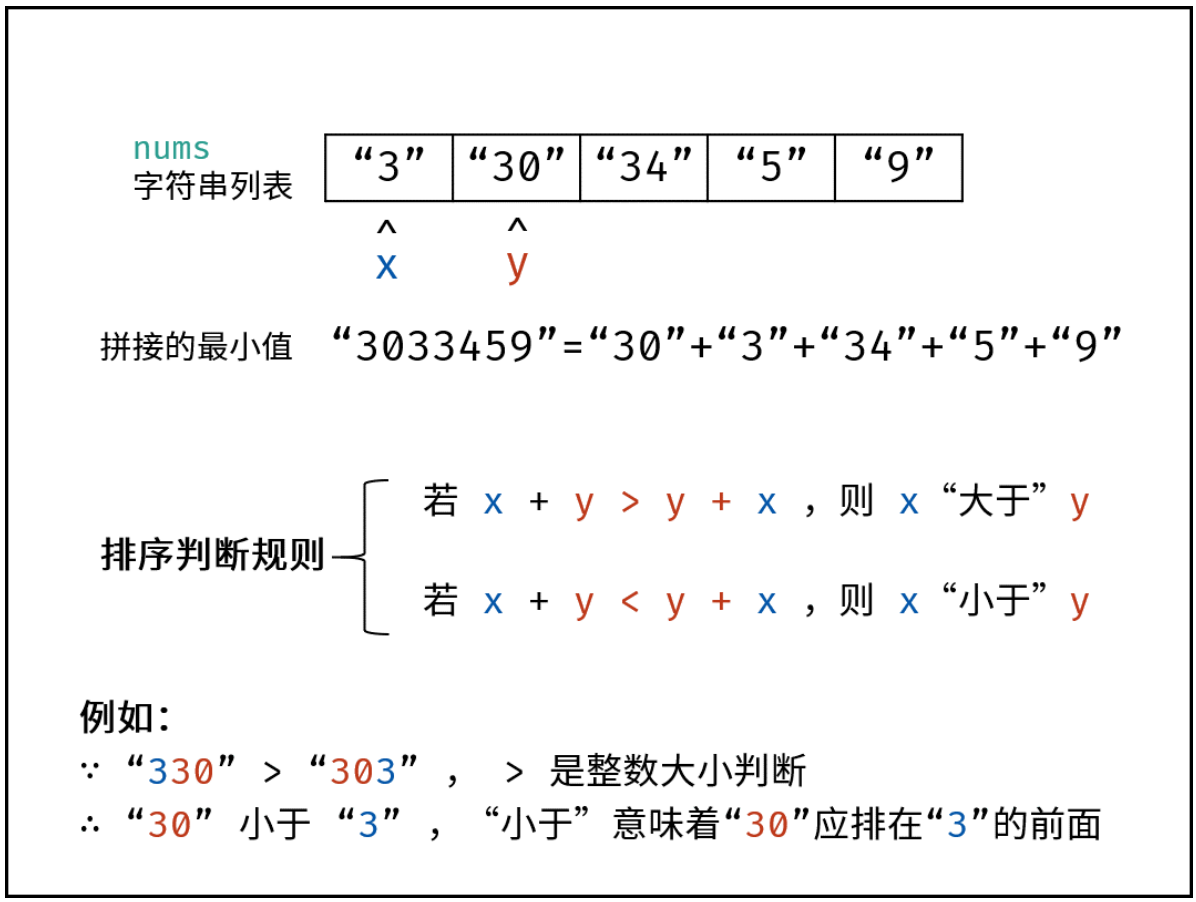
数位数量计算公式 $count = 9 \times start \times digit$

```java
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.
    }
}
```

1.44 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"
```




```
class Solution {
    public String minNumber(int[] nums) {
        String[] strs = new String[nums.length];
        for(int i = 0; i < nums.length; i++)
            strs[i] = String.valueOf(nums[i]);
        Arrays.sort(strs, (x, y) -> (x + y).compareTo(y + x));
        StringBuilder res = new StringBuilder();
        for(String s : strs)
            res.append(s);
        return res.toString();
    }
}
```

1.45 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"
```

```
//有条件的青蛙跳台阶
class Solution {
    public int translateNum(int num) {
        String s = String.valueOf(num);
        int a = 1, b = 1;
        for(int i = 2; i <= s.length(); i++) {
            String tmp = s.substring(i - 2, i);
            int c = tmp.compareTo("10") >= 0 && tmp.compareTo("25") <= 0 ? a + b
: a;
            b = a;      //a在b前
            a = c;
        }
        return a;
    }
}
```

1.46 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？

```
//经典DP问题，思路较为简单
class Solution {
    public int maxValue(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                if(i == 0 && j == 0) continue;
                if(i == 0) grid[i][j] += grid[i][j - 1] ;
                else if(j == 0) grid[i][j] += grid[i - 1][j];
                else grid[i][j] += Math.max(grid[i][j - 1], grid[i - 1][j]);
            }
        }
        return grid[m - 1][n - 1];
    }
}
```

1.47 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。
```

```
//滑动窗口
class Solution {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> dic = new HashMap<>();
        int i = -1, res = 0;
        for(int j = 0; j < s.length(); j++) {
            if(dic.containsKey(s.charAt(j)))
                i = Math.max(i, dic.get(s.charAt(j))); // 更新左指针 i
            dic.put(s.charAt(j), j); // 哈希表记录
            res = Math.max(res, j - i); // 更新结果
        }
        return res;
    }
}
```

1.48 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。
```

```
class Solution {
    public int nthUglyNumber(int n) {
        int[] dp = new int[n];  // 使用dp数组来存储丑数序列
        dp[0] = 1;  // dp[0]已知为1
        int a = 0, b = 0, c = 0;    // 下个应该通过乘2来获得新丑数的数据是第a个， 同理b,
c
        for(int i = 1; i < n; i++){
            // 第a丑数个数需要通过乘2来得到下个丑数，第b丑数个数需要通过乘2来得到下个丑数，同
理第c个数
            int n2 = dp[a] * 2, n3 = dp[b] * 3, n5 = dp[c] * 5;
            dp[i] = Math.min(Math.min(n2, n3), n5);
            if(dp[i] == n2){
                a++; // 第a个数已经通过乘2得到了一个新的丑数，那下个需要通过乘2得到一个新的
丑数的数应该是第(a+1)个数
            }
            if(dp[i] == n3){
                b++; // 第 b个数已经通过乘3得到了一个新的丑数，那下个需要通过乘3得到一个新
的丑数的数应该是第(b+1)个数
            }
            if(dp[i] == n5){
```

```
                c++; // 第 c个数已经通过乘5得到了一个新的丑数，那下个需要通过乘5得到一个新的丑数的数应该是第(c+1)个数
            }
        }
        return dp[n-1];
    }
}
```

1.49 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。

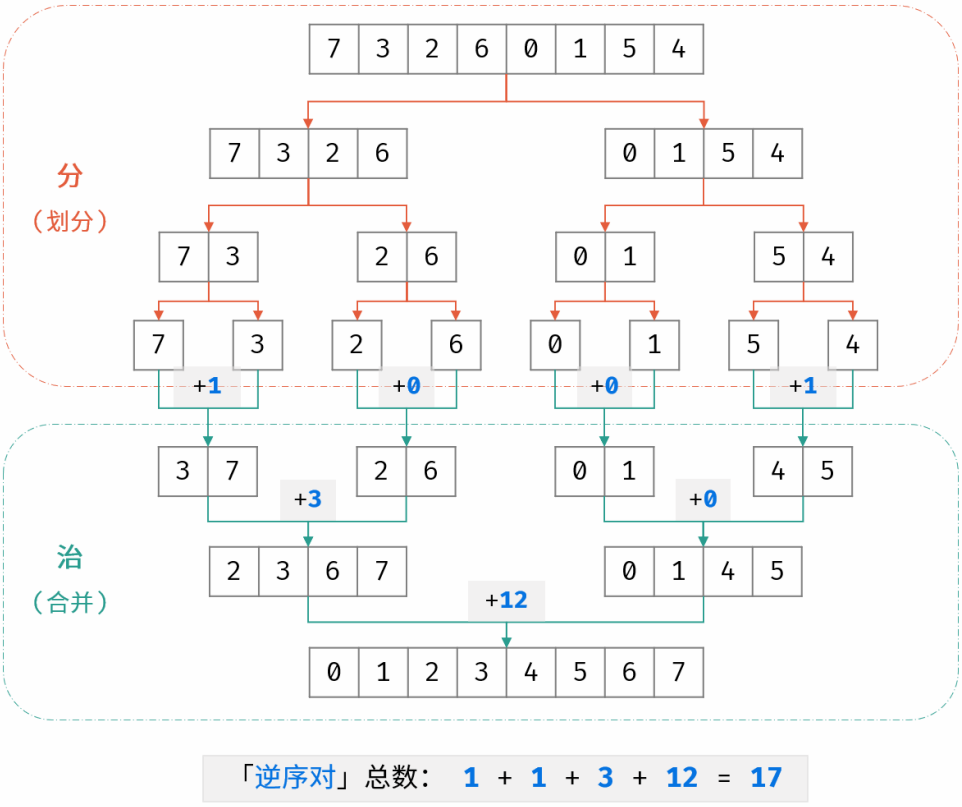
```
示例 1:
输入：s = "abaccdeff"
输出：'b'
```

```
class Solution {
    public char firstUniqChar(String s) {
        HashMap<Character,Boolean> map = new HashMap<>();
        char[] sc = s.toCharArray();
        for(char e:sc){
            map.put(e,!map.containsKey(e));
        }
        for(char e:sc){
            if(map.get(e)) return e;
        }
        return ' ';
    }
}
```

1.50 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
```



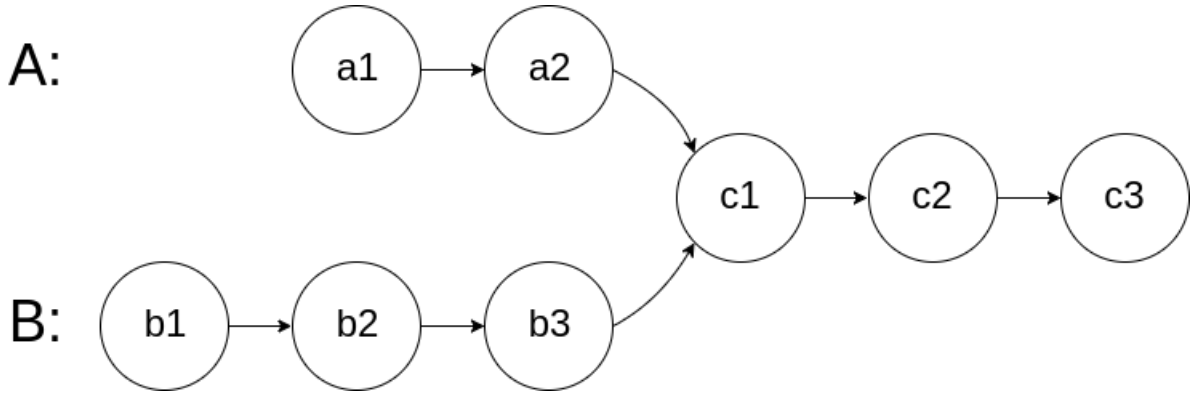

```java
//仅需要在归并排序中添加一行统计数量
class Solution {
   int count = 0;
    public int reversePairs(int[] nums) {
        this.count = 0;
        sort(nums, 0, nums.length - 1,new int[nums.length]);
        return count;
    }

    public void sort(int[] nums,int left,int right,int[] temp){
        if(left < right){
            int mid = (left+right) / 2;//开始递归划分
            sort(nums,left,mid,temp);//归并排序左部分（left,mid)
            sort(nums,mid+1,right,temp);//归并排序右部分(mid+1,right)
            merge(nums,left,mid,right,temp);//合并
        }
    }

    private void merge(int[] nums, int left, int mid, int right, int[] temp) {
        int i = left;//左部分首元素
        int j = mid + 1;//右部分首元素
        int t = 0;
        while(i <=mid && j <=right){//在范围之内
            if(nums[i] <= nums[j]){
                temp[t++] = nums[i++];
            }else{
                count += (mid - i + 1);//只需要这行代码
                temp[t++] = nums[j++];
            }
        }
        while (i <= mid){//右边遍历完事了    左边还剩呢
            temp[t++] = nums[i++];
```

```
        }
        while( j <= right){//左边遍历完事了    右边还剩了
            temp[t++] = nums[j++];
        }
        t = 0;//将temp中的元素   全部都copy到原数组里边去
        while (left <=right){
            nums[left++] = temp[t++];
        }
    }
}
```

1.51 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。

```
//双指针
//你变成我，走过我走过的路。
//我变成你，走过你走过的路。
//然后我们便相遇了...
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        ListNode A = headA,B = headB;
        while(A != B){
            A = A != null ? A.next : headB;
            B = B != null ? B.next : headA;
        }
        return A;
    }
}
```

1.52 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
```

```java
//二分查找
class Solution {
    public int search(int[] nums, int target) {
        return helper(nums, target) - helper(nums, target - 1);
    }
    int helper(int[] nums, int tar) {
        int i = 0, j = nums.length - 1;
        while(i <= j) {
            int m = (i + j) / 2;
            if(nums[m] <= tar) i = m + 1;
            else j = m - 1;
        }
        return i;
    }
}
```

1.53 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
```

```java
//二分查找类
//判断mid索引和nums[mid]值是否相等
class Solution {
    public int missingNumber(int[] nums) {
        int left = 0,right = nums.length - 1;
        while(left <= right){
            int mid = left + (right - left) / 2;
            if(mid == nums[mid]){
                left = mid + 1;
            }else{
                right = mid - 1;
            }
        }
        return left;
    }
}
```

1.54 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 大的节点的值。

```
示例 1:

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

```
//中序遍历为递增序列，改成中序得倒序遍历为递减序列
class Solution {
    int k,res;
    public int kthLargest(TreeNode root, int k) {
        this.k = k;
        dfs(root);
        return res;
    }

    void dfs(TreeNode root){
        if(root == null) return;
        dfs(root.right);            //先递归右子树
        if(k == 0) return;
        if(--k == 0) res = root.val;
        dfs(root.left);
    }
}
```

1.55 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 。
```

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

1.56 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。

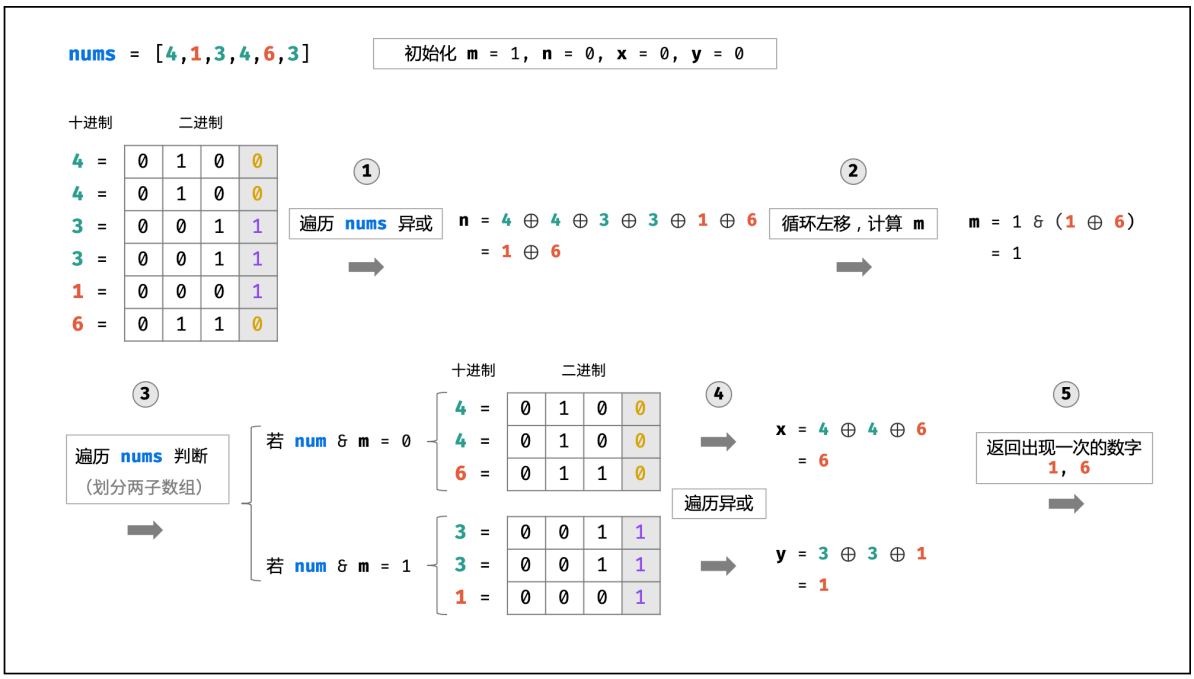
```
示例 1:

给定二叉树 [3,9,20,null,null,15,7]
    3
   / \
  9  20
    /  \
   15   7
返回 true 。
```

```
//后序遍历
class Solution {
    public boolean isBalanced(TreeNode root) {
        return recur(root) != -1;
    }

    int recur(TreeNode root){
        if(root == null) return 0;
        int left = recur(root.left);
        if(left == -1) return -1;
        int right = recur(root.right);
        if(right == -1) return -1;
        return Math.abs(left - right) < 2 ? Math.max(left,right) + 1 : -1;
    }
}
```

1.57 数组中数字出现的次数 I

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]
```




```
/*
1.先对所有数字进行一次异或，得到两个出现一次的数字的异或值。
2.在异或结果中找到任意为 1 的位。
3.根据这一位对所有的数字进行分组。
4.在每个组内进行异或操作，得到两个数字。
*/
class Solution {
    public int[] singleNumbers(int[] nums) {
        int x = 0, y = 0, n = 0, m = 1;
        for(int num : nums)                 // 1. 遍历异或
            n ^= num;
        while((n & m) == 0)                 // 2. 循环左移，计算 m
            m <<= 1;
        for(int num: nums) {                // 3. 遍历 nums 分组
            if((num & m) != 0) x ^= num;    // 4. 当 num & m != 0
            else y ^= num;                  // 4. 当 num & m == 0
        }
        return new int[] {x, y};            // 5. 返回出现一次的数字
    }
}
```

1.58 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
```

```
//位运算
class Solution {
    public int singleNumber(int[] nums) {
```

```
        int[] counts = new int[32];
        for(int num : nums) {                   //记录所有数字的各二进制位的1的出现次数
            for(int j = 0; j < 32; j++) {
                counts[j] += num & 1;
                num >>>= 1;
            }
        }
        int res = 0, m = 3;
        for(int i = 0; i < 32; i++) {
            counts[31 - i] %= m;                //将各元素对3求余，则结果为“只出现一次的数字”的各二进制位。
            res <<= 1;                          //移位恢复数字
            res |= counts[31 - i] ;
        }
        return res;
    }
}
```

1.59 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]
```

```
//双指针
class Solution {
    public int[] twoSum(int[] nums, int target) {
        int i = 0,j = nums.length - 1,sum;
        while(i < j){
            sum = nums[i] + nums[j];
            if(sum < target) i++;
            else if(sum > target) j--;
            else return new int[] {nums[i],nums[j]};
        }
        return new int[0];
    }
}
```

1.60 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]]
```


```java
//滑动窗口
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][]);
    }
}
```

1.61 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"
```

```java
//双指针
class Solution {
    public String reverseWords(String s) {
        s = s.trim(); // 删除首尾空格
        int j = s.length() - 1, i = j;
        StringBuilder res = new StringBuilder();
        while(i >= 0) {
            while(i >= 0 && s.charAt(i) != ' ') i--; // 搜索首个空格
            res.append(s.substring(i + 1, j + 1) + " "); // 添加单词
            while(i >= 0 && s.charAt(i) == ' ') i--; // 跳过单词间空格
            j = i; // j 指向下个单词的尾字符
        }
        return res.toString().trim(); // 转化为字符串并返回
    }
}
```

1.62 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"
```

```java
//遍历字符加入尾部
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()));       //利用取模运算简化
        return res.toString();
    }
}
```

1.63 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
```

```
//单调队列
//本题难点在于如何在每次窗口滑动后，将“获取窗口内最大值”的时间复杂度从 O(k) 降低至 O(1)。
//使用单调队列。遍历数组时，每轮保证单调队列 deque ：
//deque内仅包含窗口内的元素 ⇒ 每轮窗口滑动移除了元素 nums[i−1] ，需将 deque 内的对应元素一起删除。
//deque内的元素 非严格递减 ⇒ 每轮窗口滑动添加了元素 nums[j+1] ，需将 deque 内所有<nums[j+1] 的元素删除。
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if(nums.length == 0 || k == 0) return new int[0];
        Deque<Integer> deque = new LinkedList<>();
        int[] res = new int[nums.length - k + 1];
        // 未形成窗口
        for(int i = 0; i < k; i++) {
            while(!deque.isEmpty() && deque.peekLast() < nums[i])
                deque.removeLast();
            deque.addLast(nums[i]);
        }
        res[0] = deque.peekFirst();
        // 形成窗口后
        for(int i = k; i < nums.length; i++) {
            if(deque.peekFirst() == nums[i - k])
                deque.removeFirst();
            while(!deque.isEmpty() && deque.peekLast() < nums[i])
                deque.removeLast();
            deque.addLast(nums[i]);
            res[i - k + 1] = deque.peekFirst();
        }
        return res;
    }
}
```

