# 数组

***//长n数组里所有数在0到n-1内，某些数重复，不知道有几个数重复，也不知道重复几次。找出任一个重复数***public class \_1 {  
 public boolean duplicate(int[] nums, int length, int[] duplication) {  
 if (nums == null || length <= 0) //如果数组为空，返回false  
 return false;  
 for (int i = 0; i < length; i++) { //遍历数组  
 while (nums[i] != i) {  
 //如果nums[i] == nums[nums[i]] 说明找到了重复元素，  
 if (nums[i] == nums[nums[i]]) {  
 //将重复数字放入duplication，返回true  
 duplication[0] = nums[i];  
 return true;  
 }  
 swap(nums, i, nums[i]); //将num[i]放在第i位置  
 }  
 }  
 return false;  
 }  
 private void swap(int[] nums, int i, int j) {  
 int t = nums[i];  
 nums[i] = nums[j];  
 nums[j] = t;  
 }  
}

***//二维数组每行从左到右递增，从上到下递增。判断某个数是否在该二维数组***public class \_2 {  
 public boolean Find(int target, int[][] matrix) {  
 //如果数组为空，返回false  
 if (matrix == null || matrix.length == 0 || matrix[0].length == 0)  
 return false;  
 int rows = matrix.length, cols = matrix[0].length;  
 int r = 0, c = cols - 1; // 从右上角开始  
 while (r <= rows - 1 && c >= 0) {  
 if (target == matrix[r][c]) //如果找到了目标，返回true  
 return true;  
 else if (target > matrix[r][c]) //如果当前值小于目标值，行++  
 r++;  
 else //如果当前值大于目标值，列--  
 c--;  
 }  
 return false; //如果最后找不到返回false  
 }  
}

***//一个数组最开始若干元素搬到末尾称为旋转，一个非递减数组的旋转，输出旋转数组最小元素***public class \_12 {  
 public int minNumberInRotateArray(int[] nums) {  
 if (nums.length == 0) //如果数组为空，返回0  
 return 0;  
 int l = 0, h = nums.length - 1; //左右边界为数组边界  
 while (l < h) { //二分查找  
 int m = l + (h - l) / 2; //计算中间下标  
 //如果左中右三点都相等，返回l到h的最小值  
 if (nums[l] == nums[m] && nums[m] == nums[h])  
 return minNumber(nums, l, h);  
 //如果m值≤右边值，更新h为m  
 else if (nums[m] <= nums[h])  
 h = m;  
 //否则更新l为m+1  
 else  
 l = m + 1;  
 }  
 return nums[l];  
 }  
 //返回nums从l到h范围最小值，因为旋转升序数组一次，l到h或者升序，或者只有一个降序节点  
 private int minNumber(int[] nums, int l, int h) {  
 for (int i = l; i < h; i++)  
 //如果i比i+1大，说明降序节点就在这，后面的数都比i+1大，返回i+1数  
 if (nums[i] > nums[i + 1])  
 return nums[i + 1];  
 //否则返回最左侧数  
 return nums[l];  
 }  
}

***//把一根绳子剪成多段，并且使得每段的长度乘积最大***public class \_15 {  
 public int integerBreak(int n) {  
 //尽可能多剪长3的绳子，且不允许长1 绳子出现  
 if (n < 2) return 0;//如果绳子长度小于2没法剪断，返回0  
 if (n == 2) return 1; //长度2的绳子只能剪成2个1，乘积为1  
 if (n == 3) return 2; //长度3的绳子只能剪成1和2，乘积为2  
 int timesOf3 = n / 3; //计算能获取多少段3  
 if (n - timesOf3 \* 3 == 1) //如果全是3最后剩下1，就少一个3，变成2和2  
 timesOf3--;  
 int timesOf2 = (n - timesOf3 \* 3) / 2; //计算有多少个2  
 //计算总乘积，3的timesOf3次方\*2的timesOf2次方  
 return (int) (Math.pow(3, timesOf3)) \* (int) (Math.pow(2, timesOf2));  
 }  
}

***//调整数组顺序使奇数位于偶数前，要保证奇数和奇数，偶数和偶数相对位置不变***public class \_22 {  
 public void reOrderArray(int[] nums) {  
 int oddCnt = 0; // 先计算出奇数个数  
 for (int x : nums)  
 if (!isEven(x))  
 oddCnt++;  
 int[] copy = nums.clone(); //复制出临时数组  
 int i = 0, j = oddCnt;  
 //遍历原数组，如果是奇数，就放前面，偶数就从j++位置开始放，放后面，因为奇数个数是j  
 for (int num : copy) {  
 if (num % 2 == 1)  
 nums[i++] = num;  
 else  
 nums[j++] = num;  
 }  
 }  
 //如果数值对2取余为0就返回true，即是偶数  
 private boolean isEven(int x) {  
 return x % 2 == 0;  
 }  
}

***//数组 输入一个矩阵，从外向里顺时针依次打印每个数字***public class \_30 {  
 public ArrayList<Integer> printMatrix(int[][] matrix) {  
 ArrayList<Integer> ret = new ArrayList<>();  
 //r1是行的上边界，r2是行下边界，c1是列左边界，c2是列右边界  
 int r1 = 0, r2 = matrix.length - 1, c1 = 0, c2 = matrix[0].length - 1;  
 //不能超界  
 while (r1 <= r2 && c1 <= c2) {  
 //从左到右打印上边界  
 for (int i = c1; i <= c2; i++)  
 ret.add(matrix[r1][i]);  
 //从上到下打印右边界，注意初始条件是r1+1，否则右上角数字打印重复  
 for (int i = r1 + 1; i <= r2; i++)  
 ret.add(matrix[i][c2]);  
 //如果上下边界没重合，就从右往左打印下边界，注意初始条件c2-1，否则右下角数字重复打印  
 if (r1 != r2)  
 for (int i = c2 - 1; i >= c1; i--)  
 ret.add(matrix[r2][i]);  
 //如果左右边界没重合，就从下往上打印左边界，注意初始条件r2-1，否则左下角数字重复  
 if (c1 != c2)  
 for (int i = r2 - 1; i > r1; i--)  
 ret.add(matrix[i][c1]);  
 //上边界++，下边界--，左边界++，右边界--，打印里面一圈  
 r1++; r2--; c1++; c2--;  
 }  
 return ret;  
 }  
}

***//数组 从 1 到 n 整数中 1 出现的次数***public class \_47 {  
 public static int NumberOf1Between1AndN\_Solution(int n) {  
 int cnt = 0;  
 for (int m = 1; m <= n; m \*= 10) { //m从1开始，小于等于n，每次\*10  
 //a=n/m b=n%m  
 int a = n / m, b = n % m;  
 //每次增加(a+8)/10\*m再加，如果a%10==1就加b+1，否则是0  
 cnt += (a + 8) / 10 \* m + (a % 10 == 1 ? b + 1 : 0);  
 }  
 return cnt;  
 }  
}

***//数组 数组中出现次数超一半的数字，如{1,2,3,2,2,2,5,4,2}。2出现5次超过一半输出2。如不存在输出0***public class \_42 {  
 public int MoreThanHalfNum\_Solution(int[] nums) {  
 //majority代表数组中出现次数最多的元素值，首先设置第0个元素为majority  
 int majority = nums[0];  
 for (int i = 1, cnt = 1; i < nums.length; i++) {  
 //遍历数组，如果某个元素等于majority，就cnt++，否则cnt--  
 cnt = nums[i] == majority ? cnt + 1 : cnt - 1;  
 //如果cnt减到0，就更新majority为当前值，记录出现次数为1  
 if (cnt == 0) {  
 majority = nums[i];  
 cnt = 1;  
 }  
 }  
 //遍历数组，找到majority出现的次数，如果超过一半就返回majority否则返回0  
 int cnt = 0;  
 for (int val : nums)  
 if (val == majority)  
 cnt++;  
 return cnt > nums.length / 2 ? majority : 0;  
 }  
}

***//数组 求某数组的连续子数组最大和***public class \_46 {  
 public static int FindGreatestSumOfSubArray(int[] nums) {  
 //如果数组为空，返回0  
 if (nums == null || nums.length == 0) return 0;  
 //将最大和初始值设置为最小值  
 int greatestSum = Integer.MIN\_VALUE;  
 int sum = 0;  
 for (int val : nums) {  
 //遍历数组，如果sum小于0，说明前面总和为负，且前面这些数最大子数组也已保存在greatestSum中  
 //就让sum设置为当前值，否则sum就加上当前值  
 sum = sum <= 0 ? val : sum + val;  
 //每次计算出sum都要更新最大sum  
 greatestSum = Math.max(greatestSum, sum);  
 }  
 return greatestSum;  
 }  
}

***//数组 输入一个正整数数组，把数组所有数字拼成一个数，打印最小的一个。如输入数{3，32，321}，则打印的最小数字为321323***public class \_48 {  
 public String PrintMinNumber(int[] numbers) {  
 //如果数组为空，返回空  
 if (numbers == null || numbers.length == 0) return "";  
 int n = numbers.length; //数组长度  
 String[] nums = new String[n]; //构造字符数组  
 //遍历数组，将数组转变成字符串  
 for (int i = 0; i < n; i++)  
 nums[i] = numbers[i] + "";  
 //比较s1+s2和s2+s1，如果s2+s1大，就返回s1，s2顺序  
 Arrays.sort(nums, (s1, s2) -> (s1 + s2).compareTo(s2 + s1));  
 String ret = "";  
 //遍历排序后的字符串数组，拼接  
 for (String str : nums)  
 ret += str;  
 return ret;  
 }  
}

***//数组 一个整型数组里除两个数外，其他数都出现两次，找出这两个数***public class \_59 {  
 public void FindNumsAppearOnce(int[] nums, int num1[], int num2[]) {  
 int diff = 0; //初始diff是0  
 //遍历所有数字，做异或，相同数字做异或都成了0  
 for (int num : nums)  
 diff ^= num;  
 //找到不同的两个元素不一致的那位  
 diff &= -diff;  
 for (int num : nums) {  
 if ((num & diff) == 0) num1[0] ^= num;  
 else num2[0] ^= num;  
 }  
 }  
}

***//数组 数组中两个数如果前大于后则这两个数组成逆序对。输入一数组，求逆序对数***public class \_53 {  
 private long cnt = 0;//记录结果  
 private int[] tmp; //声明辅助数组  
  
 public int InversePairs(int[] nums) {  
 tmp = new int[nums.length];//根据输入数组大小构造辅助数组  
 mergeSort(nums, 0, nums.length - 1);//归并排序  
 return (int) (cnt % 1000000007);//返回结果  
 }  
  
 private void mergeSort(int[] nums, int l, int h) {  
 if (h - l < 1) return; //如果高低位相同，直接返回  
 int m = l + (h - l) / 2; //计算l和h的中间位置  
 mergeSort(nums, l, m); //排序l到m  
 mergeSort(nums, m + 1, h); //排序m+1到h  
 merge(nums, l, m, h);//合并l m h  
 }  
  
 private void merge(int[] nums, int l, int m, int h) {  
 int i = l, j = m + 1, k = l;//i和k指向l，j指向m+1  
 while (i <= m || j <= h) {//i≤m或j≤h  
 if (i > m) //如果i>m，tmp[k] = nums[j++]  
 tmp[k] = nums[j++];  
 else if (j > h) //如果j大于h，tmp[k] = nums[i++]  
 tmp[k] = nums[i++];  
 else if (nums[i] <= nums[j])//如果num[i]≤num[j]，tmp[k] = nums[i++]  
 tmp[k] = nums[i++];  
 else { //否则tmp[k] = nums[j++]  
 tmp[k] = nums[j++];  
 this.cnt += m - i + 1; //增加m-i+1  
 }  
 k++; //更新k  
 }  
 for (k = l; k <= h; k++) //遍历l到h，更新nums为tmp值  
 nums[k] = tmp[k];  
 }  
}

***//数组 输入一个递增数组和数字S，在数组中找两个数和是S。如果有多对输出乘积最小的***public class \_60 {  
 public ArrayList<Integer> FindNumbersWithSum(int[] array, int sum) {  
 int i = 0, j = array.length - 1;  
 while (i < j) { //双指针分别指向首尾  
 int cur = array[i] + array[j]; //计算当前两个指针指向数字的和  
 //找到了直接返回，因为两个数字和固定时，两个数字差越大，乘积越小  
 if (cur == sum)  
 return new ArrayList<>(Arrays.asList(array[i], array[j]));  
 if (cur < sum) i++; //如果当前和小于sum，要移动左指针  
 else j--; //否则移动右指针  
 } //最后都没找到就返回空数组  
 return new ArrayList<>();  
 }  
}