1.64 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]
```

```
//双端队列
//使用一个双端队列 deque，在每次入队时，如果 deque 队尾元素小于即将入队的元素 value，则将小于 value 的元素全部出队后，再将 value 入队；否则直接入队。

class MaxQueue {
    Queue<Integer> q;
    Deque<Integer> d;

    public MaxQueue() {
        q = new LinkedList<Integer>();
        d = new LinkedList<Integer>();
    }

    public int max_value() {
        if (d.isEmpty()) {
            return -1;
        }
        return d.peekFirst();
    }

    public void push_back(int value) {
        while (!d.isEmpty() && d.peekLast() < value) {
            d.pollLast();
        }
        d.offerLast(value);
        q.offer(value);
    }

    public int pop_front() {
        if (q.isEmpty()) {
            return -1;
        }
        int ans = q.poll();
        if (ans == d.peekFirst()) {
            d.pollFirst();
        }
        return ans;
    }
}
```

1.65 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]
```

```java
//DP
class Solution {
    public double[] dicesProbability(int n) {
        double[] dp = new double[6];
        Arrays.fill(dp, 1.0 / 6.0);
        for (int i = 2; i <= n; i++) {
            double[] tmp = new double[5 * i + 1];
            for (int j = 0; j < dp.length; j++) {
                for (int k = 0; k < 6; k++) {
                    tmp[j + k] += dp[j] / 6.0;
                }
            }
            dp = tmp;
        }
        return dp;
    }
}
```

1.66 扑克牌中的顺子

从若干副扑克牌中随机抽 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
```

```java
//Set集合
class Solution {
    public boolean isStraight(int[] nums) {
        Set<Integer> repeat = new HashSet<>();
        int max = 0, min = 14;
        for(int num : nums) {
            if(num == 0) continue; // 跳过大小王
            max = Math.max(max, num); // 最大牌
            min = Math.min(min, num); // 最小牌
            if(repeat.contains(num)) return false; // 若有重复，提前返回 false
            repeat.add(num); // 添加此牌至 Set
        }
        return max - min < 5; // 最大牌 - 最小牌 < 5 则可构成顺子
```

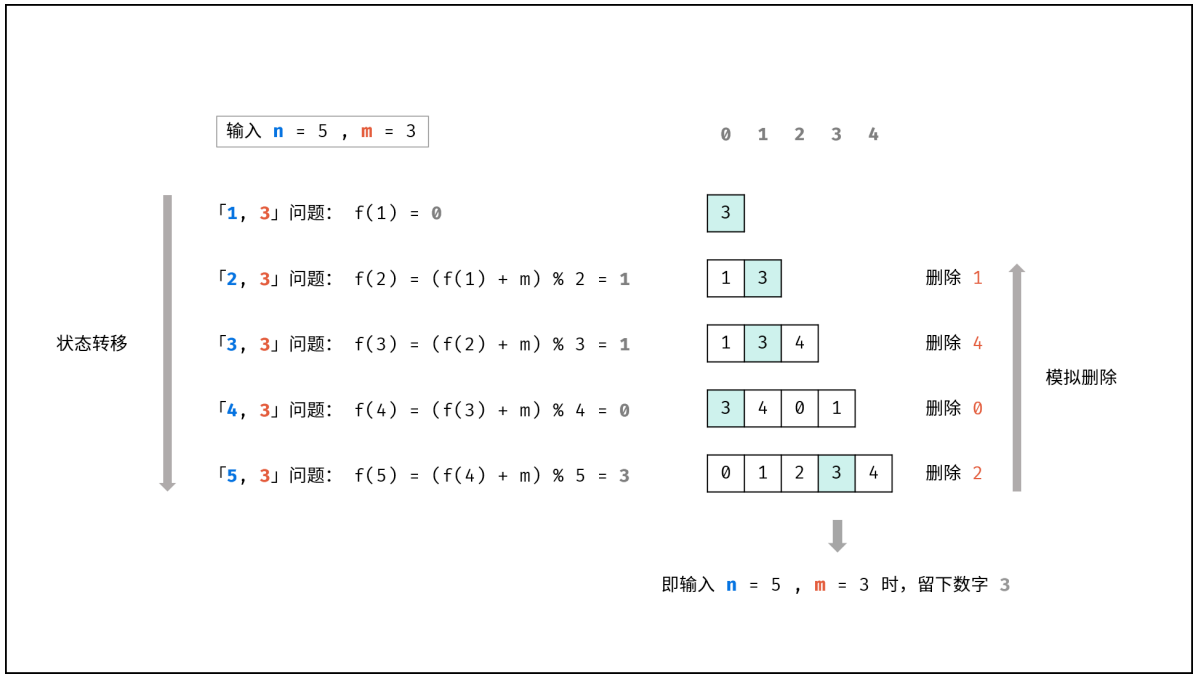
```
    }
}
```

1.67 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
```

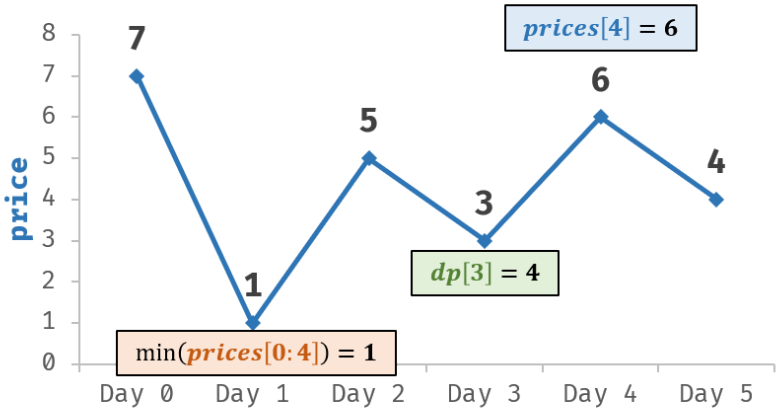



```
//DP
class Solution {
    public int lastRemaining(int n, int m) {
        int x = 0;
        for (int i = 2; i <= n; i++) {
            x = (x + m) % i;
        }
        return x;
    }
}
```

1.68 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。
```



例如前 4 日的最大利润为:

$$dp[4] = \max(dp[3], prices[4] - \min(prices[0:4]))$$
$$= \max(4, 6 - 1)$$
$$= 5$$

转移方程:

$$dp[i] = \max(\ dp[i-1], prices[i] - \min(prices[0:i])\ )$$

```java
//DP
class Solution {
    public int maxProfit(int[] prices) {
        int cost = Integer.MAX_VALUE,profit = 0;
        for(int price : prices){
            cost = Math.min(cost,price);
            profit = Math.max(profit,price - cost);
        }
        return profit;
    }
}
```

1.69 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
```

```
//逻辑符短路
class Solution {
    public int sumNums(int n) {
        boolean x = n > 1 && (n += sumNums(n - 1)) > 0;
        return n;
    }
}
```

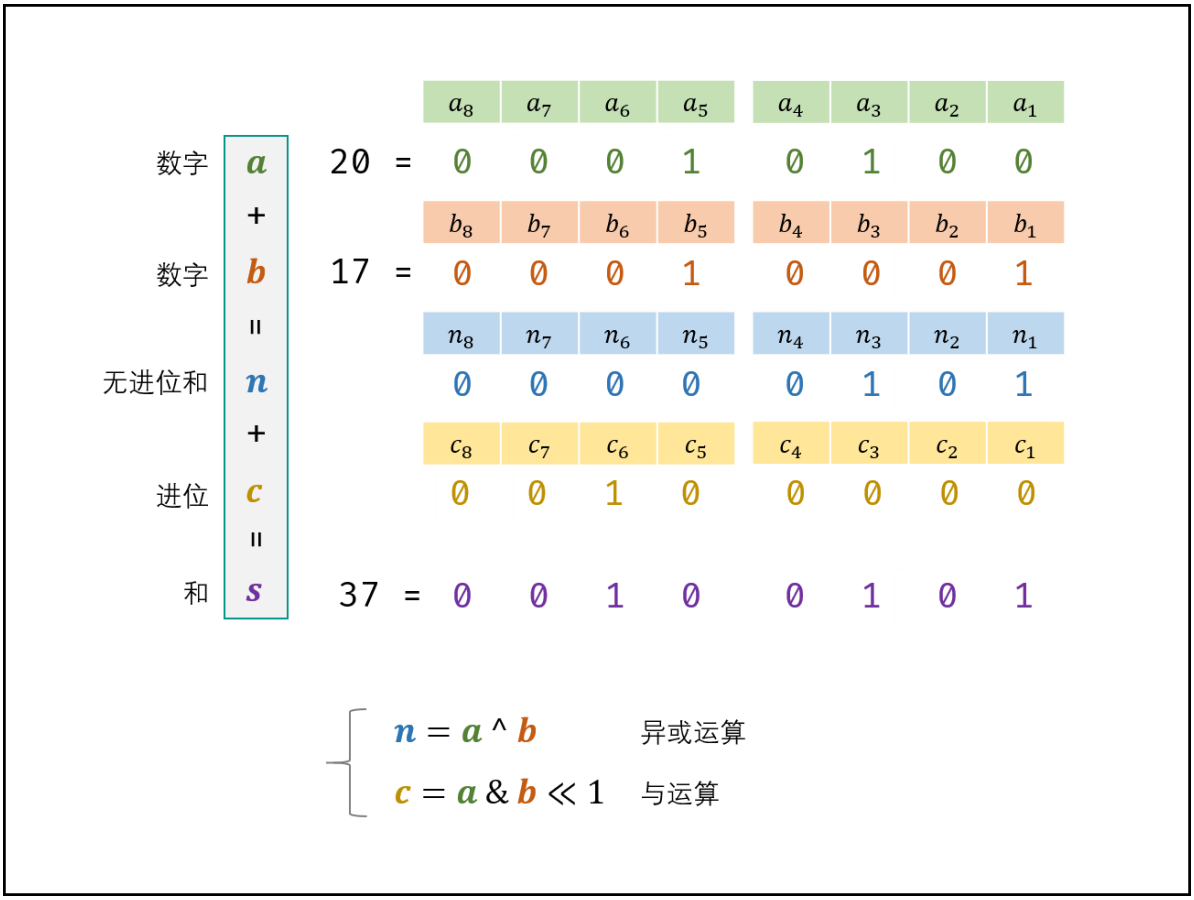
1.70 不用加减乘除做加法

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。

解：

$a(i)$	$b(i)$	**无进位和** $n(i)$	**进位** $c(i+1)$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

观察发现，无进位和 与 异或运算 规律相同，进位 和 与运算 规律相同（并需左移一位）。




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

1.71 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]
```

```
// 表格法
class Solution {
    public int[] constructArr(int[] a) {
        if(a.length == 0) return new int[0];
        int[] b = new int[a.length];
        b[0] = 1;
```

```
        int temp = 1;

        // 计算b：b[i]表示a[0...i-1]的乘积
        for(int i=1; i<a.length; i++){
            b[i] = b[i-1] * a[i-1];
        }
        // temp 表示a[i...n-1]的乘积
        for(int i=a.length-1; i>=0; i--){
            b[i] *= temp;
            temp *= a[i];
        }
        return b;
    }
}
```

1.72 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) 。

```
class Solution {
    public int strToInt(String str) {
        //去除str首尾的多余空格
        char[] array = str.trim().toCharArray();
        //如果array的长度为0 返回0
        if (array.length==0) return 0;
        //sign表示标志位  1为正  -1 为负  i代表array从何处开始遍历
        int res = 0, sign = 1, i = 1;
        //设置限制值，因为在遍历中先判断res是否越界，再向res赋值，因而对limit的要求/10
        int limit = Integer.MAX_VALUE / 10;
        //如果array[0]=- 表明该数是负数  sign =-1
        if (array[0]=='-') sign = -1;
        else if (array[0]!='+') i = 0;
        for (int j = i; j < array.length; j++) {
            //判断当前字符是否为数字  不是直接退出
            if (array[j]>'9'||array[j]<'0') break;
            //判断遍历到j-1的位置后  res是否大于limit 如果当前res已经大于limit  加上
array[j]一定越界
            //当res等于limit时，我们需要判断array[j]是否大于Integer.MAX_VALUE的末位数7
```

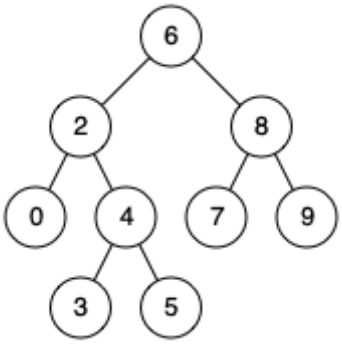
```
            if (res>limit||res==limit&&array[j]>'7') return sign == 1 ?
Integer.MAX_VALUE : Integer.MIN_VALUE;
            res = res * 10 + (array[j] - '0');
        }
        return res*sign;
    }
}
```

1.73 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。
```

```
//迭代
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        while(root != null) {
            if(root.val < p.val && root.val < q.val) // p,q 都在 root 的右子树中
                root = root.right; // 遍历至右子节点
            else if(root.val > p.val && root.val > q.val) // p,q 都在 root 的左子树
中
                root = root.left; // 遍历至左子节点
            else break;
        }
        return root;
    }
}
```

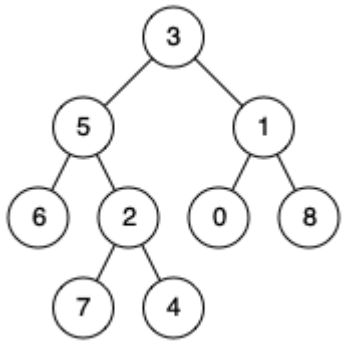
```
//递归
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
        if(root.val < p.val && root.val < q.val)
            return lowestCommonAncestor(root.right, p, q);
        if(root.val > p.val && root.val > q.val)
            return lowestCommonAncestor(root.left,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

1.74 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。
```

```
//DFS
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);
        //左右子树同时为null，root左右子树均不包含p，q，返回null
        if(left == null && right == null) return null;
        //左为空，右不为空，p，q不在左子树中，返回right，分为两种情况：
        //p,q 其中一个在root的右子树中，此时right指向p（假设为p）；
        //p,q 两节点都在root的右子树中，此时的right指向最近公共祖先节点 ；
        if(left == null) return right;
        //左不为空，右为空，p，q不在右子树中，返回left
        if(right == null) return left;
        return root;    //if(left != null and right != null)
```

```
    }
}
```

二、LeetCode热题100

2.1 两数之和

给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 target 的那 两个 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

```
示例 1：

输入：nums = [2,7,11,15], target = 9
输出：[0,1]
解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。
```

```
//建立一个哈希表存储
class Solution {
    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer,Integer> map = new HashMap<>();
        for(int i = 0;i < nums.length;i++){
            if(map.containsKey(target - nums[i]))
                return new int[]{map.get(target-nums[i]),i};
            map.put(nums[i],i);
        }
        return new int[0];
    }
}
```

2.2 两数相加

给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储 一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

```
输入：l1 = [2,4,3], l2 = [5,6,4]
输出：[7,0,8]
解释：342 + 465 = 807
```

```
//注意进位即可
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode head = null,tail = null;
        int carry = 0;
        while(l1 != null || l2 != null){
```

```
            int n1 = l1 != null ? l1.val : 0;
            int n2 = l2 != null ? l2.val : 0;
            int sum = n1 + n2 + carry;
            carry = sum / 10;
            if(head == null){
                tail = head = new ListNode(sum % 10);
            }else{
                tail.next = new ListNode(sum % 10);
                tail = tail.next;
            }
            if(l1 != null)
                l1 = l1.next;
            if(l2 != null)
                l2 = l2.next;
        }
        if(carry > 0)
            tail.next = new ListNode(carry);
        return head;
    }
}
```

2.3 无重复字符的最长子串

与剑指offer47相同

2.4 寻找两个正序数组的中位数

给定两个大小分别为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。请你找出并返回这两个正序数组的 中位数 。

算法的时间复杂度应该为 O(log (m+n)) 。

```
示例 1：

输入：nums1 = [1,3], nums2 = [2]
输出：2.00000
解释：合并数组 = [1,2,3] ，中位数 2
```

```
//二分思想，hard题目
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int length1 = nums1.length, length2 = nums2.length;
        int totalLength = length1 + length2;
        if (totalLength % 2 == 1) {
            int midIndex = totalLength / 2;
            double median = getKthElement(nums1, nums2, midIndex + 1);
            return median;
        } else {
            int midIndex1 = totalLength / 2 - 1, midIndex2 = totalLength / 2;
            double median = (getKthElement(nums1, nums2, midIndex1 + 1) +
getKthElement(nums1, nums2, midIndex2 + 1)) / 2.0;
            return median;
        }
```

```java
    }

    public int getKthElement(int[] nums1, int[] nums2, int k) {
        /* 主要思路：要找到第 k (k>1) 小的元素，那么就取 pivot1 = nums1[k/2-1] 和 pivot2 = nums2[k/2-1] 进行比较
         * 这里的 "/" 表示整除
         * nums1 中小于等于 pivot1 的元素有 nums1[0 .. k/2-2] 共计 k/2-1 个
         * nums2 中小于等于 pivot2 的元素有 nums2[0 .. k/2-2] 共计 k/2-1 个
         * 取 pivot = min(pivot1, pivot2)，两个数组中小于等于 pivot 的元素共计不会超过 (k/2-1) + (k/2-1) <= k-2 个
         * 这样 pivot 本身最大也只能是第 k-1 小的元素
         * 如果 pivot = pivot1，那么 nums1[0 .. k/2-1] 都不可能是第 k 小的元素。把这些元素全部 "删除"，剩下的作为新的 nums1 数组
         * 如果 pivot = pivot2，那么 nums2[0 .. k/2-1] 都不可能是第 k 小的元素。把这些元素全部 "删除"，剩下的作为新的 nums2 数组
         * 由于我们 "删除" 了一些元素（这些元素都比第 k 小的元素要小），因此需要修改 k 的值，减去删除的数的个数
         */

        int length1 = nums1.length, length2 = nums2.length;
        int index1 = 0, index2 = 0;
        int kthElement = 0;

        while (true) {
            // 边界情况
            if (index1 == length1) {
                return nums2[index2 + k - 1];
            }
            if (index2 == length2) {
                return nums1[index1 + k - 1];
            }
            if (k == 1) {
                return Math.min(nums1[index1], nums2[index2]);
            }

            // 正常情况
            int half = k / 2;
            int newIndex1 = Math.min(index1 + half, length1) - 1;
            int newIndex2 = Math.min(index2 + half, length2) - 1;
            int pivot1 = nums1[newIndex1], pivot2 = nums2[newIndex2];
            if (pivot1 <= pivot2) {
                k -= (newIndex1 - index1 + 1);
                index1 = newIndex1 + 1;
            } else {
                k -= (newIndex2 - index2 + 1);
                index2 = newIndex2 + 1;
            }
        }
    }
}
```

2.5 最长回文子串

给你一个字符串 `s`，找到 `s` 中最长的回文子串。

```
示例 1：
输入：s = "babad"
输出："bab"
解释："aba" 同样是符合题意的答案
```

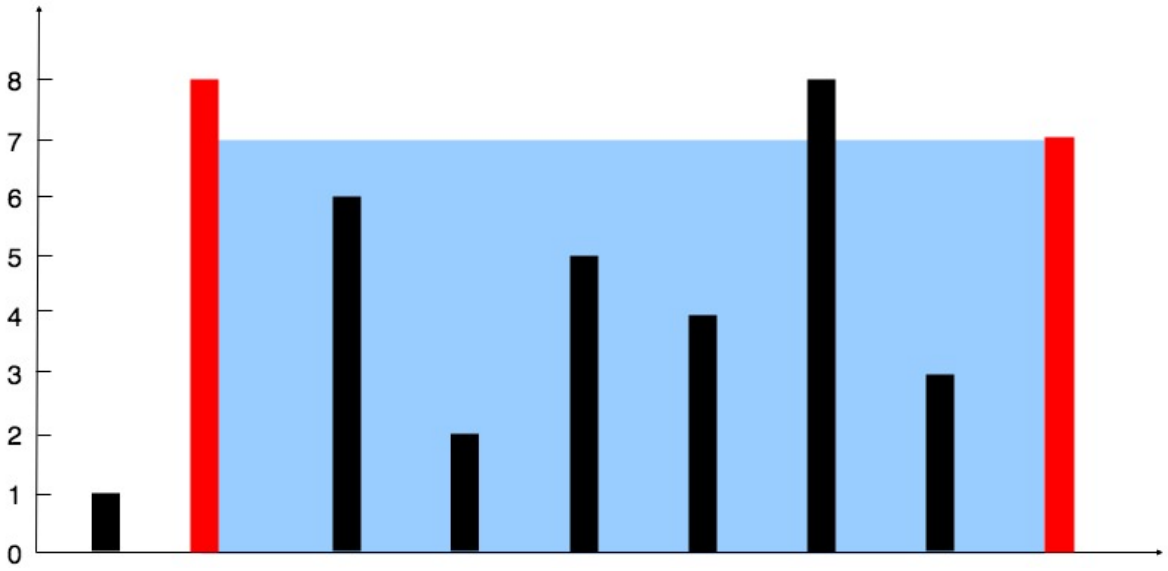
```
//中心扩散法
class Solution {
    public String longestPalindrome(String s) {
        if (s == null || s.length() < 1){
            return "";
        }
        // 初始化最大回文子串的起点和终点
        int start = 0;
        int end   = 0;
        // 遍历每个位置，当做中心位
        for (int i = 0; i < s.length(); i++) {
            // 分别拿到奇数偶数的回文子串长度
            int len_odd = expandCenter(s,i,i);
            int len_even = expandCenter(s,i,i + 1);
            // 对比最大的长度
            int len = Math.max(len_odd,len_even);
            // 计算对应最大回文子串的起点和终点
            if (len > end - start){
                start = i - (len - 1)/2;
                end = i + len/2;
            }
        }
        // 注意：这里的end+1是因为 java自带的左闭右开的原因
        return s.substring(start,end + 1);
    }

    private int expandCenter(String s,int left,int right){
        // left = right 的时候，此时回文中心是一个字符，回文串的长度是奇数
        // right = left + 1 的时候，此时回文中心是一个空隙，回文串的长度是偶数
        // 跳出循环的时候恰好满足 s.charAt(left) ! = s.charAt(right)
        while (left >= 0 && right < s.length() && s.charAt(left) ==
s.charAt(right)){
            left--;
            right++;
        }
        // 回文串的长度是right-left+1-2 = right - left - 1
        return right - left - 1;
    }
}
```

2.6 正则表达式匹配

同剑指offer18题

2.7 盛最多水的容器

给你 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 (i, ai) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, ai) 和 (i, 0) 。找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。



示例：
输入：[1,8,6,2,5,4,8,3,7]
输出：49
解释：图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

```
//双指针
public class Solution {
    public int maxArea(int[] height) {
        int l = 0, r = height.length - 1;
        int res = 0;
        while (l < r) {
            int area = Math.min(height[l], height[r]) * (r - l);     //计算面积
            res = Math.max(res, area);                               //与最大面积比
较替换
            if (height[l] <= height[r]) {                            //移动短的一边
                ++l;
            }
            else {
                --r;
            }
        }
        return res;
    }
}
```

2.8 三数之和

给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？请你找出所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

```
示例 1：

输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]
```

```
//排序+双指针
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {// 总时间复杂度：O(n^2)
        List<List<Integer>> ans = new ArrayList<>();
        if (nums == null || nums.length <= 2) return ans;

        Arrays.sort(nums); // O(nlogn)

        for (int i = 0; i < nums.length - 2; i++) { // O(n^2)
            if (nums[i] > 0) break; // 第一个数大于 0，后面的数都比它大，肯定不成立了
            if (i > 0 && nums[i] == nums[i - 1]) continue; // 去掉重复情况
            int target = -nums[i];
            int left = i + 1, right = nums.length - 1;
            while (left < right) {
                if (nums[left] + nums[right] == target) {
                    ans.add(new ArrayList<>(Arrays.asList(nums[i], nums[left],
nums[right])));

                    // 现在要增加 left，减小 right，但是不能重复，比如: [-2, -1, -1,
-1, 3, 3, 3], i = 0, left = 1, right = 6, [-2, -1, 3] 的答案加入后，需要排除重复的 -1
和 3
                    left++; right--; // 首先无论如何先要进行加减操作
                    while (left < right && nums[left] == nums[left - 1]) left++;
                    while (left < right && nums[right] == nums[right + 1])
right--;
                } else if (nums[left] + nums[right] < target) {
                    left++;
                } else {  // nums[left] + nums[right] > target
                    right--;
                }
            }
        }
        return ans;
    }
}
```

2.9 电话号码的字母组合

给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按 任意顺序 返回。

给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。

示例 1:
输入: digits = "23"
输出: ["ad","ae","af","bd","be","bf","cd","ce","cf"]

```
//回溯法
class Solution {
    // 数字到号码的映射
    private String[] map = {"abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
    // 路径
    private StringBuilder sb = new StringBuilder();
    // 结果集
    private List<String> res = new ArrayList<>();
    public List<String> letterCombinations(String digits) {
        if(digits == null || digits.length() == 0) return res;
        backtrack(digits,0);
        return res;
    }
    // 回溯函数
    private void backtrack(String digits,int index) {
        if(sb.length() == digits.length()) {
            res.add(sb.toString());
            return;
        }
        String val = map[digits.charAt(index)-'2'];
        for(char ch:val.toCharArray()) {
            sb.append(ch);
            backtrack(digits,index+1);
            sb.deleteCharAt(sb.length()-1);
        }
    }
}
```

2.10 删除链表的倒数第 N 个结点

同剑指offer21题相同思路，使用快慢指针

```
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode pre = dummy;
        ListNode slow = head;
        ListNode fast = head;
        for(int i=0;i<n;i++){
```

```
            fast = fast.next;
        }
        while(fast!=null){
            pre = pre.next;
            slow = slow.next;
            fast = fast.next;
        }
        pre.next = slow.next;
        return dummy.next;
    }
}
```

2.11 有效的括号

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。
左括号必须以正确的顺序闭合。

```
示例 1：

输入：s = "()"
输出：true
```

```
//辅助栈
class Solution {
    public boolean isValid(String s) {
        int n = s.length();
        if (n % 2 == 1) {
            return false;
        }

        Map<Character, Character> pairs = new HashMap<Character, Character>() {{
            put(')', '(');      //注意键值
            put(']', '[');
            put('}', '{');
        }};
        Deque<Character> stack = new LinkedList<Character>();
        for (int i = 0; i < n; i++) {
            char ch = s.charAt(i);
            if (pairs.containsKey(ch)) {        //左括号入栈，右括号与栈顶元素比较
                if (stack.isEmpty() || stack.peek() != pairs.get(ch)) {
                    return false;
                }
                stack.pop();
            } else {
                stack.push(ch);
            }
        }
        return stack.isEmpty();
    }
}
```

2.12 合并两个有序链表

同剑指offer23题一样

2.13 括号生成

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

```
示例 1：

输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]
```

```
//看做隐式树，DFS+剪枝
public class Solution {
    // 做加法
    public List<String> generateParenthesis(int n) {
        List<String> res = new ArrayList<>();
        if (n == 0) {
            return res;
        }
        dfs("", 0, 0, n, res);
        return res;
    }

    /**
     * @param curStr 当前递归得到的结果
     * @param left   左括号已经用了几个
     * @param right  右括号已经用了几个
     * @param n      左括号、右括号一共得用几个
     * @param res    结果集
     */
    private void dfs(String curStr, int left, int right, int n, List<String>
res) {
        if (left == n && right == n) {
            res.add(curStr);
            return;
        }
        // 剪枝
        if (left < right) {
            return;
        }
        if (left < n) {
            dfs(curStr + "(", left + 1, right, n, res);
        }
        if (right < n) {
            dfs(curStr + ")", left, right + 1, n, res);
        }
    }
}
```

2.14 合并K个升序链表

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

```
//优先级队列
class Solution {
   public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) return null;
        PriorityQueue<ListNode> queue = new PriorityQueue<>(lists.length, new
Comparator<ListNode>() {
            @Override
            public int compare(ListNode o1, ListNode o2) {
                if (o1.val < o2.val) return -1;
                else if (o1.val == o2.val) return 0;
                else return 1;
            }
        });
        ListNode dummy = new ListNode(0);
        ListNode p = dummy;
        for (ListNode node : lists) {
            if (node != null) queue.add(node);
        }
        while (!queue.isEmpty()) {
            p.next = queue.poll();
            p = p.next;
            if (p.next != null) queue.add(p.next);
        }
        return dummy.next;
    }
}
//也可以使用分治的思想，两两合并链表
```

2.15 下一个排列

整数数组的 下一个排列 是指其整数的下一个字典序更大的排列。更正式地，如果数组的所有排列根据其字典顺序从小到大排列在一个容器中，那么数组的 下一个排列 就是在这个有序容器中排在它后面的那个排列。如果不存在下一个更大的排列，那么这个数组必须重排为字典序最小的排列（即，其元素按升序排列）。

例如，arr = [1,2,3] 的下一个排列是 [1,3,2] 。
类似地，arr = [2,3,1] 的下一个排列是 [3,1,2] 。
而 arr = [3,2,1] 的下一个排列是 [1,2,3] ，因为 [3,2,1] 不存在一个字典序更大的排列。
给你一个整数数组 nums ，找出 nums 的下一个排列。

必须 原地 修改，只允许使用额外常数空间。

```
示例 1：

输入：nums = [1,2,3]
输出：[1,3,2]
```

```
/*
```

```
eg:
原数组：123654
目的：124356
算法：
1.先找到nums[i] > nums[i-1]:6 > 3,i->6
2.再将i后元素排序：456
3.找到第一个比nums[i-1]大的数：4
4.与nums[i-1]交换：124356
*/
class Solution {
    public void nextPermutation(int[] nums) {
        int len = nums.length;
        for (int i = len - 1; i > 0; i--) {
            //从后往前先找出第一个相邻的后一个大于前一个情况，此时的i-1位置就是需要交换的位置
            if (nums[i] > nums[i - 1]) {
                //对i自己和之后的元素排序，[i,len)从小到大，第一个大于i-1位置的进行交换，那么就是下一个排列
                Arrays.sort(nums, i, len);
                for (int j = i; j <len; j++) {
                    if (nums[j] > nums[i - 1]) {
                        int temp = nums[j];
                        nums[j] = nums[i - 1];
                        nums[i - 1] = temp;
                        return;
                    }
                }
            }
        }
        Arrays.sort(nums);//都没有找到，则数组是按降序排序的，返回升序排序
        return;
    }
}
```

2.16 最长有效括号

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

```
示例 1:

输入: s = "(()"
输出: 2
解释: 最长有效括号子串是 "()"
```

```
//栈、遍历一次
/*
具体做法是我们始终保持栈底元素为当前已经遍历过的元素中「最后一个没有被匹配的右括号的下标」，这样的做法主要是考虑了边界条件的处理，栈里其他元素维护左括号的下标：
    对于遇到的每个‘(’，我们将它的下标放入栈中
    对于遇到的每个‘)’，我们先弹出栈顶元素表示匹配了当前右括号：
        如果栈为空，说明当前的右括号为没有被匹配的右括号，将其下标放入栈中来更新之前提到的「最后一个没有被匹配的右括号的下标」
        如果栈不为空，当前右括号的下标减去栈顶元素即为「以该右括号为结尾的最长有效括号的长度」
```

需要注意的是，如果一开始栈为空，第一个字符为左括号的时候我们会将其放入栈中，这样就不满足提及的「最后一个没有被匹配的右括号的下标」，为了保持统一，我们在一开始的时候往栈中放入一个值为 −1 的元素。
*/
```
class Solution {
    public int longestValidParentheses(String s) {
        int maxans = 0;
        Deque<Integer> stack = new LinkedList<Integer>();
        stack.push(-1);
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                stack.push(i);
            } else {
                stack.pop();
                if (stack.isEmpty()) {
                    stack.push(i);
                } else {
                    maxans = Math.max(maxans, i - stack.peek());
                }
            }
        }
        return maxans;
    }
}
```

```
//正逆向结合，两次遍历，空间O(1)
/*
在此方法中，我们利用两个计数器left和right 。首先，我们从左到右遍历字符串，对于遇到的每个‘(’，增加left 计数器，对于遇到的每个 ‘)’，我们增加right 计数器。每当left 计数器与right 计数器相等时，我们计算当前有效字符串的长度，并且记录目前为止找到的最长子字符串。当right计数器比left计数器大时，我们将left和right计数器同时变回0。
这样的做法贪心地考虑了以当前字符下标结尾的有效括号长度，每次当右括号数量多于左括号数量的时候之前的字符我们都扔掉不再考虑，重新从下一个字符开始计算，但这样会漏掉一种情况，就是遍历的时候左括号的数量始终大于右括号的数量，即 (() ，这种时候最长有效括号是求不出来的。
解决的方法也很简单，我们只需要从右往左遍历用类似的方法计算即可，只是这个时候判断条件反了过来：
当left计数器比right计数器大时，我们将left和right计数器同时变回 0
当left计数器与right计数器相等时，我们计算当前有效字符串的长度，并且记录目前为止找到的最长子字符串
*/
class Solution {
    public int longestValidParentheses(String s) {
        int left = 0, right = 0, maxlength = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                left++;
            } else {
                right++;
            }
            if (left == right) {
                maxlength = Math.max(maxlength, 2 * right);
            } else if (right > left) {
                left = right = 0;
            }
        }
        left = right = 0;
        for (int i = s.length() - 1; i >= 0; i--) {
            if (s.charAt(i) == '(') {
```

```
                left++;
            } else {
                right++;
            }
            if (left == right) {
                maxlength = Math.max(maxlength, 2 * left);
            } else if (left > right) {
                left = right = 0;
            }
        }
        return maxlength;
    }
}
```

2.17 搜索旋转排序数组

整数数组 nums 按升序排列，数组中的值 互不相同 。

在传递给函数之前，nums 在预先未知的某个下标 k（0 <= k < nums.length）上进行了 旋转，使数组变为 [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]（下标 从 0 开始 计数）。例如， [0,1,2,4,5,6,7] 在下标 3 处经旋转后可能变为 [4,5,6,7,0,1,2] 。

给你 旋转后 的数组 nums 和一个整数 target ，如果 nums 中存在这个目标值 target ，则返回它的下标，否则返回 -1 。

```
示例 1：

输入：nums = [4,5,6,7,0,1,2], target = 0
输出：4
```

```
//二分法
class Solution {
    public int search(int[] nums, int target) {
        int n = nums.length;
        if (n == 0) {
            return -1;
        }
        if (n == 1) {
            return nums[0] == target ? 0 : -1;
        }
        int l = 0, r = n - 1;
        while (l <= r) {
            int mid = (l + r) / 2;
            if (nums[mid] == target) {
                return mid;
            }
            if (nums[0] <= nums[mid]) {
                if (nums[0] <= target && target < nums[mid]) {
                    r = mid - 1;
                } else {
                    l = mid + 1;
                }
            } else {
                if (nums[mid] < target && target <= nums[n - 1]) {
                    l = mid + 1;
```

```
                } else {
                    r = mid - 1;
                }
            }
        }
        return -1;
    }
}
```

2.18 在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 target，返回 [-1, -1]。

```
示例 1：

输入：nums = [5,7,7,8,8,10], target = 8
输出：[3,4]
```

```
//二分法
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int leftIdx = binarySearch(nums, target, true);
        int rightIdx = binarySearch(nums, target, false) - 1;
        if (leftIdx <= rightIdx && rightIdx < nums.length && nums[leftIdx] ==
target && nums[rightIdx] == target) {
            return new int[]{leftIdx, rightIdx};
        }
        return new int[]{-1, -1};
    }

    public int binarySearch(int[] nums, int target, boolean lower) {
        int left = 0, right = nums.length - 1, ans = nums.length;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] > target || (lower && nums[mid] >= target)) {
                right = mid - 1;
                ans = mid;
            } else {
                left = mid + 1;
            }
        }
        return ans;
    }
}
```

2.19 组合总和

给你一个 无重复元素 的整数数组 candidates 和一个目标整数 target ， 找出 candidates 中可以使数字和为目标数 target 的 所有 不同组合 ， 并以列表形式返回。你可以按 任意顺序 返回这些组合。

candidates 中的 同一个 数字可以 无限制重复被选取 。如果至少一个数字的被选数量不同，则两种组合是不同的。

对于给定的输入，保证和为 target 的不同组合数少于 150 个。

```
示例 1：

输入：candidates = [2,3,6,7], target = 7
输出：[[2,2,3],[7]]
解释：
2 和 3 可以形成一组候选，2 + 2 + 3 = 7 。注意 2 可以使用多次。
7 也是一个候选， 7 = 7 。
仅有这两种组合。
```

```java
//回溯法
/*
1、遍历数组中的每一个数字。
2、递归枚举每一个数字可以选多少次，递归过程中维护一个target变量。如果当前数字小于等于target，我们就将其加入我们的路径数组path中，相应的target减去当前数字的值。也就是说，每选一个分支，就减去所选分支的值。
3、当target == 0时，表示该选择方案是合法的，记录该方案，将其加入res数组中。
注：为了避免搜索过程中的重复方案，我们要去定义一个搜索起点，已经考虑过的数，以后的搜索中就不能出现，让我们的每次搜索都从当前起点往后搜索(包含当前起点)，直到搜索到数组末尾。这样我们人为规定了一个搜索顺序，就可以避免重复方案。
*/
class Solution {
    List<List<Integer>> res = new ArrayList<>(); //记录答案
    List<Integer> path = new ArrayList<>();  //记录路径

    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        dfs(candidates,0, target);
        return res;
    }
    public void dfs(int[] candidates, int start, int target) {
        if(target < 0) return ;
        if(target == 0){
            res.add(new ArrayList(path));
            return ;
        }
        for(int i = start; i < candidates.length; i++){
            if( candidates[i] <= target) {
                path.add(candidates[i]);
                dfs(candidates,i,target -  candidates[i]); // 因为可以重复使用，所以还是i
                path.remove(path.size()-1); //回溯，恢复现场
            }
        }
    }
}
```

2.20 接雨水

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1：

输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]
输出：6
解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

```
//双指针
/*
定理一：在某个位置i处，它能存的水，取决于它左右两边的最大值中较小的一个。
定理二：当我们从左往右处理到left下标时，左边的最大值left_max对它而言是可信的，但right_max对它而言是不可信的。
定理三：当我们从右往左处理到right下标时，右边的最大值right_max对它而言是可信的，但left_max对它而言是不可信的。
*/
public int trap(int[] height) {
    //left：从左往右处理的当前下标,right：从右往左处理的当前下标
    int left = 0, right = height.length - 1;
    int ans = 0;
    //left_max：左边的最大值，它是从左往右遍历找到的,right_max：右边的最大值，它是从右往左遍历找到的
    int left_max = 0, right_max = 0;
    while (left < right) {
        if (height[left] < height[right]) {
            if (height[left] >= left_max) {
                left_max = height[left];
            } else {
                ans += (left_max - height[left]);
            }
            ++left;
        } else {
            if (height[right] >= right_max) {
                right_max = height[right];
            } else {
                ans += (right_max - height[right]);
            }
            --right;
        }
    }
    return ans;
}
```

2.21 全排列

给定一个不含重复数字的数组 nums ， 返回其 所有可能的全排列 。你可以 按任意顺序 返回答案。

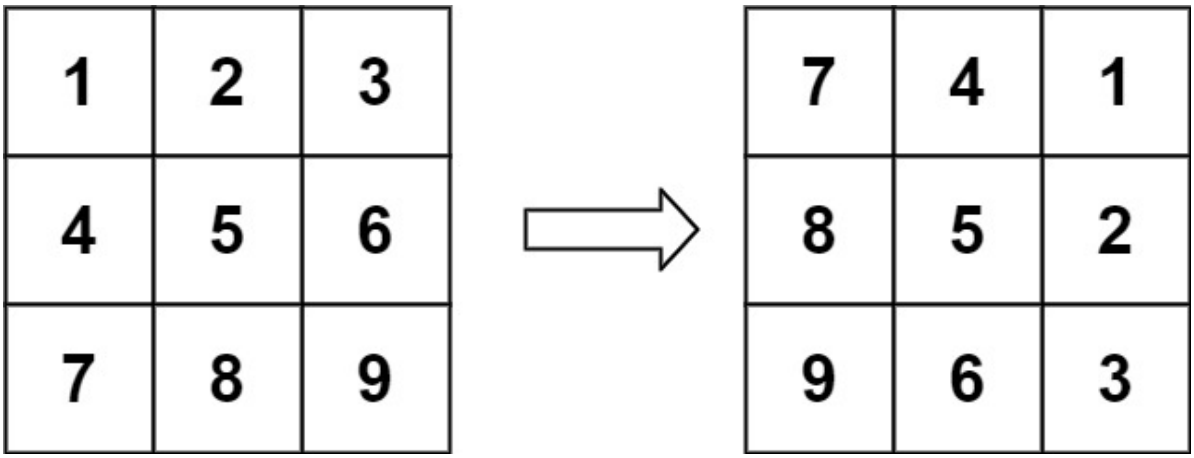
```
示例 1：

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

```java
//回溯法
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();

        List<Integer> output = new ArrayList<Integer>();
        for (int num : nums) {
            output.add(num);
        }

        int n = nums.length;
        backtrack(n, output, res, 0);
        return res;
    }

    public void backtrack(int n, List<Integer> output, List<List<Integer>> res,
int first) {
        // 所有数都填完了
        if (first == n) {
            res.add(new ArrayList<Integer>(output));
        }
        for (int i = first; i < n; i++) {
            // 动态维护数组
            Collections.swap(output, first, i);
            // 继续递归填下一个数
            backtrack(n, output, res, first + 1);
            // 撤销操作
            Collections.swap(output, first, i);
        }
    }
}
```

2.22 旋转图像

给定一个 n × n 的二维矩阵 matrix 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要 使用另一个矩阵来旋转图像。



示例 1：

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

```
//原地旋转
class Solution {
    public void rotate(int[][] matrix) {
        int n = matrix.length;
        for (int i = 0; i < n / 2; ++i) {//保证了偶数旋转(n/2)×(n/2) 个位置，奇数旋转((n-1)/2)×((n+1)/2) 个位置
            for (int j = 0; j < (n + 1) / 2; ++j) {
                int temp = matrix[i][j];                    //只需要使用一个临时变量存储一个值
                matrix[i][j] = matrix[n - j - 1][i];        //推导
                matrix[n - j - 1][i] = matrix[n - i - 1][n - j - 1];
                matrix[n - i - 1][n - j - 1] = matrix[j][n - i - 1];
                matrix[j][n - i - 1] = temp;
            }
        }
    }
}
```

2.23 字母异位词分组

给你一个字符串数组，请你将 字母异位词 组合在一起。可以按任意顺序返回结果列表。

字母异位词 是由重新排列源单词的字母得到的一个新单词，所有源单词中的字母通常恰好只用一次。