***//数组 五张牌，其中大小鬼为癞子牌为 0。判断五张牌是否能组成顺子***public class \_66 {  
 public boolean isContinuous(int[] nums) {  
 if (nums.length < 5) return false; //如果牌数小于5，返回false  
 Arrays.sort(nums); //将数组排序  
 int cnt = 0; // 统计癞子数量  
 for (int num : nums)  
 if (num == 0)  
 cnt++;  
 // 使用癞子补全不连续的顺子  
 for (int i = cnt; i < nums.length - 1; i++) {  
 //遍历除了癞子后面的数字，只要有重复的直接返回false  
 if (nums[i + 1] == nums[i]) return false;  
 //否则就用癞子填充两个数字之间的间隔  
 cnt -= nums[i + 1] - nums[i] - 1;  
 }  
 //如果最后癞子牌替换之后还是大于等于0的，说明能成顺子  
 return cnt >= 0;  
 }  
}

***//数组 股票最大利润，可以有一次买入和一次卖出，买入必须在前求最大收益***public class \_68 {  
 public int maxProfit(int[] prices) {  
 //如果数组为空，返回0  
 if (prices == null || prices.length == 0) return 0;  
 int soFarMin = prices[0]; //至此为止的最小值  
 int maxProfit = 0; //最大价值  
 for (int i = 1; i < prices.length; i++) {  
 //计算至此为止的最小值  
 soFarMin = Math.min(soFarMin, prices[i]);  
 //计算当前的最大收益，计算当前值-至此为止最小值，如果这个值比当前最大收益大，就更新当前最大收益  
 maxProfit = Math.max(maxProfit, prices[i] - soFarMin);  
 }  
 return maxProfit;  
 }  
}

***//数组 给一数组 A[0, 1,.,n-1]，构建数组B[0, 1,.,n-1]，其中B[i]=A中所有元素乘机除了A[i]，不能用除法***public class \_71 {  
 public static int[] multiply(int[] A) {  
 int n = A.length; //获取A数组长度  
 int[] B = new int[n]; //构造B数组  
 //因为product \*= A[i]写在循环的第三部分，所以每一轮都会product都在乘前一轮的A[i]  
 for (int i = 0, product = 1; i < n; product \*= A[i], i++) //这轮循环得到的B[i]都乘了i左边数  
 B[i] = product;  
 for (int i = n - 1, product = 1; i >= 0; product \*= A[i], i--)//这轮循环得到的B[i]继续乘了i右边数  
 B[i] \*= product;  
 return B; //所以最后得到的B就是没乘i本身，其余数都乘过了的数组  
 }  
}

***//数组 两个m和n的正序数组nums1和nums2。返回这两个正序数组的中位数***public class \_4 {  
 public double findMedianSortedArrays(int[] nums1, int[] nums2) {  
 int length1 = nums1.length, length2 = nums2.length; //计算两个数组的长度之和  
 int totalLength = length1 + length2;  
 if (totalLength % 2 == 1) { //如果总长度为奇数，则k=midIndex + 1  
 int midIndex = totalLength / 2;  
 return getKthElement(nums1, nums2, midIndex + 1);  
 } else { //如果为偶数，就是midIndex1 + 1和midIndex2 + 1的平均数  
 int midIndex1 = totalLength / 2 - 1, midIndex2 = totalLength / 2;  
 return (getKthElement(nums1, nums2, midIndex1 + 1) + getKthElement(nums1, nums2, midIndex2 + 1)) / 2.0;  
 }  
 }  
 public int getKthElement(int[] nums1, int[] nums2, int k) {  
 /\* 要找第k小元素，就取 pivot1=nums1[k/2-1]和pivot2=nums2[k/2-1]比较  
 \* nums1 中小于等于pivot1元素共计 k/2-1 个 nums2 中小于等于pivot2元素共计 k/2-1 个  
 \* 取 pivot = min(pivot1, pivot2)，两数组中小于等于 pivot 元素不超过(k/2-1)+(k/2-1)<= k-2 个  
 \* 这样pivot本身最大只能是第 k-1 小元素  
 \* 如果 pivot = pivot1，那么 nums1[0 .. k/2-1]全部 "删除"，剩下的作为新nums1 数组  
 \* 如果 pivot = pivot2，那么 nums2[0 .. k/2-1]全部 "删除"，剩下的作为新nums2 数组  
 \* 由于"删除"一些元素，因此要修改k，减去删除的数个数  
 int length1 = nums1.length, length2 = nums2.length; //获取两数组各自长度  
 int index1 = 0, index2 = 0; int kthElement = 0;  
 while (true) {  
 if (index1 == length1) { // 边界情况 如果index1越界了，就返回num2的第k-1个数  
 return nums2[index2 + k - 1];  
 } //如果index2越界了，就返回num1的第k-1个数  
 if (index2 == length2) {  
 return nums1[index1 + k - 1];  
 } //如果k=1，就返回两个数组队首元素的较小值  
 if (k == 1) {  
 return Math.min(nums1[index1], nums2[index2]);  
 } // 正常情况 更新index1和index2，获取两个数组对应位置的值  
 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];  
 //如果num1≤num2，更新k和index1的值  
 if (pivot1 <= pivot2) {  
 k -= (newIndex1 - index1 + 1);  
 index1 = newIndex1 + 1;  
 } else {  
 //否则更新k和index2的值  
 k -= (newIndex2 - index2 + 1);  
 index2 = newIndex2 + 1;  
 }  
 }  
 }  
}

***//数组 长n整数数组height，第i条线端点(i, 0)和(i, height[i])，找两条线使它们与x轴构成容器。返回最大水量***public class \_11 {  
 public int maxArea(int[] height) {  
 int l = 0, r = height.length - 1; //确定左右指针  
 int ans = 0;  
 while (l < r) { //循环判断条件  
 //计算当前的面积，左右高度最小值\*边界差  
 int area = Math.min(height[l], height[r]) \* (r - l);  
 ans = Math.max(ans, area); //更新最大面积  
 if (height[l] <= height[r]) { //移动高度更小那边的指针  
 ++l;  
 } else {  
 --r;  
 }  
 }  
 return ans;  
 }  
}

***//数学 整数转罗马数，罗马数含七种字符：I， V， X， L，C，D 和 M。分别是1，5，10，50，100，500，1000***public class \_12 {  
 int[] values = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1}; //保存1-4-5-9到1000整数数值  
 String[] symbols = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"}; //保存对应罗马数  
  
 public String intToRoman(int num) {  
 StringBuffer roman = new StringBuffer(); //构造结果字符串  
 for (int i = 0; i < values.length; ++i) { //遍历values数组  
 int value = values[i];  
 String symbol = symbols[i];  
 //取出小于num的最大数，注意这里要用while，因为num-value之后可能还大于value  
 while (num >= value) {  
 num -= value;  
 roman.append(symbol);  
 }  
 if (num == 0) break;//num为0是退出条件  
 }  
 return roman.toString();  
 }  
}

***//数组 n个整数数组nums和目标值target，找出和位target的所有不重复的四元组***public class \_18 {  
 public List<List<Integer>> fourSum(int[] nums, int target) {  
 List<List<Integer>> quadruplets = new ArrayList<List<Integer>>(); //存放结果数组  
 if (nums == null || nums.length < 4) { //如果数组长度小于4，返回空数组  
 return quadruplets;  
 }  
 Arrays.sort(nums); //传入数组排序  
 int length = nums.length; //获取传入数组长度  
 for (int i = 0; i < length - 3; i++) { //i从0到length-3防止越界，因为i后面还要有3个数  
 if (i > 0 && nums[i] == nums[i - 1]) continue; //去重  
 //如果从i开始四个最小数和大于target，直接退出循环  
 if ((long) nums[i] + nums[i + 1] + nums[i + 2] + nums[i + 3] > target) break;  
 //如果从i加上其余最大3个数和小于target，说明当前i值太小，进行下一轮循环  
 if ((long) nums[i] + nums[length - 3] + nums[length - 2] + nums[length - 1] < target) continue;  
 for (int j = i + 1; j < length - 2; j++) { //j从i+1到length-2  
 if (j > i + 1 && nums[j] == nums[j - 1]) continue; //去重  
 //如果i+从j开始3个最小数和大于target，直接退出循环  
 if ((long) nums[i] + nums[j] + nums[j + 1] + nums[j + 2] > target) break;  
 //如果i+j再+最大2数和小于target，说明当前j值太小，进入下一轮循环  
 if ((long) nums[i] + nums[j] + nums[length - 2] + nums[length - 1] < target) continue;  
 int left = j + 1, right = length - 1; //初始化左右下标  
 while (left < right) { //循环判断  
 int sum = nums[i] + nums[j] + nums[left] + nums[right]; //求四个数和  
 if (sum == target) { //如果等于target，将四个数放入结果集  
 quadruplets.add(Arrays.asList(nums[i], nums[j], nums[left], nums[right]));  
 while (left < right && nums[left] == nums[left + 1]) left++; //去重left  
 left++; //更新left  
 while (left < right && nums[right] == nums[right - 1]) right--; //去重right  
 right--; //更新right  
 } else if (sum < target) left++; //如果和太小，增大left  
 else right--; //如果和太大，减小right  
 }  
 }  
 }  
 return quadruplets; //返回结果数组  
 }  
}

***//数组 n个整数的数组nums，找出其中所有和为 0 且不重复的三元组。***public class \_15 {  
 public static List<List<Integer>> threeSum(int[] nums) {  
 List<List<Integer>> ans = new ArrayList(); //存放结果，结果是一个双层数组  
 int len = nums.length; //获取传入数组长度  
 if (len < 3) return ans; //如果数组不足3个元素，直接返回空结果  
 Arrays.sort(nums); // 对数组排序  
 for (int i = 0; i < len; i++) {  
 if (nums[i] > 0) break; // 如果当前数字大于0，则三数之和一定大于0，所以结束循环  
 if (i > 0 && nums[i] == nums[i - 1]) continue; // 去重  
 int L = i + 1; //从i+1到len-1找另外两个数  
 int R = len - 1;  
 while (L < R) { //循环判断条件  
 int sum = nums[i] + nums[L] + nums[R]; //求三个数的和  
 if (sum == 0) { //如果和为0，将三个数字加入结果集并且移动L和R去重  
 ans.add(Arrays.asList(nums[i], nums[L], nums[R]));  
 while (L < R && nums[L] == nums[L + 1]) L++; // 去重  
 while (L < R && nums[R] == nums[R - 1]) R--; // 去重  
 //去重后更新L和R继续找  
 L++; R--;  
 } else if (sum < 0) L++; //如果和小于0需要增大，就移动L  
 else R--; //如果和大于0需要减小，就移动R  
 }  
 }  
 return ans; //返回结果集  
 }  
}

***//数组 给正整数n，生成包含1到 n^2元素，元素顺时针螺旋排列的nxn矩阵***public class \_59 {  
 //和jzoffer的\_30操作相反  
 public int[][] generateMatrix(int n) {  
 int num = 1; //数字从1开始  
 int[][] matrix = new int[n][n]; //构造n\*n矩阵  
 //确定上下左右边界值  
 int left = 0, right = n - 1, top = 0, bottom = n - 1;  
 while (left <= right && top <= bottom) { //循环，左右和上下不能越界  
 //给最上面一行元素赋值  
 for (int column = left; column <= right; column++) {  
 matrix[top][column] = num;  
 num++;  
 } //最右侧一列元素赋值  
 for (int row = top + 1; row <= bottom; row++) {  
 matrix[row][right] = num;  
 num++;  
 } //最下一行元素和最左列要先判断不能越界，然后给最下一行元素赋值  
 if (left < right && top < bottom) {  
 for (int column = right - 1; column > left; column--) {  
 matrix[bottom][column] = num;  
 num++;  
 } //最左一列元素赋值  
 for (int row = bottom; row > top; row--) {  
 matrix[row][left] = num;  
 num++;  
 }  
 } //左++，右边界--，上++，下--，向内一层  
 left++; right--; top++; bottom--;  
 }  
 return matrix; //返回最终的矩阵  
 }  
}

***//数组 一个升序数组nums，原地删除重复元素，使每个元素只出现一次***public class \_26 {  
 public int removeDuplicates(int[] nums) {  
 //如果原数组为空，返回新长度是0  
 if (nums == null || nums.length == 0) return 0;  
 //数组有序那么重复元素一定相邻。删除重复元素就是将不重复元素移到数组左侧  
 int p = 0; //构造两个指针  
 int q = 1;  
 while (q < nums.length) { //循环判断q不能越界  
 //如果值不相等，将q位置元素复制到p+1位置，p和q都++  
 if (nums[p] != nums[q]) {  
 nums[p + 1] = nums[q];  
 p++;  
 } //如果p和q指向的数值相等，就后移q一位继续判断  
 q++;  
 } return p + 1; //最后返回p+1就是去重后的长度  
 }  
}

***//数组 长n的整数数组nums和目标值target。从nums中选三个数，使和与target最接近。返回三数和***public class \_16 {  
 public int threeSumClosest(int[] nums, int target) {  
 //这里假定给的数组长度都是大于等于3的，否则要跟上一题一样进行边界判断  
 Arrays.sort(nums); //先排序  
 int ans = nums[0] + nums[1] + nums[2]; //先让前三个数的和作为结果  
 for (int i = 0; i < nums.length - 2; i++) {  
 //i从0开始向右找到nums.length - 2，不能越界，因为i后面还有两个数  
 int start = i + 1, end = nums.length - 1; //初始化start和end，再找两个数  
 while (start < end) { //循环退出条件  
 int sum = nums[start] + nums[end] + nums[i]; //求三个数和  
 //如果sum更接近target就更新  
 if (Math.abs(target - sum) < Math.abs(target - ans)) ans = sum;  
 if (sum > target) end--; //如果sum太大就更新end  
 else if (sum < target) start++; //如果sum太小就更新start  
 else return ans; //否则sum=target，直接返回  
 }  
 }  
 return ans; //最后返回ans  
 }  
}

***//数组 判断9x9数独是否有效。1-9在每行、每列、每3x3宫内只能出现一次。验证已填数字是否有效。空白为'.'***public class \_36 {  
 public boolean isValidSudoku(char[][] board) {  
 //rows和columns记录每一行列每个数字出现次数，subboxes记录每个小九宫格中每个数字出现次数  
 int[][] rows = new int[9][9];  
 int[][] columns = new int[9][9];  
 int[][][] subboxes = new int[3][3][9];  
 for (int i = 0; i < 9; i++) {  
 for (int j = 0; j < 9; j++) {  
 //遍历数独二维矩阵每个元素  
 char c = board[i][j];  
 if (c != '.') { //如果有值就计算值，-1是因为数组下标从0开始  
 int index = c - '0' - 1;  
 //更新三个矩阵指定位置的值  
 rows[i][index]++;  
 columns[j][index]++;  
 subboxes[i / 3][j / 3][index]++;  
 //每次更新都判断，如果有出现>1的值就返回false  
 if (rows[i][index] > 1 || columns[j][index] > 1 || subboxes[i / 3][j / 3][index] > 1) {  
 return false;  
 }  
 }  
 }  
 }  
 return true; //如果通过了检测，返回true  
 }  
}

***//数组 一个未排序整数数组nums找出其中没出现的最小正整数***public class \_41 {  
 public int firstMissingPositive(int[] nums) {  
 int n = nums.length;  
 for (int i = 0; i < n; ++i) { //所有不在1到n的数设置为n+1  
 if (nums[i] <= 0) {  
 nums[i] = n + 1;  
 }  
 }  
 for (int i = 0; i < n; ++i) { //遍历获取数组的绝对值  
 int num = Math.abs(nums[i]); //将该位置数字标记为负数  
 if (num <= n) {  
 nums[num - 1] = -Math.abs(nums[num - 1]);  
 }  
 }//最后遍历，找到第一个正数位置，就是缺失的第一个正数  
 for (int i = 0; i < n; ++i) {  
 if (nums[i] > 0) {  
 return i + 1;  
 }  
 }  
 return n + 1; //如果找不到，缺失的第一个正数就是n+1  
 }  
}

***//数组 n个元素数组nums包含0，1，2，原地排序，使按照0、1和2顺序排序***public class \_75 {  
 public void sortColors(int[] nums) {  
 int len = nums.length;  
 if (len < 2) { //如果0或1个元素，直接返回  
 return;  
 }  
 //all in [0, zero) = 0，all in [zero, i) = 1，all in [two, len - 1] = 2  
 int zero = 0; //初始zero指向0，two指向末尾  
 int two = len;  
 int i = 0;  
 // 当i==two三个区间正好覆盖全部数组，因此循环继续的条件是i<two  
 while (i < two) {  
 if (nums[i] == 0) { //如果i位是0，就交换nums的i和zero位置并zero指针++  
 swap(nums, i, zero);  
 zero++;  
 i++;  
 } else if (nums[i] == 1) { //如果i位是1不需要动，直接++  
 i++;  
 } else { //如果数字是2，two指针--，交换i和two位置元素  
 two--;  
 swap(nums, i, two);  
 }  
 }  
 }  
  
 private void swap(int[] nums, int index1, int index2) {  
 int temp = nums[index1];  
 nums[index1] = nums[index2];  
 nums[index2] = temp;  
 }  
}

***//数组 返回整数数组下一个排列，如arr=[1,2,3]下一个排列是 [1,3,2]***public class \_31 {  
 public void nextPermutation(int[] nums) {  
 if (nums == null || nums.length == 0) return; //如果传入数组为空，直接返回  
 int firstIndex = -1;  
 //找到最大索引firstIndex满足nums[firstIndex]<nums[firstIndex+1]，从后往前  
 for (int i = nums.length - 2; i >= 0; i--) {  
 if (nums[i] < nums[i + 1]) {  
 firstIndex = i;  
 break;  
 }  
 }  
 if (firstIndex == -1) { //如果firstIndex是-1表示整个数组是逆序，要翻转整个数组  
 reverse(nums, 0, nums.length - 1);  
 return;  
 }  
 int secondIndex = -1;  
 //再找出另一个最大索引l满足 nums[l]>nums[k]，从后往前找  
 for (int i = nums.length - 1; i >= 0; i--) {  
 if (nums[i] > nums[firstIndex]) {  
 secondIndex = i;  
 break;  
 }  
 }  
 swap(nums, firstIndex, secondIndex);//交换nums[l]和nums[k]  
 //翻转k+1到末尾nums[k+1:]，最后更新后的nums就是要找的数组，如果要返回，就返回nums  
 reverse(nums, firstIndex + 1, nums.length - 1);  
 return;  
 }  
  
 //翻转nums的i到j位  
 private void reverse(int[] nums, int i, int j) {  
 while (i < j) { //依次翻转首尾元素，然后首尾不断靠近  
 swap(nums, i++, j--);  
 }  
 }  
  
 private void swap(int[] nums, int i, int i1) {  
 int tmp = nums[i];  
 nums[i] = nums[i1];  
 nums[i1] = tmp;  
 }  
}

***//数组 整数数组nums，返回数组answer，answer[i]等于nums中除nums[i]外其余元素乘积***public class \_238 {  
 public int[] productExceptSelf(int[] nums) {  
 int[] res = new int[nums.length]; //构造res结果数组  
 int p = 1, q = 1;//初始化p和q都是1，后面累乘  
 //先把下三角每行非1数字都乘到res中  
 for (int i = 0; i < nums.length; i++) {//i从0到len  
 res[i] = p;//res[i] = p  
 p \*= nums[i];//p\*= nums[i]  
 }  
 //再把上三角每行乘机乘到res中  
 for (int i = nums.length - 1; i > 0; i--) {//i从len到0  
 q \*= nums[i];//q \*= nums[i]  
 res[i - 1] \*= q;//res[i - 1] \*= q  
 }  
 return res;//返回res  
 }  
}

***//数组 数组nums和值val，原地移除所有数值val的元素，返回移除后数组新长度***public class \_27 {  
 public int removeElement(int[] nums, int val) {  
 int j = nums.length - 1;  
 //j<=j，因为j在前移，j后面的数都等于val是无效的  
 for (int i = 0; i <= j; i++) {  
 //如果i位置数=val，就把i和j位置的数交换，并且i位置不变重新判断一次，j前移一位  
 //因此要i--，因为每轮循环i会++  
 if (nums[i] == val) {  
 swap(nums, i--, j--);  
 }  
 }  
 return j + 1; //下标再+1才是元素个数  
 }  
 //交换i和j位置元素  
 void swap(int[] nums, int i, int j) {  
 int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;  
 }  
}

***//数组 n个非负整数表示宽度为1的柱子高度图，计算此排列柱子能接多少雨水***public class \_42 {  
 public int trap(int[] height) {  
 int ans = 0;  
 int size = height.length;  
 for (int i = 1; i < size - 1; i++) { //从1遍历到size-1  
 int max\_left = 0, max\_right = 0;  
 for (int j = i; j >= 0; j--) { //Search the left part for max bar size  
 //j从i向左搜索找到左边最大高度  
 max\_left = Math.max(max\_left, height[j]);  
 }  
 for (int j = i; j < size; j++) { //Search the right part for max bar size  
 //j从i向右搜索找到右边最大高度  
 max\_right = Math.max(max\_right, height[j]);  
 } //左右最小值-当前高度不断加到ans里  
 ans += Math.min(max\_left, max\_right) - height[i];  
 }  
 return ans; //返回ans  
 }  
}

***//数组 n×n二维矩阵matrix。将其顺时针旋转90度***public class \_48 {  
 public void rotate(int[][] matrix) {  
 int n = matrix.length; //获取行数  
 // 水平翻转  
 for (int i = 0; i < n / 2; ++i) { //i从0到n/2行  
 for (int j = 0; j < n; ++j) { //列号从0到n，因为是n\*n矩阵，行列数相等  
 //翻转i,j和n-i-1,j  
 int temp = matrix[i][j]; matrix[i][j] = matrix[n - i - 1][j]; matrix[n - i - 1][j] = temp;  
 }  
 }  
 // 主对角线翻转  
 for (int i = 0; i < n; ++i) { //i从0到n  
 for (int j = 0; j < i; ++j) { //j从0到i //ij翻转到ji  
 int temp = matrix[i][j]; matrix[i][j] = matrix[j][i]; matrix[j][i] = temp;  
 }  
 }  
 }  
}

***//数组 一副牌每张牌一个数。选数字X，将牌分1或多组：每组有X张相同数字牌。当可选X >= 2时返回true***public class \_914 {  
 public boolean hasGroupsSizeX(int[] deck) {  
 int N = deck.length;//计算牌的总数N  
 int[] count = new int[10000];//遍历每张牌，记录数字出现次数  
 for (int c : deck) {  
 count[c]++;  
 }  
 List<Integer> values = new ArrayList<Integer>();  
 for (int i = 0; i < 10000; ++i) {  
 if (count[i] > 0) {  
 values.add(count[i]);  
 }  
 }//将牌出现的次数加入values数组  
 for (int X = 2; X <= N; ++X) {//X从2到N  
 if (N % X == 0) {//如果X能被N整除  
 boolean flag = true;//默认能按X分组  
 //遍历每张牌出现次数  
 for (int v : values) {  
 //如果出现次数不能整除X，标记为false，判断下一个X  
 if (v % X != 0) {  
 flag = false;  
 break;  
 }  
 }//如果遍历完所有数字的次数都能整除X，返回true  
 if (flag) {  
 return true;  
 }  
 }  
 }  
 return false;//遍历完所有X没找到合理分组返回false  
 }  
}