```
示例 1:

输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
输出: [["bat"],["nat","tan"],["ate","eat","tea"]]
```

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        Map<String, List<String>> map = new HashMap<String, List<String>>();
        for (String str : strs) {
```

```
            char[] array = str.toCharArray();
            Arrays.sort(array);
            String key = new String(array);
            //hashmap.getOrDefault(Object key, V defaultValue)
            //获取指定key对应对value，如果找不到key，则返回设置的默认值
            List<String> list = map.getOrDefault(key, new ArrayList<String>());
            list.add(str);
            map.put(key, list);
        }
        return new ArrayList<List<String>>(map.values());
    }
}
```

2.24 最大子数组和

给你一个整数数组 nums ， 请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

子数组 是数组中的一个连续部分。

```
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 。
```

```
//DP
class Solution {
    public int maxSubArray(int[] nums) {
        int pre = 0,maxSum = nums[0];
        for(int x : nums){
            pre = Math.max(pre + x,x);
            maxSum = Math.max(pre,maxSum);
        }
        return maxSum;
    }
}
```

2.25 跳跃游戏

给定一个非负整数数组 nums ， 你最初位于数组的 第一个下标 。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标。

示例 1：

输入：nums = [2,3,1,1,4]
输出：true
解释：可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。

示例 2：

输入：nums = [3,2,1,0,4]
输出：false
解释：无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0 ， 所以永远不可能到达最后一个下标。

```
//贪心算法
class Solution {
    public boolean canJump(int[] nums) {
        int max = nums[0],len = nums.length;
        for(int i = 0;i < len;i++){
            if(max >= i){
                max = Math.max(max,i + nums[i]);
                if(max >= len - 1)
                    return true;
            }
        }
        return false;
    }
}
```

2.26 合并区间

以数组 intervals 表示若干个区间的集合，其中单个区间为 intervals[i] = [starti, endi] 。请你合并所有重叠的区间，并返回 一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间 。

示例 1：

输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
输出：[[1,6],[8,10],[15,18]]
解释：区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].

```
//排序
//对每个区间左端点升序排序
class Solution {
    public int[][] merge(int[][] intervals) {
        if (intervals.length == 0) {
            return new int[0][2];
        }
        Arrays.sort(intervals, new Comparator<int[]>() {    //重载compare，使排序规则为以数组第一列升序排序
            public int compare(int[] interval1, int[] interval2) {
                return interval1[0] - interval2[0];
            }
        });
        List<int[]> merged = new ArrayList<int[]>();
```

```
        for (int i = 0; i < intervals.length; ++i) {
            int L = intervals[i][0], R = intervals[i][1];
            //如果当前区间的左端点在数组 merged 中最后一个区间的右端点之后，那么它们不会重
合，我们可以直接将这个区间加入数组 merged 的末尾；
            if (merged.size() == 0 || merged.get(merged.size() - 1)[1] < L) {
                merged.add(new int[]{L, R});
            //否则，它们重合，我们需要用当前区间的右端点更新数组 merged 中最后一个区间的右端
点，将其置为二者的较大值。
            } else {
                merged.get(merged.size() - 1)[1] =
Math.max(merged.get(merged.size() - 1)[1], R);
            }
        }
        return merged.toArray(new int[merged.size()][]);
    }
}
```

2.27 不同路径

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为 “Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径?


```
示例 1:


输入: m = 3, n = 7
输出: 28
```

```
//DP
//每个位置的路径 = 该位置左边的路径 + 该位置上边的路径
class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];
        for (int i = 0; i < n; i++) dp[0][i] = 1;   //边界只能往右或者下，只有一种可能
        for (int i = 0; i < m; i++) dp[i][0] = 1;
        for (int i = 1; i < m; i++) {       //状态方程: dp[i][j] = dp[i-1][j] +
dp[i][j-1]
            for (int j = 1; j < n; j++) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
```

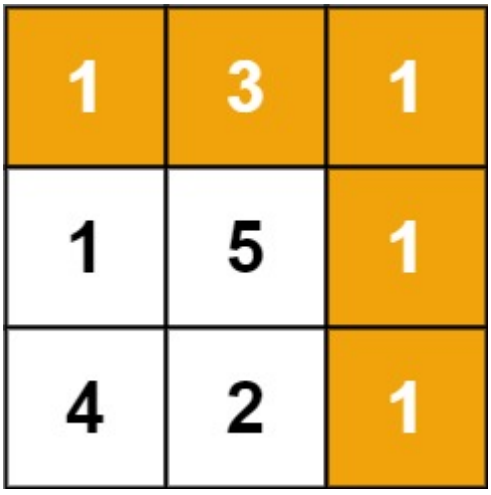
```
        return dp[m - 1][n - 1];
    }
}
```

```
//优化版本，空间复杂度O(n)
class Solution {
    public int uniquePaths(int m, int n) {
        int[] cur = new int[n];     //只需使用列数的一维数组
        Arrays.fill(cur,1);
        for (int i = 1; i < m;i++){
            for (int j = 1; j < n; j++){
                //cur[j] = cur[j] + cur[j-1]:
                /*
                *cur[j]:以前未更新的路径数，即上一次i循环（上一行）当前列的路径
                *cur[j-1]:更新后，即本行左边一个的路径
                *即左边路径+上边路径，妙啊！
                */
                cur[j] += cur[j-1] ;
            }
        }
        return cur[n-1];
    }
}
```

2.28 最小路径和

给定一个包含非负整数的 m x n 网格 grid ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。




```
示例 1:

输入: grid = [[1,3,1],[1,5,1],[4,2,1]]
输出: 7
解释: 因为路径 1→3→1→1→1 的总和最小。
```

```
class Solution {
    public int minPathSum(int[][] grid) {
        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
                if(i == 0 && j == 0) continue;
                else if(i == 0)  grid[i][j] = grid[i][j - 1] + grid[i][j];
                else if(j == 0)  grid[i][j] = grid[i - 1][j] + grid[i][j];
                else grid[i][j] = Math.min(grid[i - 1][j], grid[i][j - 1]) +
grid[i][j];
            }
        }
        return grid[grid.length - 1][grid[0].length - 1];
    }
}
```

2.29 爬楼梯

同剑指offer8题，青蛙跳台阶，dp算法

2.30 颜色分类

给定一个包含红色、白色和蓝色、共 n 个元素的数组 nums ，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

必须在不使用库的sort函数的情况下解决这个问题。

```
示例 1：

输入：nums = [2,0,2,1,1,0]
输出：[0,0,1,1,2,2]
```

```
//双指针
//0 挪到最前面，2 挪到最后面
//注意 2 挪完如果换出来的不是 1，那么指针要回退，因为 0 和 2 都是需要再次移动的
class Solution {
    public void sortColors(int[] nums) {
        int n = nums.length;
        if (n < 2) {
            return;
        }
        //双指针
        int p = 0, q = n - 1;
        for (int i = 0; i <= q; ++i) {
            if (nums[i] == 0) {
                nums[i] = nums[p];
                nums[p] = 0;
                ++p;
            }
            if (nums[i] == 2) {
                nums[i] = nums[q];
                nums[q] = 2;
```

```
                --q;
                if (nums[i] != 1) {
                    --i;
                }
            }
        }
        return;
    }
}
```

2.31 最小覆盖子串

给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "" 。

注意：

对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。
如果 s 中存在这样的子串，我们保证它是唯一的答案。

示例 1：

输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"

```
//滑动窗口
class Solution {
    public String minWindow(String s, String t) {
        if (s == null || s.length() == 0 || t == null || t.length() == 0){
            return "";
        }
        int[] need = new int[128];
        //记录需要的字符的个数
        for (int i = 0; i < t.length(); i++) {
            need[t.charAt(i)]++;
        }
        //l是当前左边界，r是当前右边界，size记录窗口大小，count是需求的字符个数，start是最小覆盖串开始的index
        int l = 0, r = 0, size = Integer.MAX_VALUE, count = t.length(), start = 0;
        //遍历所有字符
        while (r < s.length()) {
            char c = s.charAt(r);
            if (need[c] > 0) {//需要字符c
                count--;
            }
            need[c]--;//把右边的字符加入窗口
            if (count == 0) {//窗口中已经包含所有字符
                while (l < r && need[s.charAt(l)] < 0) {
                    need[s.charAt(l)]++;//释放右边移动出窗口的字符
                    l++;//指针右移
                }
                if (r - l + 1 < size) {//不能右移时候挑战最小窗口大小，更新最小窗口开始的start
                    size = r - l + 1;
```

```
                    start = l;//记录下最小值时候的开始位置，最后返回覆盖串时候会用到
                }
                //释放左边移动出窗口的字符
                need[s.charAt(l)]++;
                l++;
                count++;
            }
            r++;
        }
        return size == Integer.MAX_VALUE ? "" : s.substring(start, start +
size);
    }
}
```

2.32 子集

给你一个整数数组 nums ，数组中的元素 互不相同 。返回该数组所有可能的子集（幂集）。

解集 不能 包含重复的子集。你可以按 任意顺序 返回解集。

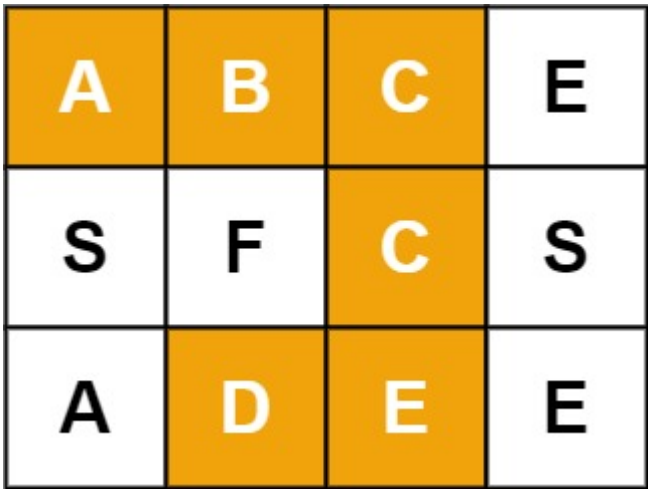
```
示例 1：

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

```
//回溯法
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        backtrack(0, nums, res, new ArrayList<Integer>());
        return res;

    }

    private void backtrack(int i, int[] nums, List<List<Integer>> res,
ArrayList<Integer> tmp) {
        res.add(new ArrayList<>(tmp));
        for (int j = i; j < nums.length; j++) {
            tmp.add(nums[j]);
            backtrack(j + 1, nums, res, tmp);
            tmp.remove(tmp.size() - 1);
        }
    }
}
```

2.33 单词搜索

给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true ；否则，返回 false 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。



示例 1：

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

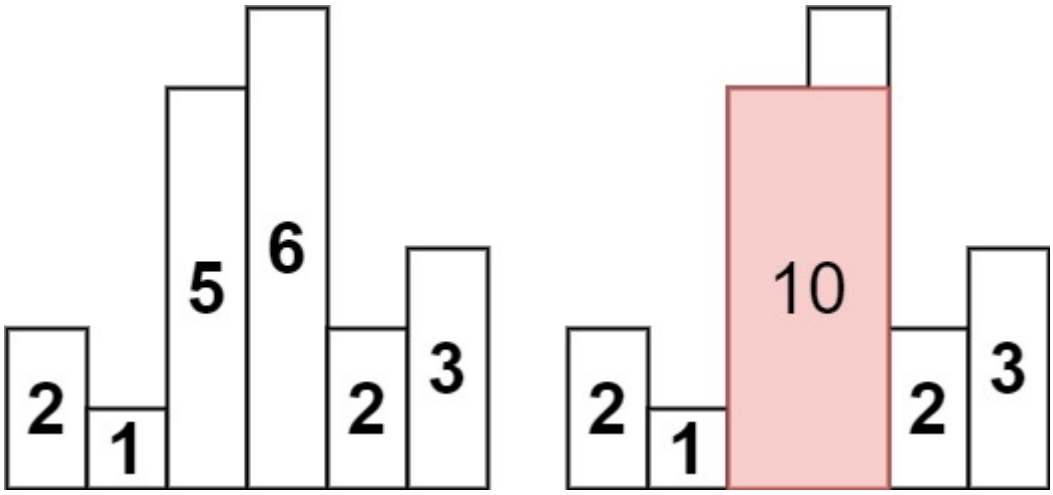
```
//回溯法
public class Solution {

    private static final int[][] DIRECTIONS = {{-1, 0}, {0, -1}, {0, 1}, {1, 0}};
    private int rows;
    private int cols;
    private int len;
    private boolean[][] visited;
    private char[] charArray;
    private char[][] board;

    public boolean exist(char[][] board, String word) {
        rows = board.length;
        if (rows == 0) {
            return false;
        }
        cols = board[0].length;
        visited = new boolean[rows][cols];

        this.len = word.length();
        this.charArray = word.toCharArray();
        this.board = board;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (dfs(i, j, 0)) {
                    return true;
                }
            }
        }
        return false;
    }

    private boolean dfs(int x, int y, int begin) {
        if (begin == len - 1) {
            return board[x][y] == charArray[begin];
```

```
        }
        if (board[x][y] == charArray[begin]) {
            visited[x][y] = true;
            for (int[] direction : DIRECTIONS) {
                int newX = x + direction[0];
                int newY = y + direction[1];
                if (inArea(newX, newY) && !visited[newX][newY]) {
                    if (dfs(newX, newY, begin + 1)) {
                        return true;
                    }
                }
            }
            visited[x][y] = false;
        }
        return false;
    }

    private boolean inArea(int x, int y) {
        return x >= 0 && x < rows && y >= 0 && y < cols;
    }
}
```

2.34 柱状图中最大的矩形

给定 *n* 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



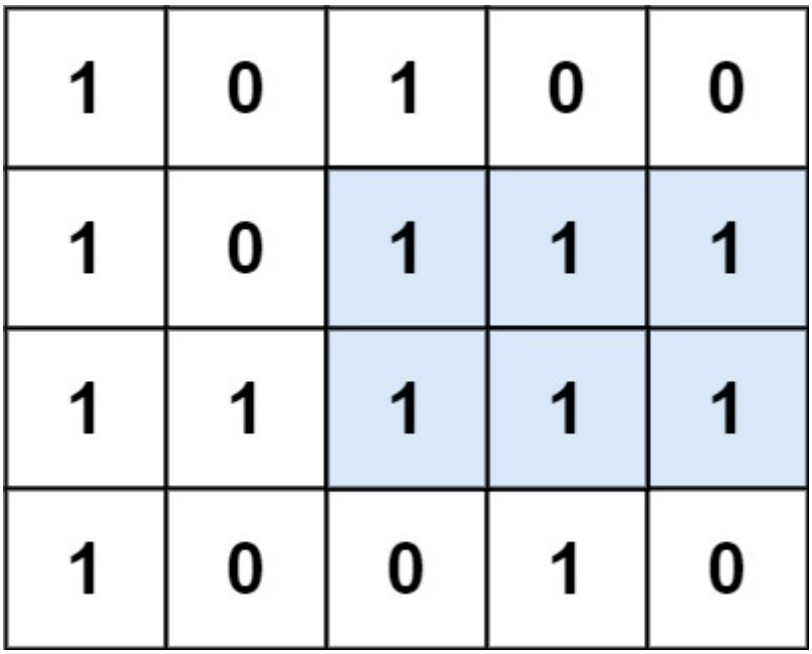

```
输入：heights = [2,1,5,6,2,3]
输出：10
解释：最大的矩形为图中红色区域，面积为 10
```

```
//单调栈
/*
栈内的元素是递增的
当元素出栈时，说明这个新元素是出栈元素向后找第一个比其小的元素
当元素出栈后，说明新栈顶元素是出栈元素向前找第一个比其小的元素
注：前后添0，强制弹出所有元素
*/
class Solution {
    public int largestRectangleArea(int[] heights) {
```

```
        int res = 0;
        Deque<Integer> stack = new ArrayDeque<>();
        int[] new_heights = new int[heights.length + 2];
        for (int i = 1; i < heights.length + 1; i++) {  //前后添0
            new_heights[i] = heights[i - 1];
        }
        for (int i = 0; i < new_heights.length; i++) {
            while (!stack.isEmpty() && new_heights[stack.peek()] >
new_heights[i]) {
                int cur = stack.pop();  //当前
                int l = stack.peek();   //左边
                int r = i;              //右边
                res = Math.max(res, (r - l - 1) * new_heights[cur]);    //计算面积
            }
            stack.push(i);
        }
        return res;
    }
}
```

2.35 最大矩形

给定一个仅包含 `0` 和 `1` 、大小为 `rows x cols` 的二维二进制矩阵，找出只包含 `1` 的最大矩形，并返回其面积。



输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
输出：6
解释：最大矩形如上图所示。

```
//单调栈，和上一题解法一样，可以抽象为柱状图中矩形
class Solution {
    public int maximalRectangle(char[][] matrix) {
        if (matrix.length == 0) {
            return 0;
        }
        int[] heights = new int[matrix[0].length];
        int maxArea = 0;
```

```
        for (int row = 0; row < matrix.length; row++) {
            //遍历每一列，更新高度
            for (int col = 0; col < matrix[0].length; col++) {
                if (matrix[row][col] == '1') {
                    heights[col] += 1;
                } else {
                    heights[col] = 0;
                }
            }
            //调用上一题的解法，更新函数
            maxArea = Math.max(maxArea, largestRectangleArea(heights));
        }
        return maxArea;
    }

    public int largestRectangleArea(int[] heights) {
        int maxArea = 0;
        Stack<Integer> stack = new Stack<>();
        int p = 0;
        while (p < heights.length) {
            //栈空入栈
            if (stack.isEmpty()) {
                stack.push(p);
                p++;
            } else {
                int top = stack.peek();
                //当前高度大于栈顶，入栈
                if (heights[p] >= heights[top]) {
                    stack.push(p);
                    p++;
                } else {
                    //保存栈顶高度
                    int height = heights[stack.pop()];
                    //左边第一个小于当前柱子的下标
                    int leftLessMin = stack.isEmpty() ? -1 : stack.peek();
                    //右边第一个小于当前柱子的下标
                    int RightLessMin = p;
                    //计算面积
                    int area = (RightLessMin - leftLessMin - 1) * height;
                    maxArea = Math.max(area, maxArea);
                }
            }
        }
        while (!stack.isEmpty()) {
            //保存栈顶高度
            int height = heights[stack.pop()];
            //左边第一个小于当前柱子的下标
            int leftLessMin = stack.isEmpty() ? -1 : stack.peek();
            //右边没有小于当前高度的柱子，所以赋值为数组的长度便于计算
            int RightLessMin = heights.length;
            int area = (RightLessMin - leftLessMin - 1) * height;
            maxArea = Math.max(area, maxArea);
        }
        return maxArea;
    }
}
```

2.36 二叉树的中序遍历

给定一个二叉树的根节点 `root` ，返回它的 **中序** 遍历。

```
//递归
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        inorder(root,res);
        return res;
    }

    void inorder(TreeNode root,List<Integer> res){
        if(root == null) return;
        inorder(root.left,res);
        res.add(root.val);
        inorder(root.right,res);
    }
}
```

```
//迭代
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        Deque<TreeNode> stk = new LinkedList<TreeNode>();
        while (root != null || !stk.isEmpty()) {
            while (root != null) {
                stk.push(root);
                root = root.left;
            }
            root = stk.pop();
            res.add(root.val);
            root = root.right;  //下次循环入栈
        }
        return res;
    }
}
```

2.37 不同的二叉搜索树

给你一个整数 `n` ，求恰由 `n` 个节点组成且节点值从 `1` 到 `n` 互不相同的 **二叉搜索树** 有多少种？返回满足题意的二叉搜索树的种数。

```
/*
假设 n 个节点存在二叉排序树的个数是 G (n)，令 f(i) 为以 i 为根的二叉搜索树的个数，则
G(n) = f(1) + f(2) + f(3) + f(4) + ... + f(n)
当 i 为根节点时，其左子树节点个数为 i-1 个，右子树节点为 n-i，则
f(i) = G(i-1)*G(n-i)
综合两个公式可以得到 卡特兰数 公式
G(n) = G(0)*G(n-1)+G(1)*(n-2)+...+G(n-1)*G(0)
*/
//DP
```

```
class Solution {
    public int numTrees(int n) {
        int[] dp = new int[n+1];
        dp[0] = 1;
        dp[1] = 1;

        for(int i = 2; i < n + 1; i++)
            for(int j = 1; j < i + 1; j++)
                dp[i] += dp[j-1] * dp[i-j];

        return dp[n];
    }
}
```

2.38 验证二叉搜索树

给你一个二叉树的根节点 `root` ，判断其是否是一个有效的二叉搜索树。

```
//中序遍历
class Solution {
    long pre = Long.MIN_VALUE;
    public boolean isValidBST(TreeNode root) {
        if (root == null) return true;

        // 访问左子树
        if (!isValidBST(root.left))  return false;

        // 访问当前节点：如果当前节点小于等于中序遍历的前一个节点，说明不满足BST，返回
false；否则继续遍历。
        if (root.val <= pre) return false;

        pre = root.val;
        // 访问右子树
        return isValidBST(root.right);
    }
}
```

2.39 对称二叉树

给你一个二叉树的根节点 `root` ， 检查它是否轴对称。

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

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