***//数组 n个非负整数表示柱状图各柱子高度。柱子相邻且宽为1。求能勾勒出来的矩形最大面积***public class \_84 {  
 public int largestRectangleArea(int[] heights) {  
 int len = heights.length; //计算一共多少矩形  
 if (len == 0) return 0; //如果为空，返回0  
 if (len == 1) return heights[0]; //如果只有一个矩形，最大矩形就是该值  
 int res = 0; //记录结果  
 Deque<Integer> stack = new ArrayDeque<>(len); //构造len长度栈  
 for (int i = 0; i < len; i++) { //遍历所有矩形  
 //循环判断栈不为空且当前高度小于栈底元素下标对应高度  
 while (!stack.isEmpty() && heights[i] < heights[stack.peekLast()]) {  
 //弹出栈底元素，获取对应下标高度  
 int curHeight = heights[stack.pollLast()];  
 //循环判断栈不为空，如果栈底元素下标对应高度等于当前高度，就弹出栈底元素  
 while (!stack.isEmpty() && heights[stack.peekLast()] == curHeight) {  
 stack.pollLast();  
 }  
 int curWidth;  
 if (stack.isEmpty()) { //如果栈为空，宽度为i  
 curWidth = i;  
 } else { //否则宽度是i-栈底下标-1  
 curWidth = i - stack.peekLast() - 1;  
 } //计算当前矩形面积并和res中取较大值更新res  
 res = Math.max(res, curHeight \* curWidth);  
 }  
 stack.addLast(i); //将下标i加入栈底  
 }  
 //遍历完上面外循环，再一次循环，和上面内循环基本一样，区别①：判断条件没后半部分；②宽度为len不是i  
 while (!stack.isEmpty()) {  
 int curHeight = heights[stack.pollLast()];  
 while (!stack.isEmpty() && heights[stack.peekLast()] == curHeight) {  
 stack.pollLast();  
 }  
 int curWidth;  
 if (stack.isEmpty()) {  
 curWidth = len;  
 } else {  
 curWidth = len - stack.peekLast() - 1;  
 }  
 res = Math.max(res, curHeight \* curWidth);  
 }  
 return res; //最后返回res  
 }  
}

***//数组 一个未排序整数数组nums，找出连续最长序列（不要求序列元素在原数组中连续）长度***public class \_128 {  
 public int longestConsecutive(int[] nums) {  
 Set<Integer> num\_set = new HashSet<Integer>();  
 for (int num : nums) { //先去重  
 num\_set.add(num);  
 }  
 int longestStreak = 0; //保存最长连续序列  
 for (int num : num\_set) { //遍历数组  
 if (!num\_set.contains(num - 1)) { //如果数组不包含当前数值-1  
 int currentNum = num; //就将num设置为当前数  
 int currentStreak = 1; //找到一个最小的数，当前长度为1  
 //不停判断数组是否包含当前数的下一个数  
 while (num\_set.contains(currentNum + 1)) {  
 currentNum += 1; //如果包含，就当前数+1，当前长度+1  
 currentStreak += 1;  
 }//退出这个循环说明找到了一条最长连续数组，更新全局最大值  
 longestStreak = Math.max(longestStreak, currentStreak);  
 }  
 }  
 return longestStreak; //返回全局最大值  
 }  
}

***//数组 数组intervals表示区间集合，单区间intervals[i]=[starti, endi]。合并所有重叠区间返回不叠区间数组***public class \_56 {  
 public int[][] merge(int[][] intervals) {  
 if (intervals.length == 0) { //如果区间列表为空，返回0到2  
 return new int[0][2];  
 }  
 Arrays.sort(intervals, new Comparator<int[]>() {  
 //return 参数1-参数2 就是升序 将区间列表按左端点升序  
 public int compare(int[] interval1, int[] interval2) {  
 return interval1[0] - interval2[0];  
 }  
 });  
 List<int[]> merged = new ArrayList<int[]>(); //构造一个数组列表保存返回值  
 for (int i = 0; i < intervals.length; ++i) { //遍历传入数组  
 //获取第i个数组的左右端  
 int L = intervals[i][0], R = intervals[i][1];  
 //如果合并数组为空或者merge最后一个数组的右端小于左端，代表不能合并，要新构建一个LR加入merge  
 if (merged.size() == 0 || merged.get(merged.size() - 1)[1] < L) {  
 merged.add(new int[]{L, R});  
 } else {//否则说明能合并，把merge最后一个数组右端设置为R和自身中较大的数  
 merged.get(merged.size() - 1)[1] = Math.max(merged.get(merged.size() - 1)[1], R);  
 }  
 }  
 return merged.toArray(new int[merged.size()][]); //返回merge  
 }  
}

***//数组 一群人乱序站成队people，people[i]=[hi, ki]，第i人高hi，前面有ki个身高≥hi的人。  
//请返回真实队列顺序queue，其中 queue[j] = [hj,kj]  
//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]]***public class \_406 {  
 public int[][] reconstructQueue(int[][] people) {  
 //对原数组排序，按hi为第一关键字降序，ki为第二关键字升序  
 Arrays.sort(people, new Comparator<int[]>() {  
 public int compare(int[] person1, int[] person2) {  
 if (person1[0] != person2[0]) {//降序是var2-var1  
 return person2[0] - person1[0];  
 } else {//升序是var1-var2  
 return person1[1] - person2[1];  
 }  
 }  
 });  
 List<int[]> ans = new ArrayList<int[]>();//构造结果数组  
 for (int[] person : people) {//遍历排序后的people数组  
 //将person插入身高的index位置  
 ans.add(person[1], person);  
 }//将ans转换成数组返回  
 return ans.toArray(new int[ans.size()][]);  
 }  
}

***//数组 将数组元素向右轮转k个位置，k是非负数，例如[1,2,3,4,5,6,7], k=3，输出[5,6,7,1,2,3,4]***public class \_189 {  
 public void rotate(int[] nums, int k) {  
 //将数组元素向右移动k次，尾部k%n元素会移动至头部，其余元素向后移动k%n位置  
 k %= nums.length;  
 //先整体翻转，再翻转0到k-1，再翻转k到n-1  
 reverse(nums, 0, nums.length - 1);  
 reverse(nums, 0, k - 1);  
 reverse(nums, k, nums.length - 1);  
 }  
 //翻转nums的start到end  
 public void reverse(int[] nums, int start, int end) {  
 while (start < end) { //循环，每次交换start和end  
 int temp = nums[start];  
 nums[start] = nums[end];  
 nums[end] = temp;  
 start += 1;//靠近start和end  
 end -= 1;  
 }  
 }  
}

***//数组 一个数组nums，将所有0移到数组末尾保持非零元素相对顺序***public class \_283 {  
 public void moveZeroes(int[] nums) {  
 //计算数组长度，左右指针最初都指向0  
 int n = nums.length, left = 0, right = 0;  
 while (right < n) {  
 if (nums[right] != 0) {//右指针指向非零数，将左右指针数交换  
 swap(nums, left, right);  
 left++; //同时左指针右移  
 }  
 right++; //右指针右移  
 }  
 }  
  
 public void swap(int[] nums, int left, int right) {  
 int temp = nums[left]; nums[left] = nums[right]; nums[right] = temp;  
 }  
}

***//数组 无限大XY平面走，(0,0)出发向北。可接收三种命令：-2左转90度; -1右转90度;1<= x <= 9 ：向前移动x  
//有些障碍物obstacles[i]=(xi, yi)。会停在障碍物前一个网格。返回从原点到机器人所有经过路径点的最大距离平方***public class \_874 {  
 public int robotSim(int[] commands, int[][] obstacles) {  
 int[] dx = new int[]{0, 1, 0, -1};//组合dx和dy代表方向  
 int[] dy = new int[]{1, 0, -1, 0};  
 int x = 0, y = 0, di = 0;//保存x，y，di代表同时取出dx[di]和dy[di]两个数代表的方向，分别是北、东、南、西  
 // Encode obstacles (x, y) as (x+30000) \* (2^16) + (y+30000)  
 //为了将障碍点xy二维数组转换成一维的编码code，放入code集合，后面容易判断  
 Set<Long> obstacleSet = new HashSet();  
 for (int[] obstacle : obstacles) {  
 long ox = (long) obstacle[0] + 30000;  
 long oy = (long) obstacle[1] + 30000;  
 obstacleSet.add((ox << 16) + oy);  
 }  
 int ans = 0;//保存最大距离答案  
 for (int cmd : commands) {//遍历命令  
 if (cmd == -2) //左转90度是di+3之后对4取余  
 di = (di + 3) % 4;  
 else if (cmd == -1) //右转90度是di+1之后对4取余  
 di = (di + 1) % 4;  
 else {//否则是数字，代表要走路  
 for (int k = 0; k < cmd; ++k) {//针对每个小于x的整数，都要计算其距离，因为最大距离可能在走路途中达到  
 int nx = x + dx[di];//计算走一步之后的坐标  
 int ny = y + dy[di];  
 //计算坐标的编码值  
 long code = (((long) nx + 30000) << 16) + ((long) ny + 30000);  
 if (!obstacleSet.contains(code)) {//判断障碍列表中如果不包含该编码  
 x = nx;//更新x和y，计算距离平方，更新最大ans  
 y = ny;  
 ans = Math.max(ans, x \* x + y \* y);  
 }  
 }  
 }  
 }  
 return ans;//返回ans  
 }  
}

***//数组 数组nums，请完成两类查询。一类要更新数组下标的值,另一类要返回数组中索引left和right之间元素的和***public class \_309 {  
 //设数组大小n，将数组分成多块，每块大小size，最后一块大小为剩余的不超过size元素数  
 class NumArray {  
 private int[] sum; // sum[i]表示第i个块的元素和  
 private int size; // 块的大小  
 private int[] nums;//存放所有数字的数组  
  
 public NumArray(int[] nums) {  
 this.nums = nums;//初始化nums  
 int n = nums.length;  
 size = (int) Math.sqrt(n);//size等于√n  
 //n先+size-1再除以size就是n/size 向上取整，初始化sum数组  
 sum = new int[(n + size - 1) / size];  
 for (int i = 0; i < n; i++) {  
 //遍历num[i]，更新sum数组  
 sum[i / size] += nums[i];  
 }  
 }  
 //更新方法  
 public void update(int index, int val) {  
 //同时更新sum和nums  
 sum[index / size] += val - nums[index];  
 nums[index] = val;  
 }  
 //求和方法  
 public int sumRange(int left, int right) {  
 //计算出left落在哪块哪个位置，right落在哪块哪个位置：b1 i1 b2 i2  
 int b1 = left / size, i1 = left % size, b2 = right / size, i2 = right % size;  
 if (b1 == b2) { //如果[left, right] 在同一块  
 int sum = 0;  
 for (int j = i1; j <= i2; j++) {  
 //就从b1块的i1遍历到i2求和返回  
 sum += nums[b1 \* size + j];  
 }  
 return sum;  
 }//否则说明包含多个块  
 int sum1 = 0;//计算b1块中需要求的和，从i1开始到b1块末尾求和  
 for (int j = i1; j < size; j++) {  
 sum1 += nums[b1 \* size + j];  
 }//计算b2块中需要求的和，从0开始到i2位置求和  
 int sum2 = 0;  
 for (int j = 0; j <= i2; j++) {  
 sum2 += nums[b2 \* size + j];  
 }//计算b1和b2之间的整块和  
 int sum3 = 0;//从b1+1到b2-1  
 for (int j = b1 + 1; j < b2; j++) {  
 sum3 += sum[j];  
 }//上面三次求的sum再求和返回  
 return sum1 + sum2 + sum3;  
 }  
 }  
}

***//数组 字符串s和整数k，每2k字符反转前k。如剩字符少于k全反转。如小于2k但大于等于k则反转前k字符***public class \_541 {  
 public String reverseStr(String s, int k) {  
 int n = s.length();//计算字符串长度n  
 char[] arr = s.toCharArray();//字符串转换为字符数组  
 for (int i = 0; i < n; i += 2 \* k) {//i+=2k  
 //翻转从i到i+k(不包含)，注意判断i+k和n的较小值再-1  
 reverse(arr, i, Math.min(i + k, n) - 1);  
 }  
 return new String(arr);//返回翻转后的字符串  
 }  
 //翻转字符串的left到right元素  
 public void reverse(char[] arr, int left, int right) {  
 while (left < right) {//不断替换首尾元素，指针逐渐靠近  
 char temp = arr[left];  
 arr[left] = arr[right];  
 arr[right] = temp;  
 left++;  
 right--;  
 }  
 }  
}

***//数组 整数数组temperatures表示温度，返回数组answer，answer[i]指对第i天，下个更高温在几天后。如果在这之后都不升温则为0***public class \_739 {  
 public int[] dailyTemperatures(int[] temperatures) {  
 int length = temperatures.length;//计算温度列表天数  
 int[] ans = new int[length];//构造结果数组  
 //存储下标的单调栈，栈底到栈顶温度递减。如果一下标在栈里表示尚未找到下一温度更高下标  
 Deque<Integer> stack = new LinkedList<Integer>();//构造栈  
 for (int i = 0; i < length; i++) {  
 //遍历每天的温度  
 int temperature = temperatures[i];  
  
 //如果栈不为空，比较栈顶下标的温度和当前温度，如果当前温度高于栈顶下标温度，将栈顶元素移除  
 //ans[prevIndex] = i - prevIndex，直到栈空或栈顶元素温度大于等于当前温度，将i进栈  
 while (!stack.isEmpty() && temperature > temperatures[stack.peek()]) {  
 int prevIndex = stack.pop();  
 ans[prevIndex] = i - prevIndex;  
 }  
 stack.push(i);//如果栈为空，则直接将 i 进栈  
 }  
 return ans;//返回ans  
 }  
}

***//数组 整数数组nums，找出所有该数组中不同的递增子序列，至少有两个元素，两数相等视作递增***public class \_491 {  
 List<Integer> temp = new ArrayList<Integer>(); //存放一种子序列，每找下一个子序列都清空它  
 List<List<Integer>> ans = new ArrayList<List<Integer>>();//用于存放结果  
 Set<Integer> set = new HashSet<Integer>(); //存放所有已经出现序列的哈希值  
 int n; //保存输入数组长度  
  
 public List<List<Integer>> findSubsequences(int[] nums) {  
 n = nums.length;//计算输入数组长度  
 //长度n的序列选子序列一共有2^n种  
 for (int i = 0; i < (1 << n); ++i) {  
 findSubsequences(i, nums);//根据标记位i从nums中找子序列  
 //要解决子序列去重问题。每找到一个合法序列都计算序列的哈希  
 int hashValue = getHash(263, (int) 1E9 + 7);  
 //如果子序列合法并且哈希值集合中不包含该子序列的哈希  
 if (check() && !set.contains(hashValue)) {  
 //就将子序列加入ans中，将哈希值加入set中  
 ans.add(new ArrayList<Integer>(temp));  
 set.add(hashValue);  
 }  
 }  
 return ans; //最后返回ans  
 }  
  
 public void findSubsequences(int mask, int[] nums) {  
 temp.clear();//清除子序列  
 for (int i = 0; i < n; ++i) {//i从0到n  
 if ((mask & 1) != 0) {//如果mask这位是1，就将nums[i]加入集合  
 temp.add(nums[i]);  
 }  
 mask >>= 1;//标记位右移一位  
 }  
 }  
 //计算哈希  
 public int getHash(int base, int mod) {  
 int hashValue = 0;  
 for (int x : temp) {//遍历子序列每个整数，不断更新哈希值  
 hashValue = hashValue \* base % mod + (x + 101);  
 hashValue %= mod;  
 }  
 return hashValue;  
 }  
 //检查子序列是否递增  
 public boolean check() {  
 for (int i = 1; i < temp.size(); ++i) {  
 //从i开始遍历子序列每个数值，如果值小于前一个值返回false  
 if (temp.get(i) < temp.get(i - 1)) {  
 return false;  
 }  
 }//保证递增，最后判断子序列是否至少含2个数  
 return temp.size() >= 2;  
 }  
}

***//数组 一个整数数组nums和整数k，统计数组中和为k的连续子数组个数***public class \_560 {  
 public int subarraySum(int[] nums, int k) {  
 int count = 0;//记录个数  
 //要统计符合条件下标j个数，其中0≤j≤i 且 [j..i]子数组和为k  
 //start从0到数组末尾，start相当于i  
 for (int start = 0; start < nums.length; ++start) {  
 int sum = 0;//求子数组的和  
 for (int end = start; end >= 0; --end) {//end从start往前，end相当于j  
 sum += nums[end];//不断累加sum  
 //如果某个[j..i]和为k，count++，还要继续向前找下一个符合条件j，  
 //因为可能前面还有一个-1和1，抵消掉之后和还是k  
 if (sum == k) {  
 count++;  
 }  
 }  
 }  
 return count;//返回记录数  
 }  
}

***//数组 含n个整数的数组nums，nums[i]在[1, n]内。找出所有[1, n]内但没出现在nums中的数字，数组返回***public class \_448 {  
 public List<Integer> findDisappearedNumbers(int[] nums) {  
 int n = nums.length;//计算数组长度n  
 for (int num : nums) {//遍历数组每个数字  
 //遇到x，就让nums[x−1]加n。增加后这些数大于n代表存在  
 int x = (num - 1) % n;  
 nums[x] += n;  
 }  
 List<Integer> ret = new ArrayList<Integer>();//构造结果集  
 for (int i = 0; i < n; i++) {//遍历nums每个位置的数  
 if (nums[i] <= n) {//如果nums[i]<=n说明i+1这个数字没出现过，加入ret  
 ret.add(i + 1);  
 }  
 }  
 return ret;//返回结果集  
 }  
}

***//数组 整数数组nums，找连续子数组，如果对这个子数组升序那么整个数组都升序，找出最短子数组输出长度***public class \_581 {  
 public int findUnsortedSubarray(int[] nums) {  
 //把数组分三段，左和右段升序，中段虽无序但最小值大于左段最大值，最大值小于右段最小值。  
 //找中段左右边界begin和end;从左到右维护max,进入右段前，遍历到的nums[i]都小于max，  
 //end就是最后一个小于max元素的位置；从右到左维护min，进入左段前nums[i]都大于min，  
 //begin是最后一个大于min元素的位置  
 int len = nums.length;//计算初始数组长度  
 int min = nums[len - 1]; //从右到左找min  
 int max = nums[0];//从左到右找max  
 int begin = 0, end = -1;//初始化中段左右边界  
 for (int i = 0; i < len; i++) { //遍历  
 if (nums[i] < max) {//从左到右，如果nums[i]小于max，更新end为i  
 end = i;  
 } else {//否则更新最大值  
 max = nums[i];  
 }  
 if (nums[len - i - 1] > min) {//从右到左，如果nums[i]大于min，更新begin为i  
 begin = len - i - 1;  
 } else {//否则更新min值  
 min = nums[len - i - 1];  
 }  
 }  
 return end - begin + 1;//返回end-begin+1就是数组长度  
 }  
}

***//数组 两数组arr1和arr2，arr2元素各不相同，arr2每个元素都出现在arr1中。对arr1排序使项顺序和arr2相同。未arr2出现元素升序放arr1末尾***public class \_1122 {  
 public int[] relativeSortArray(int[] arr1, int[] arr2) {  
 int upper = 0;  
 for (int x : arr1) {//找到arr1中的最大值upper  
 upper = Math.max(upper, x);  
 }//构造出现频率数组，frequency[i]代表数字i出现次数  
 int[] frequency = new int[upper + 1];  
 for (int x : arr1) {  
 ++frequency[x];  
 }  
 int[] ans = new int[arr1.length];//构造答案数组  
 int index = 0;//记录ans下标  
 for (int x : arr2) {//遍历arr2每个数字  
 //获取数字x出现次数frequency[x]，在ans中加入对应次数个x  
 for (int i = 0; i < frequency[x]; ++i) {  
 ans[index++] = x;  
 }  
 frequency[x] = 0;//执行完将x出现次数置0  
 }//然后把arr1中arr2中没出现的数字升序  
 //x从0到upper，将frequency[x]不为0的数字加入ans最后  
 for (int x = 0; x <= upper; ++x) {  
 for (int i = 0; i < frequency[x]; ++i) {  
 ans[index++] = x;  
 }  
 }  
 return ans;//返回ans  
 }  
}

# 字符串

***//将一个字符串中空格替换成 "%20"。***public class \_3 {  
 public String replaceSpace(StringBuffer str) {  
 int P1 = str.length() - 1; //p1指向尾部  
 for (int i = 0; i <= P1; i++) //遍历字符串，遇到空格就增加两个空位，因为空格替换成三个字符  
 if (str.charAt(i) == ' ')  
 str.append(" ");  
  
 int P2 = str.length() - 1; //让p2指向增长后的字符串尾部  
 while (P1 >= 0 && P2 > P1) {  
 char c = str.charAt(P1--); //P1从后向前遍历，获取字符  
 if (c == ' ') { //如果是空格，就在p2指向处增加0，2，%三个字符  
 str.setCharAt(P2--, '0');  
 str.setCharAt(P2--, '2');  
 str.setCharAt(P2--, '%');  
 } else {  
 str.setCharAt(P2--, c); //否则将字符放到p2处，p2--  
 }  
 }  
 return str.toString();  
 }  
}

***//字符串 把字符串前K位放到最后，例如S="abcXYZdef"，K=3；输出"XYZdefabc"***public class \_63 {  
 public String LeftRotateString(String str, int n) {  
 //如果n超出了字符串长度，直接返回原字符串  
 if (n >= str.length()) return str;  
 char[] chars = str.toCharArray(); //字符串转换为字符数组  
 reverse(chars, 0, n - 1); //翻转前K个字符的单词  
 reverse(chars, n, chars.length - 1); //翻转后面字符的单词  
 reverse(chars, 0, chars.length - 1); //整体翻转  
 return new String(chars);  
 }  
  
 private void reverse(char[] chars, int i, int j) {  
 while (i < j)  
 swap(chars, i++, j--);  
 }  
  
 private void swap(char[] chars, int i, int j) {  
 char t = chars[i]; chars[i] = chars[j]; chars[j] = t;  
 }  
}

***//表示数值的字符串***public class \_21 {  
 public boolean isNumeric(char[] str) {  
 if (str == null || str.length == 0)  
 return false;  
 //开头可以有0或者1个+/-，接下来必须是任意多个数字  
 //接下来必须是0或1个.加任意多数组  
 //接下来必须是e/E加0或1和+/-再跟1到多个数字的组合，并且该组合只能出现0或1次  
 return new String(str).matches("[+-]?\\d\*(\\.\\d+)?([eE][+-]?\\d+)?");  
 }  
}

***//字符串 输入一个字符串（只包含 a~z），求最长不含重复字符的子串长度***public class \_51 {  
 public static void main(String[] args) {  
 System.out.println(longestSubStringWithoutDuplication("arabcacfrz"));  
 }  
 public static int longestSubStringWithoutDuplication(String str) {  
 int curLen = 0; //curLen代表当前统计的最长字符长度 maxLen代表历史最大字符长度  
 int maxLen = 0;  
 int[] preIndexs = new int[26]; //构造26的整形数组，代表某字符上次出现的下标  
 Arrays.fill(preIndexs, -1); //整形数组初值设为-1  
 for (int curI = 0; curI < str.length(); curI++) {  
 //遍历字符串，计算每个字符和a的差值，因为只能是a-z，所以c>=0  
 int c = str.charAt(curI) - 'a';  
 int preI = preIndexs[c]; //获取上一次出现该字符的下标  
 //preI=-1代表当前字符之前没出现过，所以curLen要+1  
 //curI - preI > curLen代表虽然字符现在重复了，但上次出现的位置在当前统计字符串首前  
 //所以仍要curLen+1且不重置当前字符串头，如统计到"acfr"的r时发现r重复，但上一r下标是1  
 //“acfr”的字符串首下标是5，所以要把r加进来，curLen要++  
 if (preI == -1 || curI - preI > curLen) {  
 curLen++;  
 } else {  
 //每次遇到重复字符且curI-preI<=curLen时，就可能更新最大Len，  
 //且一定要重新设置当前字符串头  
 maxLen = Math.max(maxLen, curLen);  
 curLen = curI - preI;  
 }  
 preIndexs[c] = curI; //更新字符c上一次出现的位置  
 }  
 maxLen = Math.max(maxLen, curLen); //求最大长度  
 return maxLen;  
 }  
}

***//字符串 翻转单词顺序列，如将"I am a student."变为"student. a am I"***public class \_62 {  
 public static void main(String[] args) {  
 System.out.println(ReverseSentence("I am a student."));  
 }  
 public static String ReverseSentence(String str) {  
 //"I am a student."  
 int n = str.length(); //获取字符串总长度  
 char[] chars = str.toCharArray(); //字符串转换为字符数组  
 int i = 0, j = 0;  
 while (j <= n) { //遍历整个字符串，翻转每个单词  
 if (j == n || chars[j] == ' ') {  
 reverse(chars, i, j - 1);  
 i = j + 1;  
 }  
 j++;  
 }  
 //此时变为："I ma a .tneduts"，最后再整体翻转变为："student. a am I"  
 reverse(chars, 0, n - 1);  
 return new String(chars);  
 }  
  
 //翻转一个字符数组（单词）  
 private static void reverse(char[] c, int i, int j) {  
 while (i < j)  
 swap(c, i++, j--);  
 }  
  
 private static void swap(char[] c, int i, int j) {  
 char t = c[i]; c[i] = c[j]; c[j] = t;  
 }  
}