```
    boolean dfs(TreeNode left, TreeNode right) {
        //递归的终止条件是两个节点都为空
        //或者两个节点中有一个为空
        //或者两个节点的值不相等
        if(left==null && right==null) {
            return true;
        }
        if(left==null || right==null) {
            return false;
        }
        if(left.val != right.val) {
            return false;
        }
        //再递归的比较 左节点的左孩子 和 右节点的右孩子
        //以及比较  左节点的右孩子 和 右节点的左孩子
        return dfs(left.left,right.right) && dfs(left.right,right.left);
    }
}
```

2.40 二叉树的层序遍历

给你二叉树的根节点 `root` ，返回其节点值的 **层序遍历** 。 （即逐层地，从左到右访问所有节点）。

```
//BFS
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        List<List<Integer>> res = new ArrayList<>();
        TreeNode cur;
        int len;
        if(root == null) return res;

        queue.add(root);
        while(!queue.isEmpty()){
            List<Integer> level = new ArrayList<>();
            len = queue.size();          //队列大小在变，需要变量保存
            for(int i=0;i<len;i++){      //每一层需要循环操作
                cur = queue.poll();
                level.add(cur.val);
                if(cur.left != null) queue.add(cur.left);
                if(cur.right != null) queue.add(cur.right);
            }
            res.add(level);
        }
        return res;
    }
}
```

2.41 二叉树的最大深度

给定一个二叉树，找出其最大深度。

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

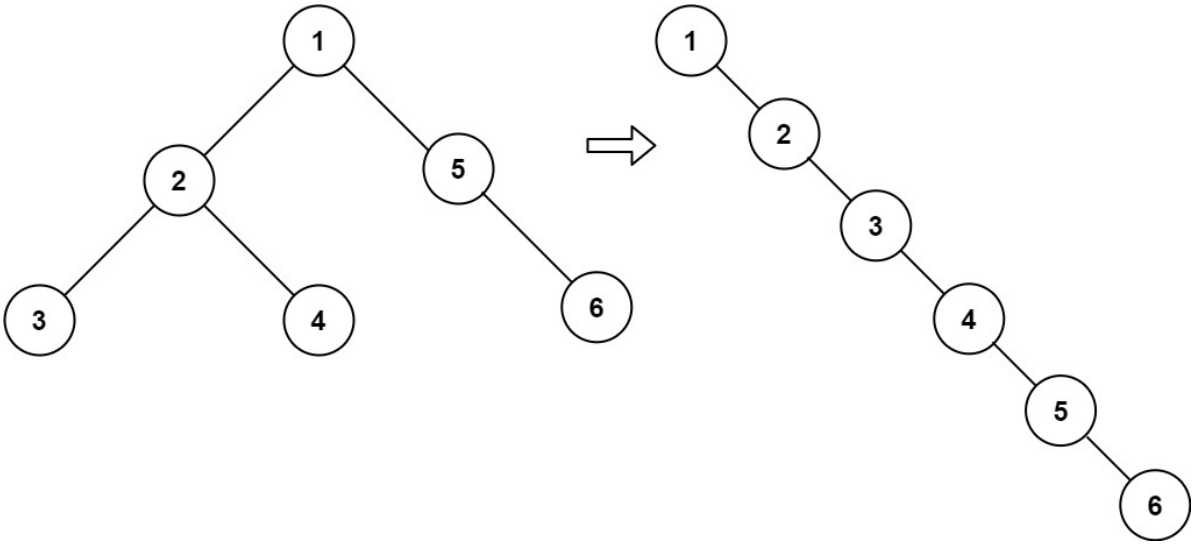
2.42 从前序与中序遍历序列构造二叉树

同剑指offer第5题

2.43 二叉树展开为链表

给你二叉树的根结点 root ，请你将它展开为一个单链表：

展开后的单链表应该同样使用 TreeNode ，其中 right 子指针指向链表中下一个结点，而左子指针始终为 null 。
展开后的单链表应该与二叉树 先序遍历 顺序相同。




```
    1
   / \
  2   5
 / \   \
3   4   6

//将 1 的左子树插入到右子树的地方
    1
     \
      2         5
     / \         \
    3   4         6
```

```
//将原来的右子树接到左子树的最右边节点
    1
     \
      2
     / \
    3   4
         \
          5
           \
            6

 //将 2 的左子树插入到右子树的地方
    1
     \
      2
       \
        3       4
                 \
                  5
                   \
                    6

 //将原来的右子树接到左子树的最右边节点
    1
     \
      2
       \
        3
         \
          4
           \
            5
             \
              6


  ......
```

```
class Solution {
    public void flatten(TreeNode root) {
        while(root != null){
            //左子树为空，直接考虑右子树
            if(root.left == null) root = root.right;
            else{
                //找左子树最右边叶子节点
                TreeNode tmp = root.left;
                while(tmp.right != null){
                    tmp = tmp.right;
                }
                //将原来的右子树插入左子树最右边节点
                tmp.right = root.right;
                //将左子树插到原来右子树地方
                root.right = root.left;
                root.left = null;   //记得置null
                //考虑下一个节点
                root = root.right;
            }
        }
```

```
    }
}
```

2.44 买卖股票的最佳时机

给定一个数组 prices ，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。

你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。

```
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；同时，你不能在买入前卖出股票。
```

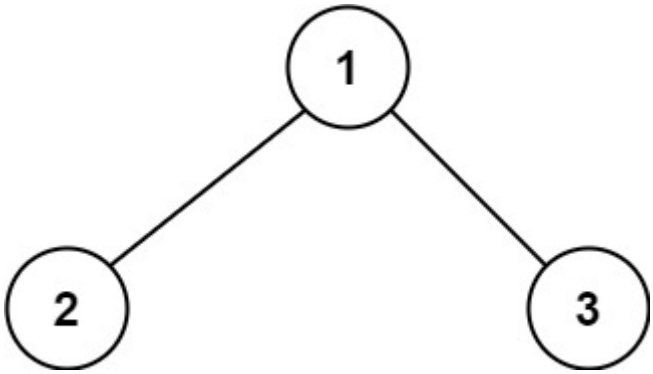
```
//使用一个值记录当前最小值，每天和最小值计算利润，保存最大的利润，最后返回
class Solution {
    public int maxProfit(int[] prices) {
        int minPrice = Integer.MAX_VALUE,maxProfit = 0;
        for(int x:prices){
            if(minPrice > x) minPrice = x;
            if((x - minPrice) > maxProfit) maxProfit = x - minPrice;
        }
        return maxProfit;
    }
}
```

2.45 二叉树中的最大路径和

路径 被定义为一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列。同一个节点在一条路径序列中 至多出现一次 。该路径 至少包含一个 节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。

给你一个二叉树的根节点 root ，返回其 最大路径和 。



```
输入：root = [1,2,3]
输出：6
解释：最优路径是 2 -> 1 -> 3 ，路径和为 2 + 1 + 3 = 6
```

```
class Solution {
    int maxSum = Integer.MIN_VALUE;

    public int maxPathSum(TreeNode root) {
        maxGain(root);
        return maxSum;
    }

    public int maxGain(TreeNode node) {
        if (node == null) {
            return 0;
        }
        // 递归计算左右子节点的最大贡献值
        // 只有在最大贡献值大于 0 时，才会选取对应子节点
        int leftGain = Math.max(maxGain(node.left), 0);
        int rightGain = Math.max(maxGain(node.right), 0);

        // 节点的最大路径和取决于该节点的值与该节点的左右子节点的最大贡献值
        int priceNewpath = node.val + leftGain + rightGain;

        // 更新答案
        maxSum = Math.max(maxSum, priceNewpath);    //始终维护这个全局变量

        // 返回节点的最大贡献值
        return node.val + Math.max(leftGain, rightGain);    //二叉树中，从根出发，只能选择走右边还是左边
    }
}
```

2.46 最长连续序列

给定一个未排序的整数数组 nums ， 找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

请你设计并实现时间复杂度为 O(n) 的算法解决此问题。

```
示例 1：

输入：nums = [100,4,200,1,3,2]
输出：4
解释：最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。
```

```
//使用Hash集合
class Solution {
    public int longestConsecutive(int[] nums) {
        Set<Integer> set = new HashSet<>();
        for(int x:nums){
            set.add(x);
        }
```

```
        int longest = 0,curNum,curLen = 0;
        for(int x:nums){
            if(!set.contains(x-1)){     //关键
                curNum = x;
                curLen = 1;
                while(set.contains(curNum+1)){
                    curNum++;
                    curLen++;
                }
            }
            longest = Math.max(longest,curLen);
        }
        return longest;
    }
}
```

2.47 只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

```
示例 1:

输入: [2,2,1]
输出: 1
```

```
//异或
//异或运算满足交换律，a^b^a=a^a^b=b,因此res相当于
nums[0]^nums[1]^nums[2]^nums[3]^nums[4]..... 然后再根据交换律把相等的合并到一块儿进行异或（结果为0），然后再与只出现过一次的元素进行异或，这样最后的结果就是，只出现过一次的元素（0^任意值=任意值）
class Solution {
    public int singleNumber(int[] nums) {
        int res = 0;
        for(int x:nums){
            res ^= x;
        }
        return res;
    }
}
```

2.48 单词拆分

给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典。请你判断是否可以利用字典中出现的单词拼接出 `s` 。

注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

```
示例 1：

输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以由 "leet" 和 "code" 拼接成。
```

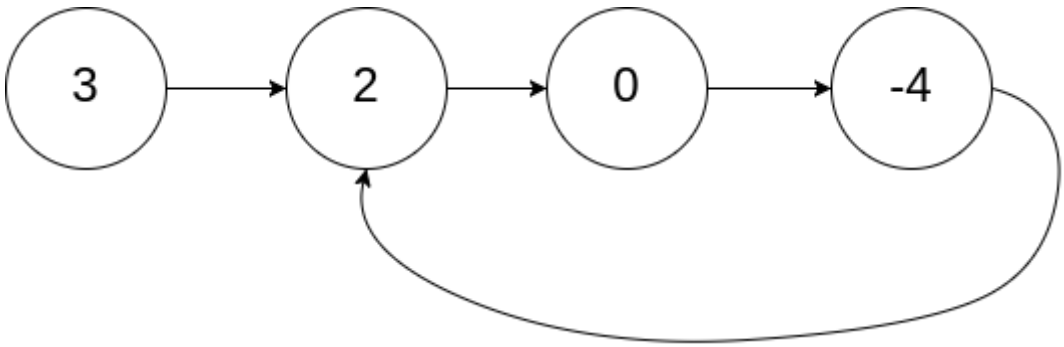
```java
//DP
//定义dp[i] 表示字符串s前i个字符组成的字符串 s[0..i-1] 是否能被空格拆分成若干个字典中出现的单词
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        int len = s.length(), maxw = 0;
        boolean[] dp = new boolean[len + 1];
        dp[0] = true;
        Set<String> set = new HashSet();
        for(String str : wordDict){
            set.add(str);
            maxw = Math.max(maxw, str.length());
        }
        for(int i = 1; i < len + 1; i++){
            for(int j = i; j >= 0 && j >= i - maxw; j--){
                if(dp[j] && set.contains(s.substring(j, i))){   //dp[i]只需要往前探索到词典里最长的单词即可
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[len];
    }
}
```

2.49 环形链表

给你一个链表的头节点 head ， 判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。 为了表示给定链表中的环，评测系统内部使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。注意：pos 不作为参数进行传递 。仅仅是为了标识链表的实际情况。

如果链表中存在环 ， 则返回 true 。 否则，返回 false 。



```
输入：head = [3,2,0,-4], pos = 1
输出：true
解释：链表中有一个环，其尾部连接到第二个节点。
```

```
//快慢指针
public boolean hasCycle(ListNode head) {
    ListNode fast = head, slow = head;
    while(fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
        if(fast == slow) { // 在指针移动后再比较，排除初始都指向头结点的情况
            return true;
        }
    }
    return false;
}
```

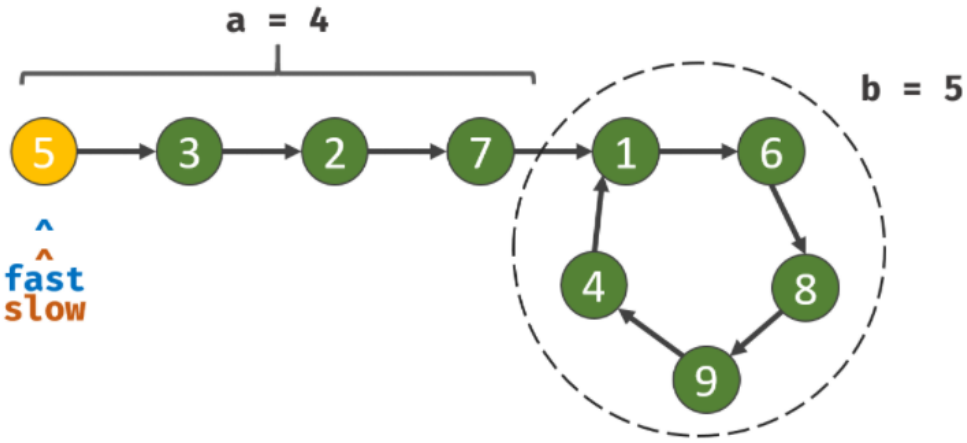
2.50 环形链表 II

给定一个链表的头节点 head ，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。

如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。 为了表示给定链表中的环，评测系统内部使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 pos 是 -1，则在该链表中没有环。注意：pos 不作为参数进行传递，仅仅是为了标识链表的实际情况。

不允许修改 链表。

```
输入：head = [3,2,0,-4], pos = 1
输出：返回索引为 1 的链表节点
解释：链表中有一个环，其尾部连接到第二个节点。
```




```
//快慢指针
```

```java
/*
1.第一次相遇，slow = nb
2.a+nb = 入口点
3.slow再走a = 入口 = head走到入口 = a
4.由3得出，起始距离入口 = 第一次相遇位置 + a
*/
public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode fast = head, slow = head;
        while (true) {
            if (fast == null || fast.next == null) return null;
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) break;
        }
        fast = head;         //第一次相遇后，slow走了nb步，只需要再走a步即入口，此时让fast移动到head，再次相遇即走了a步
        while (slow != fast) {
            slow = slow.next;
            fast = fast.next;
        }
        return fast;
    }
}
```

2.51 LRU 缓存

请你设计并实现一个满足 LRU (最近最少使用) 缓存 约束的数据结构。
实现 LRUCache 类：
LRUCache(int capacity) 以 正整数 作为容量 capacity 初始化 LRU 缓存
int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
void put(int key, int value) 如果关键字 key 已经存在，则变更其数据值 value ；如果不存在，则向缓存中插入该组 key-value 。如果插入操作导致关键字数量超过 capacity ，则应该 逐出 最久未使用的关键字。
函数 get 和 put 必须以 O(1) 的平均时间复杂度运行。

```
示例：
输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
输出
[null, null, null, 1, null, -1, null, -1, 3, 4]

解释
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

```
//哈希表+双向链表  实现LinkedHashMap?
/*
对于 get 操作，首先判断 key 是否存在：
如果 key 不存在，则返回 -1-1；
如果 key 存在，则 key 对应的节点是最近被使用的节点。通过哈希表定位到该节点在双向链表中的位置，并将其移动到双向链表的头部，最后返回该节点的值。

对于 put 操作，首先判断 key 是否存在：
如果 key 不存在，使用 key 和 value 创建一个新的节点，在双向链表的头部添加该节点，并将 key 和该节点添加进哈希表中。然后判断双向链表的节点数是否超出容量，如果超出容量，则删除双向链表的尾部节点，并删除哈希表中对应的项；
如果 key 存在，则与 get 操作类似，先通过哈希表定位，再将对应的节点的值更新为 value，并将该节点移到双向链表的头部。
*/
public class LRUCache {
    class DLinkedNode {
        int key;
        int value;
        DLinkedNode prev;
        DLinkedNode next;
        public DLinkedNode() {}
        public DLinkedNode(int _key, int _value) {key = _key; value = _value;}
    }

    private Map<Integer, DLinkedNode> cache = new HashMap<Integer, DLinkedNode>
();
    private int size;
    private int capacity;
    private DLinkedNode head, tail;

    public LRUCache(int capacity) {
        this.size = 0;
        this.capacity = capacity;
        // 使用伪头部和伪尾部节点
        head = new DLinkedNode();
        tail = new DLinkedNode();
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        DLinkedNode node = cache.get(key);
        if (node == null) {
            return -1;
        }
        // 如果 key 存在，先通过哈希表定位，再移到头部
        moveToHead(node);
        return node.value;
    }

    public void put(int key, int value) {
        DLinkedNode node = cache.get(key);
```

```
        if (node == null) {
            // 如果 key 不存在，创建一个新的节点
            DLinkedNode newNode = new DLinkedNode(key, value);
            // 添加进哈希表
            cache.put(key, newNode);
            // 添加至双向链表的头部
            addToHead(newNode);
            ++size;
            if (size > capacity) {
                // 如果超出容量，删除双向链表的尾部节点
                DLinkedNode tail = removeTail();
                // 删除哈希表中对应的项
                cache.remove(tail.key);
                --size;
            }
        }
        else {
            // 如果 key 存在，先通过哈希表定位，再修改 value，并移到头部
            node.value = value;
            moveToHead(node);
        }
    }

    private void addToHead(DLinkedNode node) {
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;
    }

    private void removeNode(DLinkedNode node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private void moveToHead(DLinkedNode node) {
        removeNode(node);
        addToHead(node);
    }

    private DLinkedNode removeTail() {
        DLinkedNode res = tail.prev;
        removeNode(res);
        return res;
    }
}
```

2.52 排序链表

给你链表的头结点 `head` ，请将其按 **升序** 排列并返回 **排序后的链表** 。

```
class Solution {
     // 自底向上归并排序
    public ListNode sortList(ListNode head) {
```

```
        if(head == null){
            return head;
        }

        // 1. 首先从头向后遍历,统计链表长度
        int length = 0; // 用于统计链表长度
        ListNode node = head;
        while(node != null){
            length++;
            node = node.next;
        }

        // 2. 初始化 引入dummynode
        ListNode dummyHead = new ListNode(0);
        dummyHead.next = head;

        // 3. 每次将链表拆分成若干个长度为subLen的子链表 ，并按照每两个子链表一组进行合并
        for(int subLen = 1;subLen < length;subLen <<= 1){ // subLen每次左移一位（即
sublen = sublen*2） PS:位运算对CPU来说效率更高
            ListNode prev = dummyHead;
            ListNode curr = dummyHead.next;     // curr用于记录拆分链表的位置

            while(curr != null){                // 如果链表没有被拆完
                // 3.1 拆分subLen长度的链表1
                ListNode head_1 = curr;         // 第一个链表的头 即 curr初始的位置
                for(int i = 1; i < subLen && curr != null && curr.next != null;
i++){     // 拆分出长度为subLen的链表1
                    curr = curr.next;
                }

                // 3.2 拆分subLen长度的链表2
                ListNode head_2 = curr.next;  // 第二个链表的头  即 链表1尾部的下一个
位置
                curr.next = null;             // 断开第一个链表和第二个链表的链接
                curr = head_2;                // 第二个链表头 重新赋值给curr
                for(int i = 1;i < subLen && curr != null && curr.next !=
null;i++){      // 再拆分出长度为subLen的链表2
                    curr = curr.next;
                }

                // 3.3 再次断开 第二个链表最后的next的链接
                ListNode next = null;
                if(curr != null){
                    next = curr.next;   // next用于记录 拆分完两个链表的结束位置
                    curr.next = null;   // 断开连接
                }

                // 3.4 合并两个subLen长度的有序链表
                ListNode merged = mergeTwoLists(head_1,head_2);
                prev.next = merged;         // prev.next 指向排好序链表的头
                while(prev.next != null){  // while循环 将prev移动到 subLen*2 的位置
后去
                    prev = prev.next;
                }
                curr = next;               // next用于记录 拆分完两个链表的结束位置
            }
        }
        // 返回新排好序的链表
```

```
        return dummyHead.next;
    }


    // 此处是Leetcode21 --> 合并两个有序链表
    public ListNode mergeTwoLists(ListNode l1,ListNode l2){
        ListNode dummy = new ListNode(0);
        ListNode curr  = dummy;

        while(l1 != null && l2!= null){ // 退出循环的条件是走完了其中一个链表
            // 判断l1 和 l2大小
            if (l1.val < l2.val){
                // l1 小 ， curr指向l1
                curr.next = l1;
                l1 = l1.next;       // l1 向后走一位
            }else{
                // l2 小 ， curr指向l2
                curr.next = l2;
                l2 = l2.next;       // l2向后走一位
            }
            curr = curr.next;       // curr后移一位
        }

        // 退出while循环之后,比较哪个链表剩下长度更长,直接拼接在排序链表末尾
        if(l1 == null) curr.next = l2;
        if(l2 == null) curr.next = l1;

        // 最后返回合并后有序的链表
        return dummy.next;
    }
}
```

2.53 乘积最大子数组

给你一个整数数组 nums ， 请你找出数组中乘积最大的非空连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

测试用例的答案是一个 32-位 整数。

子数组 是数组的连续子序列。

```
示例 1:

输入: nums = [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。
```

```
//DP
/*
令imax为当前最大值，则当前最大值为 imax = max(imax * nums[i], nums[i])
由于存在负数，那么会导致最大的变最小的，最小的变最大的。因此还需要维护当前最小值imin，imin =
min(imin * nums[i], nums[i])
当负数出现时则imax与imin进行交换再进行下一步计算
*/
class Solution {
```

```
    public int maxProduct(int[] nums) {
        int max = Integer.MIN_VALUE, imax = 1, imin = 1;
        for(int i=0; i<nums.length; i++){
            if(nums[i] < 0){
              int tmp = imax;
              imax = imin;
              imin = tmp;
            }
            imax = Math.max(imax*nums[i], nums[i]);
            imin = Math.min(imin*nums[i], nums[i]);

            max = Math.max(max, imax);
        }
        return max;
    }
}
```