***//字符串 将字符串转换成整数，字符串不是合法数值则返回 0***public class \_72 {  
 public int StrToInt(String str) {  
 //如果字符串为空，返回0  
 if (str == null || str.length() == 0) return 0;  
 //如果字符串以-开头，就是负数  
 boolean isNegative = str.charAt(0) == '-';  
 int ret = 0;  
 for (int i = 0; i < str.length(); i++) {  
 //遍历字符数组，如果是最开始有+或者-，就代表符号，进行下一轮循环  
 char c = str.charAt(i);  
 if (i == 0 && (c == '+' || c == '-')) continue;  
 //如果字符小宇0或者大于9，就是非法字符，直接返回0  
 if (c < '0' || c > '9') return 0;  
 ret = ret \* 10 + (c - '0'); //否则不停更新结果  
 }  
 return isNegative ? -ret : ret; //返回带符号的结果  
 }  
}

***//字符串 将字符串s根据行数numRows从上往下、从左到右Z形排，输入"PAYPALISHIRING"行数3时如下：  
//P A H N  
//A P L S I I G  
//Y I R***public class \_6 {  
 public String convert(String s, int numRows) {  
 int n = s.length(), r = numRows;  
 if (r == 1 || r >= n) return s; //如果只排一行或排的行数超过字符串长度，返回原串  
 int t = r \* 2 - 2; //t=2r-2  
 int c = (n + t - 1) / t \* (r - 1); //c列，c=(n+t-1)/t\*(r−1)  
 char[][] mat = new char[r][c]; //构造存放数组，r行c列  
 for (int i = 0, x = 0, y = 0; i < n; ++i) {  
 mat[x][y] = s.charAt(i); //遍历原字符数组每个字符  
 if (i % t < r - 1) { //i mod t<r−1，向下移动  
 ++x;  
 } else { //否则向右上移动  
 --x; ++y;  
 }  
 }  
 StringBuffer ans = new StringBuffer();  
 for (char[] row : mat) { //遍历每一行，不是空字符就拼接到ans  
 for (char ch : row) {  
 if (ch != 0) {  
 ans.append(ch);  
 }  
 }  
 }  
 return ans.toString(); //返回ans  
 }  
}

***//字符串 将字符串转换成一个32位有符号整数，丢弃无用的前导空格***public class \_8 {  
 public int myAtoi(String str) {  
 int len = str.length();  
 char[] charArray = str.toCharArray();// 转换成字符数组  
 int index = 0;  
 while (index < len && charArray[index] == ' ') index++; // 1、去除前导空格  
 if (index == len) return 0; // 2、如果遍历完成（针对极端用例 " "）返回0  
 int sign = 1; // 3、如果出现符号字符，仅第1个有效，记录正负sign  
 char firstChar = charArray[index];  
 if (firstChar == '+') { index++;}

else if (firstChar == '-') { index++; sign = -1;}  
 int res = 0; // 4、将后续出现的数字字符进行转换，不能用 long 类型，这是题目说的  
 while (index < len) {  
 char currChar = charArray[index];  
 if (currChar > '9' || currChar < '0') break; // 4.1 先判断不合法的情况  
 // 题中说只能存储32位大小有符号数，提前判断乘10后是否越界  
 if (res > Integer.MAX\_VALUE / 10 || (res == Integer.MAX\_VALUE / 10 && (currChar - '0') > Integer.MAX\_VALUE % 10)) {  
 return Integer.MAX\_VALUE;  
 }  
 if (res < Integer.MIN\_VALUE / 10 || (res == Integer.MIN\_VALUE / 10 && (currChar - '0') > -(Integer.MIN\_VALUE % 10))) {  
 return Integer.MIN\_VALUE;  
 }  
 res = res \* 10 + sign \* (currChar - '0'); // 4.2合法情况下才转换，每步都把符号位乘进去  
 index++;  
 }  
 return res;//返回结果  
 }  
}

***//字符串 查找字符串数组中最长公共前缀。如不存在返回空字符串***public class \_14 {  
 public String longestCommonPrefix(String[] strs) {  
 if (strs == null || strs.length == 0) { //如果传入字符串数组位空，返回空  
 return "";  
 }  
 //随便找一个字符串长度遍历，如果strs[0]是最长的，遍历其他字符串时肯定不需要遍历这么长  
 //如果strs[0]是最短的，那最多只需要比较strs[0]的长度个位数  
 int length = strs[0].length();  
 int count = strs.length; //获取一共有多少个字符串  
 for (int i = 0; i < length; i++) { //从第0位开始纵向比较  
 char c = strs[0].charAt(i); //获取第一个字符串i位字符  
 for (int j = 1; j < count; j++) { //遍历所有字符串的第i位，如果达到了长度或者这一位字符不同，直接返回  
 if (i == strs[j].length() || strs[j].charAt(i) != c) {  
 return strs[0].substring(0, i);  
 }  
 }  
 }  
 return strs[0]; //如果遍历结束都没返回，说明strs[0]就是最短的且其余字符串前缀全包含strs[0]，返回strs[0]  
 }  
}

***//字符串 字符串haystack和needle，在haystack中找出needle出现的第一个位置（下标0开始）。不存在返回-1***public class \_28 {  
 public int strStr(String ss, String pp) {  
 int n = ss.length(), m = pp.length(); //计算ss和pp的字符串长度  
 char[] s = ss.toCharArray(), p = pp.toCharArray(); //将字符串转换为字符数组  
 // 枚举原串的「发起点」  
 for (int i = 0; i <= n - m; i++) {//<=n-m。因为p有m个字符，如果s剩余字符少于m个肯定无法匹配  
 // 从原串「发起点」和匹配串「首位」开始尝试匹配  
 int a = i, b = 0;  
 while (b < m && s[a] == p[b]) {  
 a++;  
 b++;  
 }  
 // 如果能够完全匹配，返回原串「发起点」下标  
 if (b == m) return i;  
 }  
 return -1;//无法匹配返回-1  
 }  
}

***//字符串 输出外观数列第n项。从1开始，序列中每项都是前项描述。前4项如下：1 11 21 1211***public class \_38 {  
 public String countAndSay(int n) {  
 String str = "1"; //构造结果字符串，因为数字从1开始，n=1时返回字符串1  
 for (int i = 2; i <= n; ++i) { //n从2开始遍历  
 StringBuilder sb = new StringBuilder(); //构造拼接字符串  
 int start = 0; //从start到pos是相同数字  
 int pos = 0;  
 while (pos < str.length()) { //pos记录当前已经判断到哪个位置，遍历str  
 //如果pos和start字符相同，右移pos  
 while (pos < str.length() && str.charAt(pos) == str.charAt(start)) {  
 pos++;  
 } //直到下一个和start不相同的字符  
 //先拼接pos-start，代表个数，然后拼接start处字符，代表字符，如1111对应41  
 sb.append(Integer.toString(pos - start)).append(str.charAt(start));  
 start = pos;//让start指向当前pos  
 } //退出循环代表完成了前一个字符串的描述串，更新str，继续描述当前更新的str  
 str = sb.toString();  
 }  
 return str; //返回最终的str  
 }  
}

***//字符串 将字符串数组的字母异位词组合。异位词是由重排列源单词的字母得到新单词，源单词字母恰好只用一次***public class \_49 {  
 public List<List<String>> groupAnagrams(String[] strs) {  
 Map<String, List<String>> map = new HashMap<String, List<String>>(); //key是排序后字符串，value是异位词  
 for (String str : strs) { //遍历字符串数组  
 char[] array = str.toCharArray(); //字符串转换成数组  
 Arrays.sort(array); //字符数组排序  
 String key = new String(array); //排序后的数组作为key  
 //从map中找这个key是否有value，没有就初始化一个list  
 List<String> list = map.getOrDefault(key, new ArrayList<String>());  
 list.add(str); //将str放入list  
 map.put(key, list); //将list和key放入map  
 }  
 return new ArrayList<List<String>>(map.values()); //最后返回map的value这个双重数组  
 }  
}

***//字符串 字符串s由若干单词组成，单词用一些空格隔开。返回最后一个单词长度***public class \_58 {  
 public int lengthOfLastWord(String s) {  
 int index = s.length() - 1; //获取传入字符串长度  
 while (s.charAt(index) == ' ') { //从后往前，如果字符是空格，index--  
 index--;  
 }  
 int wordLength = 0;  
 //从index开始往前，如果字符不是空格，就把单词长度+1，index--  
 while (index >= 0 && s.charAt(index) != ' ') {  
 wordLength++; index--;  
 }  
 return wordLength; //返回最后一个单词长度  
 }  
}

***//字符串 两个字符串s和t，判断是否同构。如果s中字符可按某种映射替换得到t，s和t就是是同构***public class \_205 {  
 public boolean isIsomorphic(String s, String t) {  
 //两个map分别表示s到t的字符映射和t到s的字符映射  
 Map<Character, Character> s2t = new HashMap<Character, Character>();  
 Map<Character, Character> t2s = new HashMap<Character, Character>();  
 int len = s.length(); //计算s的长度  
 for (int i = 0; i < len; ++i) {  
 //遍历s和t相同位置字符x和y  
 char x = s.charAt(i), y = t.charAt(i);  
 if ((s2t.containsKey(x) && s2t.get(x) != y) || (t2s.containsKey(y) && t2s.get(y) != x)) {  
 return false; //如果s2t映射中有x这个k但get(x)不是y；或t2s中包含y但get(y)不是x，映射失败返回false  
 }//否则将xy yx映射分别加入s2t和t2s  
 s2t.put(x, y);  
 t2s.put(y, x);  
 }  
 return true; //遍历完毕都匹配了，返回true  
 }  
}

***//字符串 两字符串s和t，判断t是否是s的字母异位词。若s和t中每个字符出现次数都相同则是***public class \_242 {  
 public boolean isAnagram(String s, String t) {  
 if (s.length() != t.length()) {//如果s和t长度不等，返回false  
 return false;  
 }  
 char[] str1 = s.toCharArray();//s和t转换成字符数组后排序  
 char[] str2 = t.toCharArray();  
 Arrays.sort(str1); Arrays.sort(str2);  
 return Arrays.equals(str1, str2);//判断排序后数组是否相同  
 }  
}

***//字符串 给一种规律pattern和一个字符串s ，判断 s 是否遵循相同规律***public class \_290 {  
 public boolean wordPattern(String pattern, String str) {  
 //建立两个映射表，字符串到字符，字符到字符串  
 Map<String, Character> str2ch = new HashMap<String, Character>();  
 Map<Character, String> ch2str = new HashMap<Character, String>();  
 int m = str.length(); //单词字符串长度m  
 int i = 0;//代表单词起始下标  
 for (int p = 0; p < pattern.length(); ++p) {//遍历模式串  
 char ch = pattern.charAt(p);//获取模式串每个位置字符  
 if (i >= m) {//如果单词起始下标超出单词字符串长度m，返回false  
 return false;  
 }  
 int j = i; //j是单词结束下标  
 while (j < m && str.charAt(j) != ' ') {//找到字符串中对应单词的结束下标  
 j++;  
 }  
 String tmp = str.substring(i, j); //获取单词  
 //如果str2ch映射包含该单词但该单词对应的字符不是ch返回false  
 if (str2ch.containsKey(tmp) && str2ch.get(tmp) != ch) {  
 return false;  
 }//如果ch2str映射包含该字符但该字符对应的单词不是tmp返回false  
 if (ch2str.containsKey(ch) && !tmp.equals(ch2str.get(ch))) {  
 return false;  
 }  
 str2ch.put(tmp, ch);//否则说明没找到映射，将映射关系放入两个表  
 ch2str.put(ch, tmp);  
 i = j + 1;//重置i位置为当前单词最后一个字母后一位  
 }  
 return i >= m;//返回i是否≥m，即单词字符串都被遍历完了  
 }  
}

***//字符串 将输入字符串反转。输入字符串以字符数组s形式***public class \_344 {  
 public void reverseString(char[] s) {  
 int n = s.length; //计算输入字符串长度  
 for (int left = 0, right = n - 1; left < right; ++left, --right) {  
 //不断交换首尾字符，首尾字符不断靠近  
 char tmp = s[left];  
 s[left] = s[right];  
 s[right] = tmp;  
 }  
 }  
}

***//字符串 一个字符串s，反转每个单词字符顺序，保留空格和单词初始顺序***public class \_557 {  
 public String reverseWords(String s) {  
 StringBuffer ret = new StringBuffer();//构造stringbuffer(sb)  
 int length = s.length();//计算初始字符串长度  
 int i = 0;  
 while (i < length) {//遍历字符串  
 int start = i; //标记单词起始位置  
 while (i < length && s.charAt(i) != ' ') {  
 i++;//让i递增到单词后的空格位置  
 }//指针从start到i就是一个单词的所有下标  
 for (int p = start; p < i; p++) {  
 //逆序将单词每个字符拼接到ret  
 ret.append(s.charAt(start + i - 1 - p));  
 }//将所有空格顺序拼接ret  
 while (i < length && s.charAt(i) == ' ') {  
 i++; ret.append(' ');  
 }  
 }  
 return ret.toString();//返回ret  
 }  
}

***//字符串 将字符串S中大写字母转换成相同的小写字母，返回新字符串***public class \_709 {  
 public String toLowerCase(String s) {  
 //用于存储新字符串  
 StringBuilder sb = new StringBuilder();  
 for (int i = 0; i < s.length(); ++i) {  
 //遍历原字符串每个字符  
 char ch = s.charAt(i);  
 //如果asc码是[65 90]，就+32，也就是|=32  
 if (ch >= 65 && ch <= 90) {  
 ch |= 32;  
 }  
 sb.append(ch);//将字符依次加入sb  
 }//返回转换后的字符串  
 return sb.toString();  
 }  
}

***//字符串 字符串s，非英文字母保留原位置。英文字母位置反转。返回反转后的s***public class \_917 {  
 public String reverseOnlyLetters(String s) {  
 int n = s.length();//计算字符串长度n  
 char[] arr = s.toCharArray();//字符串转换为字符数组  
 int left = 0, right = n - 1;//初始化左右指针  
 while (true) {//循环  
 //左指针跳过所有非字母字符  
 while (left < right && !Character.isLetter(s.charAt(left))) {  
 left++;  
 }//右指针跳过所有非字母字符  
 while (right > left && !Character.isLetter(s.charAt(right))) {  
 right--;  
 }//如果左大于等于右，跳出循环  
 if (left >= right) {  
 break;  
 }//否则交换左右字符，左右靠近一位  
 swap(arr, left, right); left++; right--;  
 }  
 return new String(arr);  
 }  
 public void swap(char[] arr, int left, int right) {  
 char temp = arr[left];  
 arr[left] = arr[right];  
 arr[right] = temp;  
 }  
}

***//字符串 字符串jewels代表宝石类型，字符串stones代表拥有的石头。计算拥有的石头中多少个宝石***public class \_771 {  
 public int numJewelsInStones(String jewels, String stones) {  
 int jewelsCount = 0;//初始化宝石数=0  
 Set<Character> jewelsSet = new HashSet<Character>();//构造宝石类型Set  
 //计算宝石列表和石头列表大小jl和sl  
 int jewelsLength = jewels.length(), stonesLength = stones.length();  
 for (int i = 0; i < jewelsLength; i++) {//遍历宝石字符串每个字符，将其加入宝石类型Set  
 char jewel = jewels.charAt(i);  
 jewelsSet.add(jewel);  
 }//再次遍历石头列表，取出每个字符，如果是宝石，就count++  
 for (int i = 0; i < stonesLength; i++) {  
 char stone = stones.charAt(i);  
 if (jewelsSet.contains(stone)) {  
 jewelsCount++;  
 }  
 }  
 return jewelsCount;//返回宝石数  
 }  
}

# 链表

***//链表 从尾到头打印链表***public class \_4 {  
 public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
 Stack<Integer> stack = new Stack<>(); //构造一个栈  
 while (listNode != null) { //将链表值依次放入栈  
 stack.add(listNode.val);  
 listNode = listNode.next;  
 }  
 ArrayList<Integer> ret = new ArrayList<>();  
 while (!stack.isEmpty()) //依次弹栈元素就是反向链表顺序  
 ret.add(stack.pop());  
 return ret;  
 }  
}

***//O(1)时间内删除链表节点***public class \_18 {  
 public ListNode deleteNode(ListNode head, ListNode tobeDelete) {  
 if (head == null || tobeDelete == null) return null; //如果头结点或者待删除节点为空，返回空  
 if (tobeDelete.next != null) {  
 // 如果要删除的节点不是尾节点，就获取待删除节点下一节点  
 ListNode next = tobeDelete.next;  
 //将next节点值赋给待删除节点，待删除的next指向next.next，就能跳过next  
 tobeDelete.val = next.val;  
 tobeDelete.next = next.next;  
 //将next的next指向空，就删除了next节点  
 next.next = null;  
 } else { //如果待删除节点是尾节点  
 if (head == tobeDelete)  
 head = null; // 如果待删除节点也是头结点，说明只有一个节点，删除该节点即可  
 else { //否则删除尾，从头往后找，直到cur.next是tobeDelete，将cur.next = null  
 ListNode cur = head;  
 while (cur.next != tobeDelete)  
 cur = cur.next;  
 cur.next = null;  
 }  
 }  
 return head; //最后返回原链表  
 }  
}

***//删除链表中重复结点***public class \_19 {  
 public ListNode deleteDuplication(ListNode pHead) {  
 //链表为空或者只有一个元素，肯定没有重复元素，直接返回原链表  
 if (pHead == null || pHead.next == null) return pHead;  
 ListNode next = pHead.next; //获取头下一个节点  
 if (pHead.val == next.val) { //如果头结点值和next值相等  
 //就不断跳值相等的节点，直到找到值不等的节点，递归删除该节点  
 while (next != null && pHead.val == next.val)  
 next = next.next;  
 return deleteDuplication(next);  
 } else {//如果头结点值和next不等，头的next就是递归删除头.next的节点  
 pHead.next = deleteDuplication(pHead.next);  
 return pHead;  
 }  
 }  
}

***//返回链表中倒数第 K 个结点***public class \_23 {  
 public ListNode FindKthToTail(ListNode head, int k) {  
 if (head == null) return null; //如果链表为空，返回空  
 //p1从前往后移动k个节点  
 ListNode P1 = head;  
 while (P1 != null && k-- > 0)  
 P1 = P1.next;  
 //如果k>0说明原链表不够k个节点，所以无法找到倒数第k个节点  
 if (k > 0) return null;  
 //否则让p2指向头节点，p1和p2同时向后移动，p1指向尾节点时，p2指向的就是倒数k节点  
 ListNode P2 = head;  
 while (P1 != null) {  
 P1 = P1.next;  
 P2 = P2.next;  
 }  
 return P2;  
 }  
}

***//一个链表包含环，请找出环入口结点。不能使用额外空间***public class \_24 {  
 public ListNode EntryNodeOfLoop(ListNode pHead) {  
 //空链表或者只有一个节点的链表不能成环，直接返回空  
 if (pHead == null || pHead.next == null)  
 return null;  
 ListNode slow = pHead, fast = pHead; //设置快慢两个指针  
 do { //让快慢指针第一次相遇  
 fast = fast.next.next;  
 slow = slow.next;  
 } while (slow != fast);  
 //此时让快指针指向头，再次让快慢指针相同速度移动，相遇处就是成环的地方  
 fast = pHead;  
 while (slow != fast) {  
 slow = slow.next;  
 fast = fast.next;  
 }  
 return slow;  
 }  
}

***//反转链表***public class \_25 {  
 public ListNode ReverseList(ListNode head) {  
 //如果为空或者只有一个节点，直接返回，因为翻转之后还是原链表  
 if (head == null || head.next == null)  
 return head;  
 //获取头的next，头的next指向空  
 ListNode next = head.next;  
 head.next = null;  
 ListNode newHead = ReverseList(next); //翻转next  
 //next的next指向头  
 next.next = head;  
 return newHead; //返回新的头  
 }  
}

***//输入两个递增链表，输出两链表合成后的链表，合成后的链表单调不减***public class \_26 {  
 public ListNode Merge(ListNode list1, ListNode list2) {  
 ListNode list = null; //构造结果链表  
 //如果list1和list2都是空，返回空  
 if (list1 == null && list2 == null) return null;  
 if (list1 == null) //list1为空返回list2，list2为空返回list1  
 return list2;  
 if (list2 == null)  
 return list1;  
 //如果list1的节点值比list2的值小，list的这个节点就是list1的节点，并将list1节点向后延一个继续merge  
 if (list1.val < list2.val) {  
 list = list1;  
 list.next = Merge(list1.next, list2);  
 } else {  
 //否则，list的这个节点就是list2的节点，并将list2节点向后延一个继续merge  
 list = list2;  
 list.next = Merge(list1, list2.next);  
 }  
 return list;  
 }  
}

***//链表 输入一个复杂链表，节点有节点值及两个指针，一个指向下一节点，另一个指向任意节点，返回复制后复杂链表***public class \_38 {  
 public class RandomListNode {  
 int label;  
 RandomListNode next = null;  
 RandomListNode random = null;  
  
 RandomListNode(int label) {  
 this.label = label;  
 }  
 }  
 public RandomListNode Clone(RandomListNode pHead) {  
 if (pHead == null) return null; //如果原链表为空，返回空  
 // 在原链表每个节点后面插入相同新节点  
 RandomListNode cur = pHead;  
 while (cur != null) {  
 RandomListNode clone = new RandomListNode(cur.label);  
 clone.next = cur.next;  
 cur.next = clone;  
 cur = clone.next;  
 }  
 cur = pHead; //建立random 链接  
 while (cur != null) {  
 RandomListNode clone = cur.next;  
 //如果有random，说明是原链表节点，就把当前节点的next（clone链表节点）的random  
 //设置为当前的节点的random的next  
 if (cur.random != null)  
 clone.random = cur.random.next;  
 cur = clone.next;  
 }  
 cur = pHead; //拆分  
 RandomListNode pCloneHead = pHead.next; //克隆头结点的next  
 while (cur.next != null) { //获取next，让cur的next指向next的next，cur指向next  
 RandomListNode next = cur.next;  
 cur.next = next.next;  
 cur = next;  
 }  
 return pCloneHead; //返回克隆头结点  
 }  
}

***//链表 求两个链表的第一个公共结点***public class \_54 {  
 public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {  
 ListNode l1 = pHead1, l2 = pHead2;  
 while (l1 != l2) {  
 l1 = (l1 == null) ? pHead2 : l1.next; //如果l1指向了空，就让l1指向p2，否则指向next  
 l2 = (l2 == null) ? pHead1 : l2.next; //如果l2指向了空，就让l1指向p1，否则指向next  
 }  
 return l1; //最后l1和l2相等，返回任意一个，都是公共节点  
 }  
}

***//链表 两个非空链表表示两个非负整数。每节点存一位数。返回表示和的链表。例如l1=[2,4,3],l2=[5,6,4]返回[7,0,8]***public class \_2 {  
 public ListNode addTwoNumbers(ListNode l1, ListNode l2) {  
 ListNode head = null, tail = null;  
 int carry = 0; //carry是进位  
 while (l1 != null || l2 != null) {  
 //获取两个链表相同位置节点的值，求和，记得加上carry，最开始的carry为0  
 //如果链表元素为null，就把值看做0  
 int n1 = l1 != null ? l1.val : 0;  
 int n2 = l2 != null ? l2.val : 0;  
 int sum = n1 + n2 + carry;  
 if (head == null) { //如果是第一轮计算，同时更新头和尾，值为sum%10  
 head = tail = new ListNode(sum % 10);  
 } else { //否则只需要更新尾节点  
 tail.next = new ListNode(sum % 10);  
 tail = tail.next;  
 }  
 carry = sum / 10; //计算一次和，就要计算进位  
 if (l1 != null) l1 = l1.next; //同时移动l1和l2  
 if (l2 != null) l2 = l2.next;  
 }  
 if (carry > 0) { //如果最后一位carry不为0，要在最后再插入一个节点保存进位  
 tail.next = new ListNode(carry);  
 }//最后返回链表  
 return head;  
 }  
}