2.54 最小栈

同剑指offer28题　辅助栈

2.55 相交链表

同剑指offer51题　双指针

2.56 多数元素

同剑指offer38题　摩尔投票法

2.57 打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

```
示例 1:

输入: [1,2,3,1]
输出: 4
解释: 偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
     偷窃到的最高金额 = 1 + 3 = 4 。
```

```
//DP
//dp[n+1] = max(dp[n],dp[n-1]+num)
class Solution {
    public int rob(int[] nums) {
```

```
        int pre = 0, cur = 0, tmp;        //pre表示当前房间的上一间的上一间房，cur（未更新前）表示当前房间上一间房
        for(int num : nums) {
            tmp = cur;
            cur = Math.max(pre + num, cur); //偷当前这间（pre+num）和不偷这间（cur）中取最大的
            pre = tmp;
        }
        return cur;
    }
}
```

2.58 岛屿数量

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

```
示例 1：

输入：grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
输出：1
```

```
//DFS
class Solution {
    public int numIslands(char[][] grid) {
        int count = 0;
        for(int i = 0; i < grid.length; i++) {
            for(int j = 0; j < grid[0].length; j++) {
                if(grid[i][j] == '1'){  //找到岛屿时进行dfs
                    dfs(grid, i, j);
                    count++;            //岛屿数量++
                }
            }
        }
        return count;
    }
    private void dfs(char[][] grid, int i, int j){
        if(i < 0 || j < 0 || i >= grid.length || j >= grid[0].length || grid[i][j] == '0') return;
        grid[i][j] = '0';       //已经搜索过当前网格，将网格清除（置为0）
        dfs(grid, i + 1, j);    //继续搜索为1的网格，直到周围全是0，从而组成1个岛屿
        dfs(grid, i, j + 1);
        dfs(grid, i - 1, j);
        dfs(grid, i, j - 1);
    }
}
```

2.59 反转链表

同剑指offer22题　双指针

2.60 课程表

你这个学期必须选修 numCourses 门课程，记为 0 到 numCourses - 1 。

在选修某些课程之前需要一些先修课程。 先修课程按数组 prerequisites 给出，其中 prerequisites[i] = [ai, bi] ，表示如果要学习课程 ai 则 必须 先学习课程 bi 。

例如，先修课程对 [0, 1] 表示：想要学习课程 0 ，你需要先完成课程 1 。
请你判断是否可能完成所有课程的学习？如果可以，返回 true ；否则，返回 false 。

```
示例 1：
输入：numCourses = 2, prerequisites = [[1,0]]
输出：true
解释：总共有 2 门课程。学习课程 1 之前，你需要完成课程 0 。这是可能的。

示例 2：
输入：numCourses = 2, prerequisites = [[1,0],[0,1]]
输出：false
解释：总共有 2 门课程。学习课程 1 之前，你需要先完成课程 0 ；并且学习课程 0 之前，你还应先完成
课程 1 。这是不可能的。
```

```java
//DFS，拓扑排序
/*
i == 0 ：  干净的，未被 DFS 访问
i == -1：其他节点启动的 DFS 访问过了，路径没问题，不需要再访问了
i == 1  ：本节点启动的 DFS 访问过了，一旦遇到了也说明有环了
*/
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<List<Integer>> adjacency = new ArrayList<>();
        for(int i = 0; i < numCourses; i++)
            adjacency.add(new ArrayList<>());
        int[] flags = new int[numCourses];
        for(int[] cp : prerequisites)       //建立邻接表
            adjacency.get(cp[1]).add(cp[0]);
        for(int i = 0; i < numCourses; i++)//遍历每个节点开始dfs
            if(!dfs(adjacency, flags, i)) return false;
        return true;
    }
    private boolean dfs(List<List<Integer>> adjacency, int[] flags, int i) {
        if(flags[i] == 1) return false;  //先判断再修改标志位
        if(flags[i] == -1) return true;  //别的dfs路径访问过了，我不需要访问了
        flags[i] = 1;   //只有这个标志位是干净的，别人还没有动过，我才能标记为1，说明本次
dfs我遍历过它
        for(Integer j : adjacency.get(i))
           if(!dfs(adjacency, flags, j))
               return false;
        flags[i] = -1;    //只有一次DFS完整结束了，才能执行到这一步，标记为-1，说明这条路
没问题，再遇到不需要遍历了
        return true;
```

```
    }
}
```

2.61 实现 Trie (前缀树)

Trie（发音类似 "try"）或者说 前缀树 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补完和拼写检查。

请你实现 Trie 类：

Trie() 初始化前缀树对象。
void insert(String word) 向前缀树中插入字符串 word 。
boolean search(String word) 如果字符串 word 在前缀树中，返回 true（即，在检索之前已经插入）；否则，返回 false 。
boolean startsWith(String prefix) 如果之前已经插入的字符串 word 的前缀之一为 prefix ，返回 true ；否则，返回 false 。

```
示例：

输入
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
输出
[null, null, true, false, true, null, true]

解释
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // 返回 True
trie.search("app");     // 返回 False
trie.startsWith("app"); // 返回 True
trie.insert("app");
trie.search("app");     // 返回 True
```

```
class Trie {

    class TireNode {        //26叉树
        private boolean isEnd;
        TireNode[] next;

        public TireNode() {
            isEnd = false;      //该结点是否是一个串的结束
            next = new TireNode[26];//字母映射表
        }
    }

    private TireNode root;

    public Trie() {
        root = new TireNode();
    }
```

```
    //这个操作和构建链表很像。首先从根结点的子结点开始与 word 第一个字符进行匹配，一直匹配到前缀链上没有对应的字符，这时开始不断开辟新的结点，直到插入完 word 的最后一个字符，同时还要将最后一个结点isEnd = true;，表示它是一个单词的末尾。
    public void insert(String word) {
        TireNode node = root;
        for (char c : word.toCharArray()) {
            if (node.next[c - 'a'] == null) {
                node.next[c - 'a'] = new TireNode();
            }
            node = node.next[c - 'a'];
        }
        node.isEnd = true;
    }

    //从根结点的子结点开始，一直向下匹配即可，如果出现结点值为空就返回 false，如果匹配到了最后一个字符，那我们只需判断 node->isEnd即可。
    public boolean search(String word) {
        TireNode node = root;
        for (char c : word.toCharArray()) {
            node = node.next[c - 'a'];
            if (node == null) {
                return false;
            }
        }
        return node.isEnd;
    }

    //和 search 操作类似，只是不需要判断最后一个字符结点的isEnd，因为既然能匹配到最后一个字符，那后面一定有单词是以它为前缀的。
    public boolean startsWith(String prefix) {
        TireNode node = root;
        for (char c : prefix.toCharArray()) {
            node = node.next[c - 'a'];
            if (node == null) {
                return false;
            }
        }
        return true;
    }
}
```

2.62 数组中的第K个最大元素

给定整数数组 nums 和整数 k，请返回数组中第 k 个最大的元素。

请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

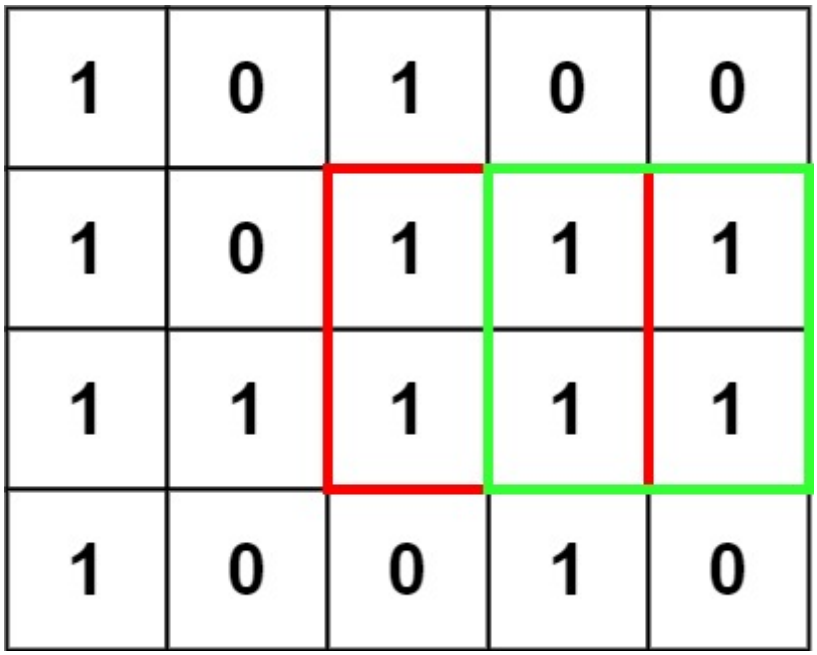
```
示例 1:

输入: [3,2,1,5,6,4] 和 k = 2
输出: 5
```

```
//基于快排并减而治之
public class Solution {
```

```java
    public int findKthLargest(int[] nums, int k) {
        int len = nums.length;
        int left = 0;
        int right = len - 1;

        // 转换一下，第 k 大元素的下标是 len - k
        int target = len - k;

        while (true) {
            int index = partition(nums, left, right);
            if (index == target) {
                return nums[index];
            } else if (index < target) {
                left = index + 1;
            } else {
                right = index - 1;
            }
        }
    }

    /**
     * 对数组 nums 的子区间 [left..right] 执行 partition 操作，返回 nums[left] 排序以后应该在的位置
     * 在遍历过程中保持循环不变量的定义：
     * nums[left + 1..j] < nums[left]
     * nums(j..i) >= nums[left]
     */
    public int partition(int[] nums, int left, int right) {
        int pivot = nums[left];
        int j = left;
        for (int i = left + 1; i <= right; i++) {
            if (nums[i] < pivot) {
                // j 的初值为 left，先右移，再交换，小于 pivot 的元素都被交换到前面
                j++;
                swap(nums, j, i);
            }
        }
        // 在之前遍历的过程中，满足 nums[left + 1..j] < pivot，并且 nums(j..i) >= pivot
        swap(nums, j, left);
        // 交换以后 nums[left..j - 1] < pivot, nums[j] = pivot, nums[j + 1..right] >= pivot
        return j;
    }

    private void swap(int[] nums, int index1, int index2) {
        int temp = nums[index1];
        nums[index1] = nums[index2];
        nums[index2] = temp;
    }
}
```

2.63 最大正方形

在一个由 `'0'` 和 `'1'` 组成的二维矩阵内，找到只包含 `'1'` 的最大正方形，并返回其面积。




```
输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
输出：4
```

```
//DP
//设dp(i,j) 表示以 (i,j) 为右下角，且只包含 1 的正方形的边长最大值。
//如果该位置的值是 1，则dp(i,j) 的值由其上方、左方和左上方的三个相邻位置的 dp 值决定。具体而言，当前位置的元素值等于三个相邻位置的元素中的最小值加 1
//状态转移方程：dp(i,j)=min(dp(i-1,j),dp(i-1,j-1),dp(i,j-1))+1
class Solution {
    public int maximalSquare(char[][] matrix) {
        int maxSide = 0;
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return 0;
        }
        int rows = matrix.length, columns = matrix[0].length;
        int[][] dp = new int[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                if (matrix[i][j] == '1') {
                    if (i == 0 || j == 0) {     //边界上只能为1
                        dp[i][j] = 1;
                    } else {
                        dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) + 1;
                    }
                    maxSide = Math.max(maxSide, dp[i][j]);
                }
            }
        }
        return maxSide * maxSide;
    }
}
```

2.64 翻转二叉树

同剑指offer25题 递归解法

2.65 回文链表

给你一个单链表的头节点 `head` ，请你判断该链表是否为回文链表。如果是，返回 `true` ；否则，返回 `false` 。

```
输入：head = [1,2,2,1]
输出：true
```

```java
//快慢指针、反转链表
class Solution {
    public boolean isPalindrome(ListNode head) {
        if (head == null) {
            return true;
        }

        // 找到前半部分链表的尾节点并反转后半部分链表
        ListNode firstHalfEnd = endOfFirstHalf(head);
        ListNode secondHalfStart = reverseList(firstHalfEnd.next);

        // 判断是否回文
        ListNode p1 = head;
        ListNode p2 = secondHalfStart;
        boolean result = true;
        while (result && p2 != null) {
            if (p1.val != p2.val) {
                result = false;
            }
            p1 = p1.next;
            p2 = p2.next;
        }

        // 还原链表并返回结果
        firstHalfEnd.next = reverseList(secondHalfStart);
        return result;
    }

    //反转链表
    private ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode nextTemp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextTemp;
        }
        return prev;
    }
```

```
    //通过快慢指针，当fast走完，slow正好在链表中间
    private ListNode endOfFirstHalf(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
        while (fast.next != null && fast.next.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }
}
```

2.66 二叉树的最近公共祖先

同剑指offer74题　递归解法

1.67 自身以外数组的乘积

同剑指offer71题

2.68 滑动窗口的最大值

同剑指offer61题　单调队列

2.69 搜索二维矩阵 II

同剑指offer2题

2.70 完全平方数

给你一个整数 n ， 返回 和为 n 的完全平方数的最少数量 。

完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。

```
示例 1：

输入：n = 12
输出：3
解释：12 = 4 + 4 + 4
```

```
//DP
//动态转移方程: dp[i] = Math.min(dp[i], dp[i - j * j] + 1),dp[i-j*j]找到一个j，减去j*j还剩i-j*j继续找最少数量平方数，需要加上本次的j（+1）
class Solution {
    public int numSquares(int n) {
        int[] dp = new int[n + 1]; // 默认初始化值都为0
        for (int i = 1; i <= n; i++) {
            dp[i] = i; // 最坏的情况就是每次+1（1*1）
```

```
            for (int j = 1; i >= j * j; j++) {
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1); // 动态转移方程
            }
        }
        return dp[n];
    }
}
```

2.71 移动零

给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

请注意 ，必须在不复制数组的情况下原地对数组进行操作。

```
示例 1:
输入: nums = [0,1,0,3,12]
输出: [1,3,12,0,0]
```

```
//快排思想
//遍历数组，遇到非零数字放到数组前面（交换），交换完后末尾自动全是0
class Solution {
    public void moveZeroes(int[] nums) {
        if(nums == null) return;
        int j = 0;       //指针从下标0依次递增，依次存放非零数字
        for(int i = 0;i < nums.length;i++){
            if(nums[i] != 0){   //非零数字交换
                int tmp = nums[i];  //先保存i（后面一个指针）位置数字
                nums[i] = nums[j];
                nums[j] = tmp;
                j++;
            }
        }
    }
}
```

2.72 寻找重复数

给定一个包含 n + 1 个整数的数组 nums ， 其数字都在 [1, n] 范围内（包括 1 和 n），可知至少存在一个重复的整数。

假设 nums 只有 一个重复的整数 ， 返回 这个重复的数 。

你设计的解决方案必须 不修改 数组 nums 且只用常量级 O(1) 的额外空间。

```
示例 1:

输入: nums = [1,3,4,2,2]
输出: 2
```

```
//转换为环形链表    快慢指针解法
/*
```

```
如果数组中有重复的数，以数组 [1,3,4,2,2] 为例,我们将数组下标 n 和数 nums[n] 建立一个映射关系 f(n)，
其映射关系 n->f(n) 为：
0->1
1->3
2->4
3->2
4->2
同样的，我们从下标为 0 出发，根据 f(n) 计算出一个值，以这个值为新的下标，再用这个函数计算，以此类推产生一个类似链表一样的序列。从理论上讲，数组中如果有重复的数，那么就会产生多对一的映射，这样，形成的链表就一定会有环路了
1.数组中有一个重复的整数 <==> 链表中存在环
2.找到数组中的重复整数 <==> 找到链表的环入口
慢指针走一步 slow = slow.next ==> 本题 slow = nums[slow]
快指针走两步 fast = fast.next.next ==> 本题 fast = nums[nums[fast]]
*/
class Solution {
    public int findDuplicate(int[] nums) {
        int slow = 0;
        int fast = 0;
        slow = nums[slow];
        fast = nums[nums[fast]];
        while(slow != fast){
            slow = nums[slow];
            fast = nums[nums[fast]];
        }
        int pre1 = 0;
        int pre2 = slow;
        while(pre1 != pre2){
            pre1 = nums[pre1];
            pre2 = nums[pre2];
        }
        return pre1;
    }
}
```

2.73 二叉树的序列化与反序列化

同剑指offer36题

2.74 最长递增子序列

给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。

子序列 是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。

```
示例 1：

输入：nums = [10,9,2,5,3,7,101,18]
输出：4
解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。
```

```
//DP
/*dp[i] 的值代表nums以nums[i]结尾的最长子序列长度
设j∈[0,i)，考虑每轮计算新dp[i] 时，遍历[0,i) 列表区间，做以下判断：
当 nums[i]>nums[j] 时：nums[i] 可以接在nums[j] 之后（此题要求严格递增），此情况下最长上升子序列长度为 dp[j]+1 ；
当 nums[i]<=nums[j] 时：nums[i] 无法接在nums[j] 之后，此情况上升子序列不成立，跳过。
*/
class Solution {
    public int lengthOfLIS(int[] nums) {
        if(nums.length == 0) return 0;
        int[] dp = new int[nums.length];
        int res = 0;
        Arrays.fill(dp, 1);
        for(int i = 0; i < nums.length; i++) {
            for(int j = 0; j < i; j++) {
                if(nums[j] < nums[i]) dp[i] = Math.max(dp[i], dp[j] + 1);
            }
            res = Math.max(res, dp[i]);
        }
        return res;
    }
}
```

2.75 最佳买卖股票时机含冷冻期

给定一个整数数组prices，其中第 prices[i] 表示第 i 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。
注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

```
示例 1:

输入: prices = [1,2,3,0,2]
输出: 3
解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]
```

```
//DP
class Solution {
    public int maxProfit(int[] prices) {
        if (prices.length == 0) {
            return 0;
        }
        // f0: 手上持有股票的最大收益
        // f1: 手上不持有股票，并且处于冷冻期中的累计最大收益
        // f2: 手上不持有股票，并且不在冷冻期中的累计最大收益
        int n = prices.length;
        int f0 = -prices[0];
        int f1 = 0;
        int f2 = 0;
        for (int i = 1; i < n; ++i) {
```

```
            //我们目前持有的这一支股票可以是在第 i-1 天就已经持有的，对应的状态为 f0；或者是第 i 天买入的，那么第 i-1 天就不能持有股票并且不处于冷冻期中，对应的状态为 f2 加上买入股票的负收益 prices[i]
            int newf0 = Math.max(f0, f2 - prices[i]);
            //我们在第 i 天结束之后处于冷冻期的原因是在当天卖出了股票，那么说明在第 i-1 天时我们必须持有一支股票，对应的状态为 f0 加上卖出股票的正收益prices[i]
            int newf1 = f0 + prices[i];
            //我们在第 i 天结束之后不持有任何股票并且不处于冷冻期，说明当天没有进行任何操作，即第 i-1 天时不持有任何股票：如果处于冷冻期，对应的状态为 f1；如果不处于冷冻期，对应的状态为 f2。
            int newf2 = Math.max(f1, f2);
            f0 = newf0;
            f1 = newf1;
            f2 = newf2;
        }

        return Math.max(f1, f2);
    }
}
```

2.76 戳气球

有 n 个气球，编号为0 到 n - 1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。戳破第 i 个气球，你可以获得 nums[i - 1] * nums[i] * nums[i + 1] 枚硬币。 这里的 i - 1 和 i + 1 代表和 i 相邻的两个气球的序号。如果 i - 1或 i + 1 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

```
示例 1：
输入：nums = [3,1,5,8]
输出：167
解释：
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins =  3*1*5    +   3*5*8   +  1*3*8  + 1*8*1 = 167
```

```
//DP   hard题，有点难理解
class Solution {
    public int maxCoins(int[] nums) {
        int n = nums.length;
        // 创建一个辅助数组，并在首尾各添加1，方便处理边界情况
        int[] temp = new int[n+2];
        temp[0] = 1;
        temp[n+1] = 1;
        for(int i=0; i<n; i++){
            temp[i+1] = nums[i];
        }
        int[][] dp = new int[n+2][n+2];
        // len表示开区间长度
        for(int len=3; len<=n+2; len++){
            // i表示开区间左端点
            for(int i=0; i<=n+2-len; i++){
                int res = 0;
                // k为开区间内的索引
```

```
                for(int k = i+1; k<i+len-1; k++){
                    int left = dp[i][k];
                    int right = dp[k][i+len-1];
                    res = Math.max(res, left + temp[i]*temp[k]*temp[i+len-1] +
right);
                }
                dp[i][i+len-1] = res;
            }
        }
        return dp[0][n+1];
    }
}
```

2.77 零钱兑换

给你一个整数数组 coins ， 表示不同面额的硬币；以及一个整数 amount ， 表示总金额。

计算并返回可以凑成总金额所需的 最少的硬币个数 。如果没有任何一种硬币组合能组成总金额，返回 -1 。

你可以认为每种硬币的数量是无限的。

```
示例 1：

输入：coins = [1, 2, 5], amount = 11
输出：3
解释：11 = 5 + 5 + 1
```

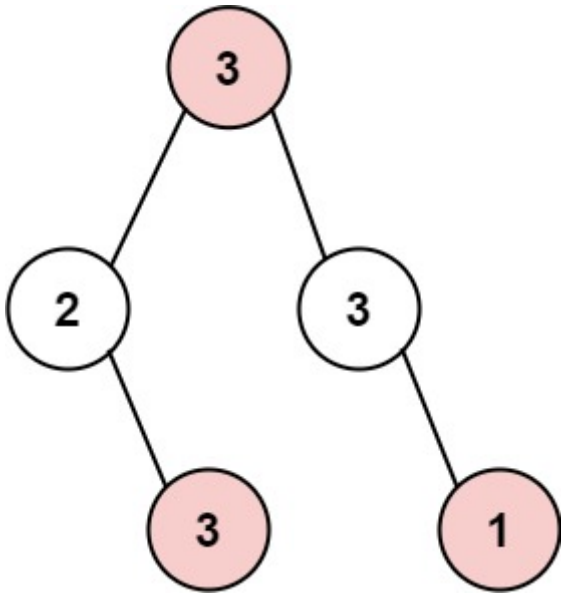
```
//DP
public class Solution {
    public int coinChange(int[] coins, int amount) {
        int max = amount + 1;
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, max);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.length; j++) {
                if (coins[j] <= i) {
                    dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);//从枚举的这枚硬
币面值转移，再加上本身这一枚的贡献
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
}
```

2.78 打家劫舍 III

小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为 root 。

除了 root 之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果 两个直接相连的房子在同一天晚上被打劫 ， 房屋将自动报警。

给定二叉树的 root 。返回 在不触动警报的情况下 ， 小偷能够盗取的最高金额 。




```
//DP
/*使用一个大小为 2 的数组来表示 int[] res = new int[2]; 0 代表不偷，1 代表偷
任何一个节点能偷到的最大钱的状态可以定义为:
当前节点选择不偷：当前节点能偷到的最大钱数 = 左孩子能偷到的钱 + 右孩子能偷到的钱
当前节点选择偷：当前节点能偷到的最大钱数 = 左孩子选择自己不偷时能得到的钱 + 右孩子选择不偷时能得到的钱 + 当前节点的钱数
*/
public int rob(TreeNode root) {
    int[] result = robInternal(root);
    return Math.max(result[0], result[1]);
}

public int[] robInternal(TreeNode root) {
    if (root == null) return new int[2];
    int[] result = new int[2];

    int[] left = robInternal(root.left);
    int[] right = robInternal(root.right);

    result[0] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
    result[1] = left[0] + right[0] + root.val;

    return result;
}
```

2.79 比特位计数

给你一个整数 n ， 对于 0 <= i <= n 中的每个 i ， 计算其二进制表示中 1 的个数 ， 返回一个长度为 n + 1 的数组 ans 作为答案。

```
示例 1:

输入: n = 2
输出: [0,1,1]
解释:
0 --> 0
1 --> 1
2 --> 10
```

```
class Solution {
    public int[] countBits(int n) {
        int[] res = new int[n+1];
        for(int i = 0;i <= n;i++){
            if(i % 2 == 1){          //奇数的1的个数=前一个数（偶数）+1
                res[i] = res[i-1] + 1;
            }else{
                res[i] = res[i/2];  //偶数的1的个数=当前数/2的数的1的个数（因为偶数二进制第一位为0，除以2相当于右移一位）
            }
        }
        return res;
    }
}
```

2.80 前 K 个高频元素

给你一个整数数组 nums 和一个整数 k ， 请你返回其中出现频率前 k 高的元素。你可以按 任意顺序 返回答案。

```
示例 1:

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

```
//HashMap + 堆
/*
首先遍历整个数组，并使用哈希表记录每个数字出现的次数，并形成一个「出现次数数组」。找出原数组的前 k个高频元素，就相当于找出「出现次数数组」的前 k 大的值。

建立一个小顶堆，然后遍历「出现次数数组」：
如果堆的元素个数小于 k，就可以直接插入堆中。
如果堆的元素个数等于 k，则检查堆顶与当前出现次数的大小。如果堆顶更大，说明至少有 k 个数字的出现次数比当前值大，故舍弃当前值；否则，就弹出堆顶，并将当前值插入堆中。
遍历完成后，堆中的元素就代表了「出现次数数组」中前 k 大的值。
*/
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> occurrences = new HashMap<Integer, Integer>();
```

```
        for (int num : nums) {
            occurrences.put(num, occurrences.getOrDefault(num, 0) + 1);
        }

        // int[] 的第一个元素代表数组的值，第二个元素代表了该值出现的次数
        PriorityQueue<int[]> queue = new PriorityQueue<int[]>(new
Comparator<int[]>() {
            public int compare(int[] m, int[] n) {
                return m[1] - n[1];
            }
        });
        for (Map.Entry<Integer, Integer> entry : occurrences.entrySet()) {
            int num = entry.getKey(), count = entry.getValue();
            if (queue.size() == k) {
                if (queue.peek()[1] < count) {
                    queue.poll();
                    queue.offer(new int[]{num, count});
                }
            } else {
                queue.offer(new int[]{num, count});
            }
        }
        int[] ret = new int[k];
        for (int i = 0; i < k; ++i) {
            ret[i] = queue.poll()[0];
        }
        return ret;
    }
}
```

2.81字符串解码

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: k[encoded_string]，表示其中方括号内部的 encoded_string 正好重复 k 次。注意 k 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

```
示例 1:
输入: s = "3[a]2[bc]"
输出: "aaabcbc"

示例 2:
输入: s = "3[a2[c]]"
输出: "accaccacc"
```

```
//辅助栈
class Solution {
    public String decodeString(String s) {
        StringBuilder res = new StringBuilder();
```

```
        int multi = 0;
        Deque<Integer> stack_multi = new ArrayDeque<>();
        Deque<String> stack_res = new ArrayDeque<>();
        for(Character c : s.toCharArray()) {
            if(c == '[') {      //将数字和前面字母入栈
                stack_multi.push(multi);
                stack_res.push(res.toString());
                multi = 0;      //重新恢复初始数字
                res = new StringBuilder();
            }
            else if(c == ']') {//出栈，拼接括号内字符*数字+当前括号外字符（数字左边那个）
                StringBuilder tmp = new StringBuilder();
                int cur_multi = stack_multi.pop();
                for(int i = 0; i < cur_multi; i++) {
                    tmp.append(res);
                }
                res = new StringBuilder(stack_res.pop() + tmp);
            }
            else if(c >= '0' && c <= '9') multi = multi * 10 +
Integer.parseInt(c + "");//内层嵌套需要乘上外层
            else res.append(c);
        }
        return res.toString();
    }
}
```

2.82 根据身高重建队列

假设有打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi ，前面 正好 有 ki 个身高大于或等于 hi 的人。

请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue ，其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性（queue[0] 是排在队列前面的人）。

```
示例 1：

输入：people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]
输出：[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]
解释：
编号为 0 的人身高为 5 ，没有身高更高或者相同的人排在他前面。
编号为 1 的人身高为 7 ，没有身高更高或者相同的人排在他前面。
编号为 2 的人身高为 5 ，有 2 个身高更高或者相同的人排在他前面，即编号为 0 和 1 的人。
编号为 3 的人身高为 6 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
编号为 4 的人身高为 4 ，有 4 个身高更高或者相同的人排在他前面，即编号为 0、1、2、3 的人。
编号为 5 的人身高为 7 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
因此 [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]] 是重新构造后的队列。
```

```
class Solution {
    public int[][] reconstructQueue(int[][] people) {
        //按数组第一个元素进行降序，按第二个元素进行升序
        Arrays.sort(people, new Comparator<int[]>() {
            @Override
            public int compare(int[] person1, int[] person2){
                if (person1[0] != person2[0]){
                    //第一个元素不相等时，第一个元素降序
```

```
                    return person2[0] - person1[0];
                }else{
                    //第一个元素相等时，第二个元素升序
                    return person1[1] - person2[1];
                }
            }
        });
        //新建一个list,用于保存结果集
        List<int[]> list = new LinkedList<>();
        for (int i = 0; i < people.length; i++) {
            if (list.size() > people[i][1]){
                //结果集中元素个数大于第i个人前面应有的人数时，将第i个人插入到结果集的
people[i][1]位置
                list.add(people[i][1],people[i]);
            }else{
                //结果集中元素个数小于等于第i个人前面应有的人数时，将第i个人追加到结果集的后
面
                list.add(list.size(),people[i]);
            }
        }
        //将list转化为数组，然后返回
        return list.toArray(new int[list.size()][]);
    }
}
```

2.83 分割等和子集

给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

```
示例 1：
输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11] 。
```

```
//DP   01背包问题
/*
等价转换：是否可以从输入数组中挑选出一些正整数，使得这些数的和等于整个数组元素的和的一半。
状态定义：dp[i][j]表示从数组的 [0, i] 这个子区间内挑选一些正整数，每个数只能用一次，使得这些数
的和恰好等于 j。
状态转移方程：很多时候，状态转移方程思考的角度是「分类讨论」，对于「0-1 背包问题」而言就是「当前
考虑到的数字选与不选」。
不选择 nums[i]，如果在 [0, i - 1] 这个子区间内已经有一部分元素，使得它们的和为 j ，那么
dp[i][j] = true；
选择 nums[i]，如果在 [0, i - 1] 这个子区间内就得找到一部分元素，使得它们的和为 j -
nums[i]。
*/
public class Solution {
    public boolean canPartition(int[] nums) {
        int len = nums.length;
        // 题目已经说非空数组，可以不做非空判断
        int sum = 0;
        for (int num : nums) {
            sum += num;
```

```
        }
        // 特判：如果是奇数，就不符合要求
        if ((sum & 1) == 1) {
            return false;
        }

        int target = sum / 2;
        // 创建二维状态数组，行：物品索引，列：容量（包括 0）
        boolean[][] dp = new boolean[len][target + 1];

        // 先填表格第 0 行，第 1 个数只能让容积为它自己的背包恰好装满
        if (nums[0] <= target) {
            dp[0][nums[0]] = true;
        }
        // 再填表格后面几行
        for (int i = 1; i < len; i++) {
            for (int j = 0; j <= target; j++) {
                // 直接从上一行先把结果抄下来，然后再修正
                dp[i][j] = dp[i - 1][j];

                if (nums[i] == j) {
                    dp[i][j] = true;
                    continue;
                }
                if (nums[i] < j) {
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i]];
                }
            }
        }
        return dp[len - 1][target];
    }
}
```

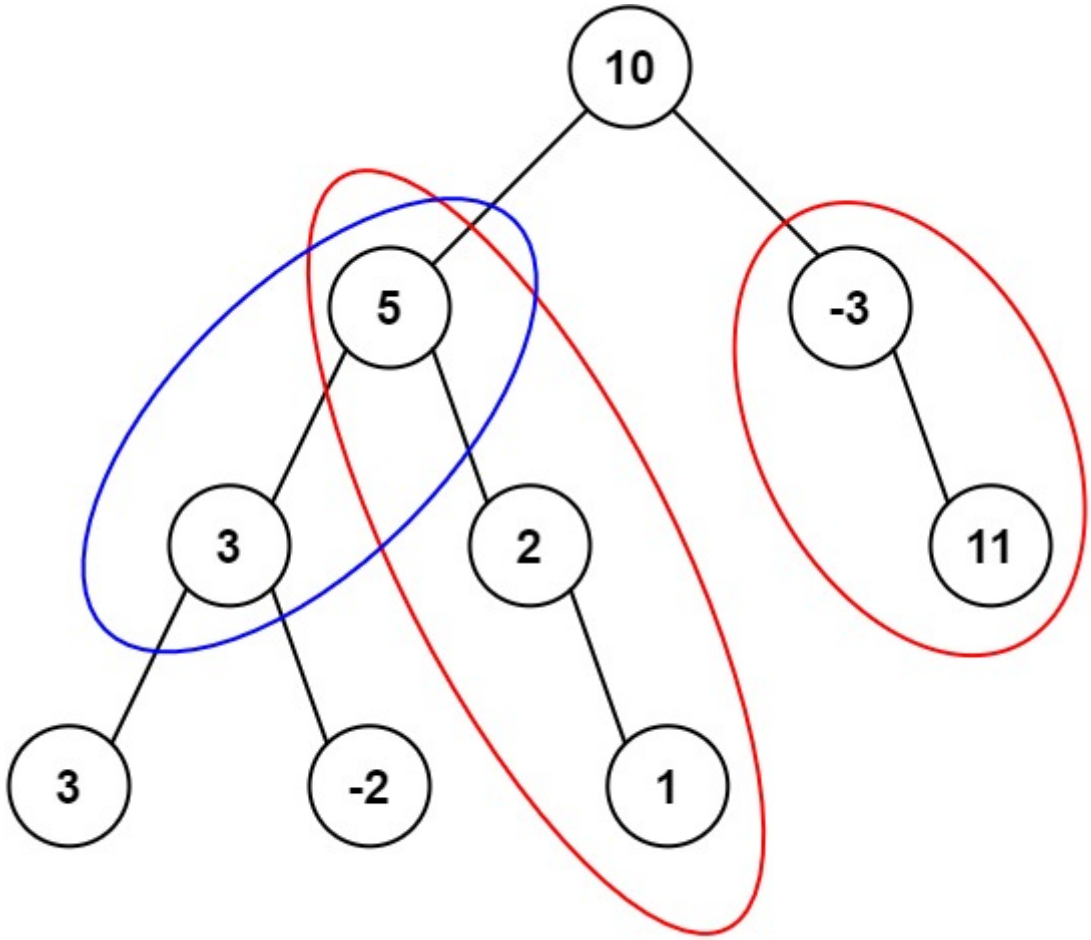
```
//优化
//当前行总是参考了它上面一行 「头顶上」 那个位置和「左上角」某个位置的值
public class Solution {
    public boolean canPartition(int[] nums) {
        int len = nums.length;
        int sum = 0;
        for (int num : nums) {
            sum += num;
        }
        if ((sum & 1) == 1) {
            return false;
        }

        int target = sum / 2;
        boolean[] dp = new boolean[target + 1];
        dp[0] = true;

        if (nums[0] <= target) {
            dp[nums[0]] = true;
        }
        for (int i = 1; i < len; i++) {
            for (int j = target; nums[i] <= j; j--) {   //需要从右向左填，因为需要参
考左上方某一个值
                if (dp[target]) {
```

```
                    return true;
                }
                dp[j] = dp[j] || dp[j - nums[i]];
            }
        }
        return dp[target];
    }
}
```

2.84 路径总和 III

给定一个二叉树的根节点 root ， 和一个整数 targetSum ， 求该二叉树里节点值之和等于 targetSum 的 路径 的数目。

路径 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。




```
输入：root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
输出：3
解释：和等于 8 的路径有 3 条，如图所示。
```

```
//前缀和+回溯
/*
前缀和：到达当前元素的路径上，之前所有元素的和
在同一个路径之下（可以理解成二叉树从root节点出发，到叶子节点的某一条路径），如果两个数的前缀总和是相同的，那么这些节点之间的元素总和为零。进一步扩展相同的想法，如果前缀总和currSum，在节点A和节点B处相差target，则位于节点A和节点B之间的元素之和是target
```

```
*/
class Solution {
    public int pathSum(TreeNode root, int sum) {
        // key是前缀和, value是大小为key的前缀和出现的次数
        Map<Integer, Integer> prefixSumCount = new HashMap<>();
        // 前缀和为0的一条路径
        prefixSumCount.put(0, 1);
        // 前缀和的递归回溯思路
        return recursionPathSum(root, prefixSumCount, sum, 0);
    }

    /**
     * 前缀和的递归回溯思路
     * 从当前节点反推到根节点(反推比较好理解，正向其实也只有一条)，有且仅有一条路径，因为这是一棵树
     * 如果此前有和为currSum-target,而当前的和又为currSum,两者的差就肯定为target了
     * 所以前缀和对于当前路径来说是唯一的，当前记录的前缀和，在回溯结束，回到本层时去除，保证其不影响其他分支的结果
     * @param node 树节点
     * @param prefixSumCount 前缀和Map
     * @param target 目标值
     * @param currSum 当前路径和
     * @return 满足题意的解
     */
    private int recursionPathSum(TreeNode node, Map<Integer, Integer> prefixSumCount, int target, int currSum) {
        // 1.递归终止条件
        if (node == null) {
            return 0;
        }
        // 2.本层要做的事情
        int res = 0;
        // 当前路径上的和
        currSum += node.val;

        //---核心代码
        // 看看root到当前节点这条路上是否存在节点前缀和加target为currSum的路径
        // 当前节点->root节点反推，有且仅有一条路径，如果此前有和为currSum-target,而当前的和又为currSum,两者的差就肯定为target了
        // currSum-target相当于找路径的起点，起点的sum+target=currSum，当前点到起点的距离就是target
        res += prefixSumCount.getOrDefault(currSum - target, 0);
        // 更新路径上当前节点前缀和的个数
        prefixSumCount.put(currSum, prefixSumCount.getOrDefault(currSum, 0) + 1);
        //---核心代码

        // 3.进入下一层
        res += recursionPathSum(node.left, prefixSumCount, target, currSum);
        res += recursionPathSum(node.right, prefixSumCount, target, currSum);

        // 4.回到本层，恢复状态，去除当前节点的前缀和数量
        prefixSumCount.put(currSum, prefixSumCount.get(currSum) - 1);
        return res;
    }
}
```

2.85 找到字符串中所有字母异位词

给定两个字符串 s 和 p，找到 s 中所有 p 的 异位词 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。

```
示例 1:

输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

```java
//滑动窗口（数组实现）
class Solution {
    public List<Integer> findAnagrams(String s, String p) {
        int n = s.length(), m = p.length();
        List<Integer> res = new ArrayList<>();
        if(n < m) return res;
        int[] pCnt = new int[26];
        int[] sCnt = new int[26];
        for(int i = 0; i < m; i++){     //根据p的长度，先把p和s前m个转换为字母出现频次
            pCnt[p.charAt(i) - 'a']++;
            sCnt[s.charAt(i) - 'a']++;
        }
        if(Arrays.equals(sCnt, pCnt)){  //初始化时就相等，那么下标0满足条件
            res.add(0);
        }
        for(int i = m; i < n; i++){     //滑动窗口开始滑动
            sCnt[s.charAt(i - m) - 'a']--;  //每次滑动一个字母，sCnt中左边字母出去，右边进来一个字母
            sCnt[s.charAt(i) - 'a']++;
            if(Arrays.equals(sCnt, pCnt)){  //每次比较
                res.add(i - m + 1);
            }
        }
        return res;
    }
}
```

2.86 找到所有数组中消失的数字

给你一个含 n 个整数的数组 nums ，其中 nums[i] 在区间 [1, n] 内。请你找出所有在 [1, n] 范围内但没有出现在 nums 中的数字，并以数组的形式返回结果。

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

```java
//原地哈希
```

```java
//遍历原数组，将数组值-1作为数组下标索引，找到对应值+n；再次遍历数组，下标对应值小于n的下标+1即为消失的数字（有点绕）
class Solution {
    public List<Integer> findDisappearedNumbers(int[] nums) {
        int n = nums.length;
        for (int num : nums) {
            int x = (num - 1) % n;//当遍历到某个位置时，其中的数可能已经被增加过，因此需要对 n 取模来还原出它本来的值。
            nums[x] += n;
        }
        List<Integer> ret = new ArrayList<Integer>();
        for (int i = 0; i < n; i++) {
            if (nums[i] <= n) {
                ret.add(i + 1);
            }
        }
        return ret;
    }
}
```

2.87 汉明距离

两个整数之间的 汉明距离 指的是这两个数字对应二进制位不同的位置的数目。

给你两个整数 x 和 y，计算并返回它们之间的汉明距离。

```
示例 1：

输入：x = 1, y = 4
输出：2
解释：
1   (0 0 0 1)
4   (0 1 0 0)
       ↑   ↑
上面的箭头指出了对应二进制位不同的位置。
```

```java
//先异或，再&1判断末尾，最后右移一位即可
class Solution {
    public int hammingDistance(int x, int y) {
        int res = 0,z = x ^ y;
        while(z != 0){
            res += z & 1;
            z >>= 1;
        }
        return res;
    }
}
```

2.88 目标和

给你一个整数数组 nums 和一个整数 target 。

向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式 ：

例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，然后串联起来得到表达式 "+2-1" 。
返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。

```
示例 1：

输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 3 。
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3
```

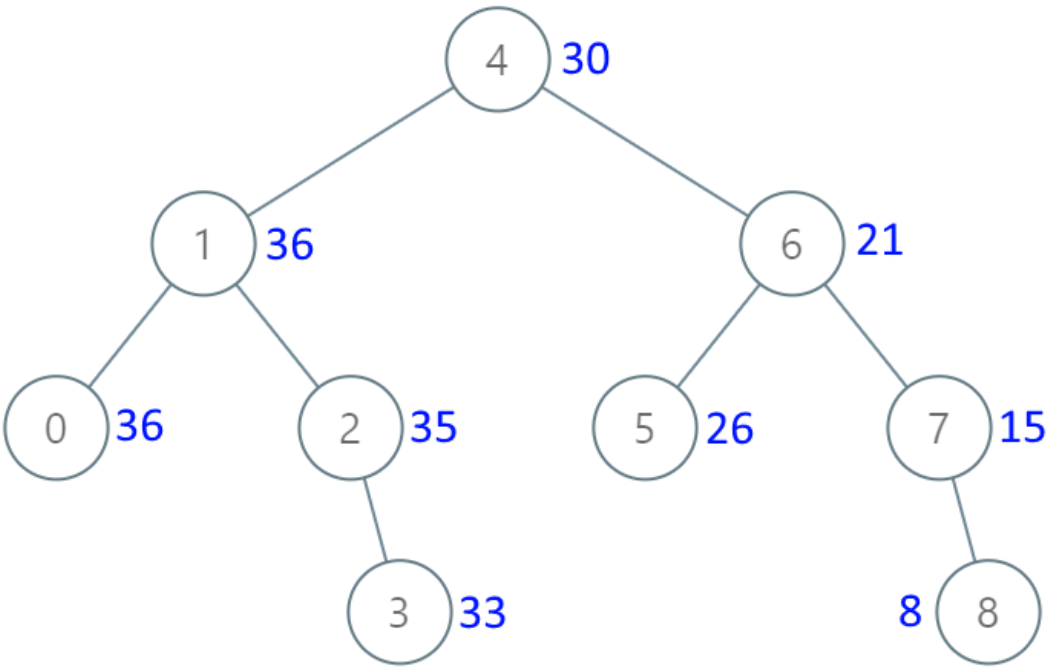
```java
//DP  同样的01背包问题
/*
     * 转换思路
     * 数组和（全部+）为sum  1  2  3  4    +1 -2 +3 +4     sum=10  neg=2  8  8-2=6为方案值
     * 假设在一个方案中  -的数的总和为neg  那么+的数的总和就应该为sum-neg
     * 那么一个方案结果为(sum-neg)-neg
     * 令这个方案值为target   target=sum-2*neg   ->  neg=(sum-target)/2
     * 首先  sum肯定要大于target  否则没办法实现(全+)
     * 所以 sum-target>0  且sum-target为偶数
     *
     * 然后以neg为目标数  每次做减法
     * dp[i][j]表示 第i个数 值为j的方案
     * dp[i][j]=dp[i-1][j]+dp[i-1][j-nums[i]];
*/
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        int sum = 0;
        for (int num : nums) {
            sum += num;
        }
        int diff = sum - target;
        if (diff < 0 || diff % 2 != 0) {
            return 0;
        }
        int n = nums.length, neg = diff / 2;
        int[][] dp = new int[n + 1][neg + 1];
        dp[0][0] = 1;
        for (int i = 1; i <= n; i++) {
            int num = nums[i - 1];
            for (int j = 0; j <= neg; j++) {
                dp[i][j] = dp[i - 1][j];
                if (j >= num) {
                    dp[i][j] += dp[i - 1][j - num];
                }
            }
        }
```

```
        return dp[n][neg];
    }
}
```

```
//空间优化
class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        int sum = 0;
        for (int num : nums) {
            sum += num;
        }
        int diff = sum - target;
        if (diff < 0 || diff % 2 != 0) {
            return 0;
        }
        int neg = diff / 2;
        int[] dp = new int[neg + 1];
        dp[0] = 1;
        for (int num : nums) {
            for (int j = neg; j >= num; j--) {
                dp[j] += dp[j - num];
            }
        }
        return dp[neg];
    }
}
```

2.89 把二叉搜索树转换为累加树

给出二叉 搜索 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 node 的新值等于原树中大于或等于 node.val 的值之和。



```
//本题中要求我们将每个节点的值修改为原来的节点值加上所有大于它的节点值之和。这样我们只需要反序中序遍历该二叉搜索树，记录过程中的节点值之和，并不断更新当前遍历到的节点的节点值，即可得到题目要求的累加树
class Solution {
    int sum = 0;    //定义在递归函数外面
    public TreeNode convertBST(TreeNode root) {
        if (root != null) {
            convertBST(root.right);
            sum += root.val;
            root.val = sum;
            convertBST(root.left);
        }
        return root;
    }
}
```

2.90 二叉树的直径

给定一棵二叉树，你需要计算它的直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

```
示例 :
给定二叉树
      1
     / \
    2   3
   / \
  4   5
返回 3, 它的长度是路径 [4,2,1,3] 或者 [5,2,1,3]。
```

```
//DFS
//找出左右子树深度相加
class Solution {
    int ans;
    public int diameterOfBinaryTree(TreeNode root) {
        ans = 1;
        depth(root);
        return ans - 1;     //路径=节点数-1
    }
    public int depth(TreeNode node) {
        if (node == null) {
            return 0; // 访问到空节点了，返回0
        }
        int L = depth(node.left); // 左儿子为根的子树的深度
        int R = depth(node.right); // 右儿子为根的子树的深度
        ans = Math.max(ans, L+R+1); // 计算d_node即L+R+1 并更新ans
        return Math.max(L, R) + 1; // 返回该节点为根的子树的深度
    }
}
```

2.91 和为 K 的子数组

给你一个整数数组 nums 和一个整数 k ， 请你统计并返回该数组中和为 k 的连续子数组的个数。

```
示例 1:
输入: nums = [1,1,1], k = 2
输出: 2
```

```
//前缀和
class Solution {
    public int subarraySum(int[] nums, int k) {
        // 记录合适的连续字符串数量
        int count=0;
        // 记录前面数字相加之和
        int pre=0;
        // map记录前几个数字之和为key，出现相同和的次数为value
        HashMap<Integer,Integer> map = new HashMap<>();
        // 初始化
        map.put(0,1);
        for (int i = 0; i < nums.length; i++) {
            pre+= nums[i];  // 前缀和
            // 设
            // pre[i]=pre[i-1]+nums[i]
            // 由于补上了0，1这个情况 问题由多少个连续数字之和等于k 转为
            // pre[i]-pre[j-1]==k （前缀和之差为k，代表这两个前缀和中间的数字相加就是K）
            // 如果前面某些数字之和加上这个数字正好等于K（存在一个数字加上nums[i]结果为K
            // 说明找到了
            if (map.containsKey(pre-k)){
                // 累计
                count+=map.get(pre-k);
            }
            // 计算新的和放入map
            map.put(pre,map.getOrDefault(pre,0)+1);
        }
        return count;
    }
}
```

2.92 最短无序连续子数组

给你一个整数数组 nums ， 你需要找出一个 连续子数组 ， 如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。

请你找出符合题意的 最短 子数组，并输出它的长度。

```
示例 1:

输入: nums = [2,6,4,8,10,9,15]
输出: 5
解释: 你只需要对 [6, 4, 8, 10, 9] 进行升序排序，那么整个表都会变为升序排序。
```

```
/*
我们可以假设把这个数组分成三段，左段和右段是标准的升序数组，中段数组虽是无序的，但满足最小值大于左段的最大值，最大值小于右段的最小值。
```

```
那么我们目标就很明确了，找中段的左右边界，我们分别定义为begin 和 end;
分两头开始遍历:

从左到右维护一个最大值max,在进入右段之前，那么遍历到的nums[i]都是小于max的，我们要求的end就是遍历中最后一个小于max元素的位置;
同理，从右到左维护一个最小值min，在进入左段之前，那么遍历到的nums[i]也都是大于min的，要求的begin也就是最后一个大于min元素的位置。
*/
//带入题目数组模拟一下
class Solution {
    public int findUnsortedSubarray(int[] nums) {
        //初始化
        int len = nums.length;
        int min = nums[len-1];
        int max = nums[0];
        int begin = 0, end = -1;     //初始化为-1，保证原数组升序时返回0
        //遍历
        for(int i = 0; i < len; i++){
            if(nums[i] < max){       //从左到右维持最大值，寻找右边界end
                end = i;
            }else{
                max = nums[i];
            }
            if(nums[len-i-1] > min){     //从右到左维持最小值，寻找左边界begin
                begin = len-i-1;
            }else{
                min = nums[len-i-1];
            }
        }
        return end-begin+1;
    }
}
```

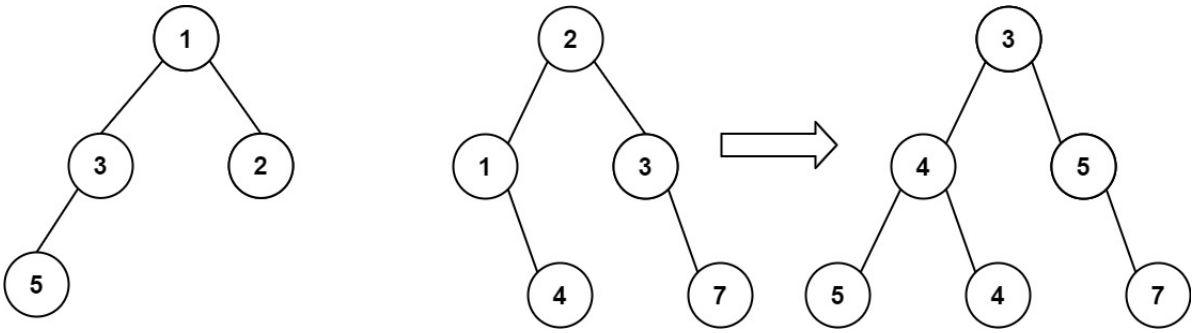
2.93 合并二叉树

给你两棵二叉树： root1 和 root2 。

想象一下，当你将其中一棵覆盖到另一棵之上时，两棵树上的一些节点将会重叠（而另一些不会）。你需要将这两棵树合并成一棵新二叉树。合并的规则是：如果两个节点重叠，那么将这两个节点的值相加作为合并后节点的新值；否则，不为 null 的节点将直接作为新二叉树的节点。

返回合并后的二叉树。

注意: 合并过程必须从两个树的根节点开始。




```
//DFS
```

```
class Solution {
    public TreeNode mergeTrees(TreeNode root1, TreeNode root2) {
        if(root1 == null){
            return root2;
        }
        if(root2 == null){
            return root1;
        }
        TreeNode merged = new TreeNode(root1.val + root2.val);  //构建新的节点，相当于重新生成一棵树
        merged.left = mergeTrees(root1.left,root2.left);
        merged.right = mergeTrees(root1.right,root2.right);
        return merged;
    }
}
```

2.94 回文子串

给你一个字符串 s ，请你统计并返回这个字符串中 回文子串 的数目。

回文字符串 是正着读和倒过来读一样的字符串。

子字符串 是字符串中的由连续字符组成的一个序列。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。

```
示例 1：

输入：s = "abc"
输出：3
解释：三个回文子串: "a", "b", "c"
```

```
class Solution {
    public int countSubstrings(String s) {
        // 中心扩展法
        int ans = 0;
        for (int center = 0; center < 2 * s.length() - 1; center++) {
            // left和right指针和中心点的关系是？
            // 首先是left，有一个很明显的2倍关系的存在，其次是right，可能和left指向同一个（偶数时），也可能往后移动一个（奇数）
            // 大致的关系出来了，可以选择带两个特殊例子进去看看是否满足。
            int left = center / 2;
            int right = left + center % 2;

            while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
                ans++;
                left--;
                right++;
            }
        }
        return ans;
    }
}
```

2.95 每日温度

给定一个整数数组 temperatures ，表示每天的温度，返回一个数组 answer ，其中 answer[i] 是指在第 i 天之后，才会有更高的温度。如果气温在这之后都不会升高，请在该位置用 0 来代替。

```
示例 1:

输入: temperatures = [73,74,75,71,69,72,76,73]
输出: [1,1,4,2,1,1,0,0]
```

```java
/**
 * 最简单莫过于双重循环，笔试时至少不会丢分
 */
public int[] dailyTemperatures(int[] T) {
    int[] res = new int[T.length];
    for (int i = 0; i < T.length - 1; i++) {
        for (int j =  i + 1; j < T.length; j++) {
            if (T[j] > T[i]) {
                res[i] = j - i;
                break;
            }
        }
    }
    return res;
}

/**
 * 根据题意，从最后一天推到第一天，这样会简单很多。因为最后一天显然不会再有升高的可能，结果直接为0。
 * 再看倒数第二天的温度，如果比倒数第一天低，那么答案显然为1，如果比倒数第一天高，又因为倒数第一天
 * 对应的结果为0，即表示之后不会再升高，所以倒数第二天的结果也应该为0。
 * 自此我们容易观察出规律，要求出第i天对应的结果，只需要知道第i+1天对应的结果就可以：
 * - 若T[i] < T[i+1]，那么res[i]=1；
 * - 若T[i] > T[i+1]
 *   - res[i+1]=0，那么res[i]=0;
 *   - res[i+1]!=0，那就比较T[i]和T[i+1+res[i+1]]（即将第i天的温度与比第i+1天大的那天的温度进行比较）
 */
public int[] dailyTemperatures(int[] T) {
    int[] res = new int[T.length];
    res[T.length - 1] = 0;
    for (int i = T.length - 2; i >= 0; i--) {
        for (int j = i + 1; j < T.length; j += res[j]) {
            if (T[i] < T[j]) {
                res[i] = j - i;
                break;
            } else if (res[j] == 0) {
                res[i] = 0;
                break;
            }
        }
    }
    return res;
}
```

2.96 编辑距离

给你两个单词 word1 和 word2， 请返回将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符
删除一个字符
替换一个字符

```
示例 1：

输入：word1 = "horse", word2 = "ros"
输出：3
解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

```
/*
dp[i][j] 代表 word1 到 i 位置转换成 word2 到 j 位置需要最少步数
所以，
当 word1[i] == word2[j]，dp[i][j] = dp[i-1][j-1]；
当 word1[i] != word2[j]，dp[i][j] = min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1
其中，dp[i-1][j-1] 表示替换操作，dp[i-1][j] 表示删除操作，dp[i][j-1] 表示插入操作
注意，针对第一行，第一列要单独考虑.
*/
//DP
class Solution {
    public int minDistance(String word1, String word2) {
        int n1 = word1.length();
        int n2 = word2.length();
        int[][] dp = new int[n1 + 1][n2 + 1];
        // 第一行
        for (int j = 1; j <= n2; j++) dp[0][j] = dp[0][j - 1] + 1;
        // 第一列
        for (int i = 1; i <= n1; i++) dp[i][0] = dp[i - 1][0] + 1;

        for (int i = 1; i <= n1; i++) {
            for (int j = 1; j <= n2; j++) {
                if (word1.charAt(i - 1) == word2.charAt(j - 1)) dp[i][j] = dp[i
- 1][j - 1];
                else dp[i][j] = Math.min(Math.min(dp[i - 1][j - 1], dp[i][j -
1]), dp[i - 1][j]) + 1;
            }
        }
        return dp[n1][n2];
    }
}
```

三、其它

3.1 字形变换

将一个给定字符串 `s` 根据给定的行数 `numRows` ，以从上往下、从左到右进行 Z 字形排列。

```
示例 2：
输入：s = "PAYPALISHIRING", numRows = 4
输出："PINALSIGYAHRPI"
解释：
P     I    N
A   L S  I G
Y A   H R
P     I
```

```
class Solution {
    public String convert(String s, int numRows) {
        if(numRows < 2) return s;
        List<StringBuilder> rows = new ArrayList<StringBuilder>();
        //每行一个StringBuilder
        for(int i = 0; i < numRows; i++) rows.add(new StringBuilder());
        int i = 0, flag = -1;
        for(char c : s.toCharArray()) {
            rows.get(i).append(c);
            if(i == 0 || i == numRows -1) flag = - flag;    //转折点反向
            i += flag;      //更新行索引
        }
        StringBuilder res = new StringBuilder();
        for(StringBuilder row : rows) res.append(row);
        return res.toString();
    }
}
```

3.2 整数反转

给你一个 32 位的有符号整数 x ，返回将 x 中的数字部分反转后的结果。

如果反转后整数超过 32 位的有符号整数的范围 [−231，231 − 1] ，就返回 0。

假设环境不允许存储 64 位整数（有符号或无符号）。

```
示例 1：
输入：x = 123
输出：321
```

```
class Solution {
    public int reverse(int x) {
        int rev = 0;
        while(x != 0){
            if(rev < Integer.MIN_VALUE / 10 || rev > Integer.MAX_VALUE / 10)
                return 0;
            // 弹出 x 的末尾数字
            int tmp = x % 10;
            x /= 10;
```

```
            // 将数字推入rev末尾
            rev = rev * 10 + tmp;
        }
        return rev;
    }
}
```

3.3 字符串转换整数 (atoi)

同剑指offer72题

3.4 回文数

给你一个整数 x ，如果 x 是一个回文整数，返回 true ；否则，返回 false 。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。例如，121 是回文，而 123 不是。

```
示例 1：

输入：x = 121
输出：true
```

```
class Solution {
    public boolean isPalindrome(int x) {
        // 特殊情况：
        // 如上所述，当 x < 0 时，x 不是回文数。
        // 同样地，如果数字的最后一位是 0，为了使该数字为回文，
        // 则其第一位数字也应该是 0
        // 只有 0 满足这一属性
        if (x < 0 || (x % 10 == 0 && x != 0)) {
            return false;
        }

        int revertedNumber = 0;
        while (x > revertedNumber) {
            revertedNumber = revertedNumber * 10 + x % 10;
            x /= 10;
        }

        // 当数字长度为奇数时，我们可以通过 revertedNumber/10 去除处于中位的数字。
        // 例如，当输入为 12321 时，在 while 循环的末尾我们可以得到 x = 12，
revertedNumber = 123，
        // 由于处于中位的数字不影响回文（它总是与自己相等），所以我们可以简单地将其去除。
        return x == revertedNumber || x == revertedNumber / 10;
    }
}
```

2.12 整数转罗马数字

罗马数字包含以下七种字符： I， V， X， L，C，D 和 M。

字符　　数值
I　　1
V　　5
X　　10
L　　50
C　　100
D　　500
M　　1000
例如， 罗马数字 2 写做 II ，即为两个并列的 1。12 写做 XII ，即为 X + II 。 27 写做 XXVII, 即为 XX + V + II

通常情况下，罗马数字中小的数字在大的数字的右边。但也存在特例，例如 4 不写做 IIII，而是 IV。数字 1 在数字 5 的左边，所表示的数等于大数 5 减小数 1 得到的数值 4 。同样地，数字 9 表示为 IX。这个特殊的规则只适用于以下六种情况：

I 可以放在 V (5) 和 X (10) 的左边，来表示 4 和 9。
X 可以放在 L (50) 和 C (100) 的左边，来表示 40 和 90。
C 可以放在 D (500) 和 M (1000) 的左边，来表示 400 和 900。
给你一个整数，将其转为罗马数字

```
//贪心算法，尽可能先选出大的数字进行转换
public class Solution {
    public String intToRoman(int num) {
        // 把阿拉伯数字与罗马数字可能出现的所有情况和对应关系，放在两个数组中，并且按照阿拉伯数字的大小降序排列
        int[] nums = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
        String[] romans = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};
        StringBuilder stringBuilder = new StringBuilder();
        int index = 0;
        while (index < 13) {
            // 特别注意：这里是等号
            while (num >= nums[index]) {
                stringBuilder.append(romans[index]);
                num -= nums[index];
            }
            index++;
        }
        return stringBuilder.toString();
    }
}
```

3.5 罗马数字转整数

给定一个罗马数字，将其转换成整数

```
//模拟法
class Solution {
    Map<Character, Integer> symbolValues = new HashMap<Character, Integer>() {{
        put('I', 1);
        put('V', 5);
        put('X', 10);
        put('L', 50);
        put('C', 100);
```

```
        put('D', 500);
        put('M', 1000);
    }};

    public int romanToInt(String s) {
        int ans = 0;
        int n = s.length();
        for (int i = 0; i < n; ++i) {
            int value = symbolValues.get(s.charAt(i));
            if (i < n - 1 && value < symbolValues.get(s.charAt(i + 1))) {
                ans -= value;
            } else {
                ans += value;
            }
        }
        return ans;
    }
}
```

3.6 最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。

如果不存在公共前缀，返回空字符串 ""。

```
示例 1:
输入: strs = ["flower","flow","flight"]
输出: "fl"
```

```
//纵向比较
class Solution {
    public String longestCommonPrefix(String[] strs) {
        if(strs.length == 0 || strs == null) return "";
        int count = strs.length;
        int len = strs[0].length();          //以第一个字符串为参考
        for(int i = 0;i < len;i++){
            char c = strs[0].charAt(i);
            for(int j = 1;j < count;j++){
                if(i == strs[j].length() || c != strs[j].charAt(i))     //注意i,j
                    return strs[0].substring(0,i);              //直接在第一个字符串
上修改
            }
        }
        return strs[0];
    }
}
```

3.7 最接近的三数之和

给你一个长度为 n 的整数数组 nums 和 一个目标值 target。请你从 nums 中选出三个整数，使它们的和与 target 最接近。

返回这三个数的和。

假定每组输入只存在恰好一个解。

```
示例 1：

输入：nums = [-1,2,1,-4], target = 1
输出：2
解释：与 target 最接近的和是 2 (-1 + 2 + 1 = 2) 。
```

```java
//同样使用排序+双指针
class Solution {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int n = nums.length;
        int best = 10000000;

        // 枚举 a
        for (int i = 0; i < n; ++i) {
            // 保证和上一次枚举的元素不相等
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            // 使用双指针枚举 b 和 c
            int j = i + 1, k = n - 1;
            while (j < k) {
                int sum = nums[i] + nums[j] + nums[k];
                // 如果和为 target 直接返回答案
                if (sum == target) {
                    return target;
                }
                // 根据差值的绝对值来更新答案
                if (Math.abs(sum - target) < Math.abs(best - target)) {
                    best = sum;
                }
                if (sum > target) {
                    // 如果和大于 target，移动 c 对应的指针
                    int k0 = k - 1;
                    // 移动到下一个不相等的元素
                    while (j < k0 && nums[k0] == nums[k]) {
                        --k0;
                    }
                    k = k0;
                } else {
                    // 如果和小于 target，移动 b 对应的指针
                    int j0 = j + 1;
                    // 移动到下一个不相等的元素
                    while (j0 < k && nums[j0] == nums[j]) {
                        ++j0;
                    }
                    j = j0;
                }
            }
        }
```

```
        return best;
    }
}
```