***//链表 一个链表数组每个链表都升序。将所有链表合并到一个升序链表，返回合并后的链表***public class \_23 {  
 public ListNode mergeKLists(ListNode[] lists) {  
 return merge(lists, 0, lists.length - 1);  
 }  
 public ListNode merge(ListNode[] lists, int l, int r) {  
 if (l == r) { //如果l=r，直接返回对应链表  
 return lists[l];  
 }  
 if (l > r) { //如果l>r，返回空  
 return null;  
 }  
 int mid = (l + r) >> 1; //否则求l和r的中间  
 //递归合并l到mid和mid+1到r，最后将两个链表合并  
 return mergeTwoLists(merge(lists, l, mid), merge(lists, mid + 1, r));  
 }  
  
 public ListNode mergeTwoLists(ListNode a, ListNode b) {  
 if (a == null || b == null) { //如果a或者b有空的，就返回另一个  
 return a != null ? a : b;  
 }  
 ListNode head = new ListNode(0); //构造合并后的链表头和尾，头是一个虚拟节点  
 ListNode tail = head, aPtr = a, bPtr = b;  
 while (aPtr != null && bPtr != null) {//循环判断两个链表都不为空  
 //如果a的节点<b的节点值，就把a的节点从a中取出放入tail后面，a后移一个节点  
 if (aPtr.val < bPtr.val) {  
 tail.next = aPtr;  
 aPtr = aPtr.next;  
 } else {//否则就把b的节点从b中取出放入tail后面，b后移一个节点  
 tail.next = bPtr;  
 bPtr = bPtr.next;  
 } //更新tail  
 tail = tail.next;  
 }//最后把不为空的剩余链表节点都拼到tail后面  
 tail.next = (aPtr != null ? aPtr : bPtr);  
 return head.next; //返回拼接后的链表，因为头指向虚拟节点，返回头的next  
 }  
}

***//链表 一个链表，两两交换其相邻节点，返回交换后链表的头节点***public class \_24 {  
 public ListNode swapPairs(ListNode head) {  
 ListNode dummyHead = new ListNode(0); //构造一个哑节点  
 dummyHead.next = head; //哑节点的next指向头  
 ListNode temp = dummyHead; //temp表示当前到达节点，初始指向哑节点  
 while (temp.next != null && temp.next.next != null) { //每次更换temp后面2个节点  
 ListNode node1 = temp.next; //取出temp后面依次两个节点node1和node2  
 ListNode node2 = temp.next.next;  
 //核心4步，temp.next = node2，为了断开指向node1的连接，实现node2放到node1前  
 //node1.next = node2.next，让node1指向node2后面的元素  
 //node2.next = node1，此时再让node1成为node2后面的元素就完成了node1和nide2交换  
 //temp = node1，最后更新temp为交换后的第二个节点也就是node1  
 temp.next = node2;  
 node1.next = node2.next;  
 node2.next = node1;  
 temp = node1;  
 }  
 return dummyHead.next; //最后返回哑节点的next  
 }  
}

***//链表 一个链表的头节点，判断链表是否有环***public class \_141 {  
 public boolean hasCycle(ListNode head) {  
 //如果链表为空或者只有一个节点，返回false  
 if (head == null || head.next == null) { return false; }  
 ListNode slow = head; //定义快慢指针，快指针在前  
 ListNode fast = head.next;  
 while (slow != fast) {//循环判断快慢指针是否指向同节点  
 //如果快指针到末尾了，说明没有环，返回false  
 if (fast == null || fast.next == null) {  
 return false;  
 }  
 slow = slow.next; //更新快慢指针  
 fast = fast.next.next;  
 }  
 return true; //如果能退出，说明快慢指针相遇，有环返回true  
 }  
}

***//链表 给链表头节点，每k个节点一组翻转，返回修改后的链表。最后剩余少于k个节点保持原序***public class \_25 {  
 public ListNode reverseKGroup(ListNode head, int k) {  
 ListNode hair = new ListNode(0); //构造虚拟节点  
 hair.next = head; //虚拟节点指向头  
 ListNode pre = hair; //pre指向虚拟节点  
  
 while (head != null) { //遍历链表  
 ListNode tail = pre; //先让tail也指向pre  
 // 如果剩余部分长度小于 k，就不用翻转，直接返回hair的next也就是头  
 for (int i = 0; i < k; ++i) {  
 tail = tail.next;  
 if (tail == null) {  
 return hair.next;  
 }  
 }  
 ListNode nex = tail.next; //保存之前尾节点的next  
 ListNode[] reverse = myReverse(head, tail); //翻转head到tail这部分链表  
 head = reverse[0]; //获取翻转后的头和尾  
 tail = reverse[1];  
 // 把子链表重新接回原链表  
 pre.next = head;  
 tail.next = nex;  
 pre = tail; //更新pre，因为下一次要反转tail.next开始的子链表，所以pre=tail  
 head = tail.next; //头更新为tail.next  
 }  
 return hair.next; //最后返回虚拟头的next  
 }  
 //翻转头到尾的链表  
 public ListNode[] myReverse(ListNode head, ListNode tail) {  
 ListNode prev = tail.next; //tail的next指向prev  
 ListNode p = head; //记录当前节点p  
 while (prev != tail) { //循环判断prev不等于tail  
 ListNode nex = p.next;//取出p的next  
 p.next = prev; //p的next指向prev  
 prev = p;//prev等于p  
 p = nex;//p等于next  
 }  
 return new ListNode[]{tail, head}; //最后返回tail到head  
 }  
}

***//链表 给链表头结点，请按升序排列并返回排序后的链表***public class \_148 {  
 public ListNode sortList(ListNode head) {  
 if (head == null || head.next == null)  
 return head; //如果链表为空或者只有1个节点，直接返回  
 ListNode fast = head.next, slow = head; //快慢指针都指向head  
 //移动快慢指针，让快指针移动到末尾  
 while (fast != null && fast.next != null) {  
 slow = slow.next; fast = fast.next.next;  
 }  
 ListNode tmp = slow.next;//保存slow的下一个节点  
 slow.next = null; //从slow处切断链表  
 ListNode left = sortList(head);//排序左右链表  
 ListNode right = sortList(tmp);  
 ListNode h = new ListNode(0);//构造一个虚拟节点res  
 ListNode res = h;  
 while (left != null && right != null) {//循环左右不能为空  
 if (left.val < right.val) {//如果left小，就让h的next为left  
 h.next = left;  
 left = left.next;//left++  
 } else { //否则让h的next为right  
 h.next = right;  
 right = right.next;//right++  
 }  
 h = h.next; //h++  
 }//某一半为空，剩下不为空的子链表直接加入h的next  
 h.next = left != null ? left : right;  
 return res.next;//然会res的next就是排序后的头节点  
 }  
}

***//链表 删除单链表中某个特定节点***public class \_237 {  
 public void deleteNode(ListNode node) {  
 node.val = node.next.val;//将next的值赋予该节点  
 node.next = node.next.next;//node的next指向next的next，删除next节点  
 }  
}

***//链表 给你单链表头head和两个数left和right，反转从left到right的链表节点，返回反转后的链表***public class \_92 {  
 public ListNode reverseBetween(ListNode head, int left, int right) {  
 // 因为left可能是0即从头开始翻转，因此要建立哑节点  
 ListNode dummyNode = new ListNode(-1); //设置哑节点  
 dummyNode.next = head; //让哑节点next指向头  
 ListNode pre = dummyNode; //找到pre，即left的前一个节点  
 for (int i = 0; i < left - 1; i++) {  
 pre = pre.next;  
 }  
 ListNode cur = pre.next; //cur指向待翻转区第一个节点，cur不断后移  
 ListNode next; //next指向cur下一个节点  
 for (int i = 0; i < right - left; i++) {//需要翻转right-left次  
 next = cur.next; //curr 的下一个节点记录为 next  
 cur.next = next.next; //curr的next指向 next 的next  
 next.next = pre.next;//next 的next指向 pre 的next  
 pre.next = next;//pre 的next指向 next  
 }  
 return dummyNode.next; //返回头节点，也就是哑节点的next  
 }  
}

***//链表 包含n+1个整数的数组nums，数字在[1, n]内，假设只有一个重复整数，返回这个重复数***public class \_287 {  
 public int findDuplicate(int[] nums) {  
 int slow = 0, fast = 0;//定义快慢指针，最初指向0  
 //1.数组中有一个重复整数 <==> 链表中存在环  
 //2.找到数组中重复整数 <==> 找到链表的环入口  
 //至此转换为找到环入口节点。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]]  
 do {  
 slow = nums[slow];  
 fast = nums[nums[fast]];  
 } while (slow != fast);  
 //先让快慢节点相遇  
 slow = 0;//然后慢节点指向头  
 //快慢指针都每次走一步直到相遇就是环入口节点，返回  
 while (slow != fast) {  
 slow = nums[slow];  
 fast = nums[fast];  
 }  
 return slow;  
 }  
}

***//链表 给一个链表头节点，返回链表入环第一个节点。如果无环返回 null***public class \_142 {  
 public ListNode detectCycle(ListNode head) {  
 if (head == null) { //如果链表为空返回空  
 return null;  
 } //定义快慢指针  
 ListNode slow = head, fast = head;  
 //因为可能无环，所以不能用fast!=slow判断  
 while (fast != null) {//遍历判断快指针是否到头  
 slow = slow.next; //更新慢指针  
 //如果快指针下一节点到头，返回null代表无环，否则走两步  
 if (fast.next != null) {  
 fast = fast.next.next;  
 } else {  
 return null;  
 } //如果快慢指针相遇  
 if (fast == slow) {  
 ListNode ptr = head; //使用指针ptr指向头部  
 //ptr和 slow每次向后移动。最终在入环点相遇  
 while (ptr != slow) {  
 ptr = ptr.next;  
 slow = slow.next;  
 }  
 return ptr; //返回相遇点  
 }  
 }  
 return null; //如果快指针到头了，说明无环，返回null  
 }  
}

***//链表 一个单链表头节点head ，判断是否为回文链表***public class \_234 {  
 public boolean isPalindrome(ListNode head) {  
 if (head == null) {//如果链表为空，返回true  
 return true;  
 }  
 // 找到前半部分链表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，p1指向头，p2指向后半部分的头  
 ListNode p1 = head;  
 ListNode p2 = secondHalfStart;  
 boolean result = true;  
 while (result && p2 != null) {//当标志位为true且p2没到终点时  
 if (p1.val != p2.val) {//如果p1和p2值不相等，返回false  
 result = false;  
 }//否则p1和p2依次移动一位  
 p1 = p1.next;  
 p2 = p2.next;  
 }  
 // 翻转后半部分链表，还原链表并返回结果  
 firstHalfEnd.next = reverseList(secondHalfStart);  
 return result;  
 }  
  
 //ok 翻转链表  
 private ListNode reverseList(ListNode head) {  
 ListNode prev = null; //pre=null  
 ListNode curr = head; //cur指向head  
 while (curr != null) {//循环  
 //四部曲，构造temp是cur的next  
 //cur的next是pre，pre=cur,cur=temp  
 ListNode nextTemp = curr.next;  
 curr.next = prev;  
 prev = curr;  
 curr = nextTemp;  
 }  
 return prev; //返回pre  
 }  
  
 //ok 找到链表中间节点  
 private ListNode endOfFirstHalf(ListNode head) {  
 ListNode fast = head;//构造快慢指针，最初都指向head  
 ListNode slow = head;//当快指针的next和next.next都不到末尾时  
 while (fast.next != null && fast.next.next != null) {  
 fast = fast.next.next;  
 slow = slow.next;//移动快慢指针  
 }  
 return slow;//返回慢指针  
 }  
}

# 树

***//根据二叉树前序和中序遍历重建该二叉树，假设前序和中序遍历中不含重复数字***public class \_5 {  
 private Map<Integer, Integer> indexForInOrders = new HashMap<>();// 缓存中序遍历数组值对应索引  
  
 public TreeNode reConstructBinaryTree(int[] pre, int[] in) {  
 //将中序遍历的值和下标放入map  
 for (int i = 0; i < in.length; i++)  
 indexForInOrders.put(in[i], i);  
 return reConstructBinaryTree(pre, 0, pre.length - 1, 0);  
 }  
  
 private TreeNode reConstructBinaryTree(int[] pre, int preL, int preR, int inL) {  
 if (preL > preR) //前序左右边界  
 return null;  
 TreeNode root = new TreeNode(pre[preL]); //前序第一个节点是根节点  
 int inIndex = indexForInOrders.get(root.val); //获取根节点在中序遍历的下标  
 int leftTreeSize = inIndex - inL; //计算左子树大小  
 //左子树的前序左右下标分别是preL+1, preL+leftTreeSize，inL不变  
 root.left = reConstructBinaryTree(pre, preL + 1, preL + leftTreeSize, inL);  
 //右子树的前序左右下标分别是preL+leftTreeSize+1, preR，inL+ leftTreeSize+1  
 root.right = reConstructBinaryTree(pre, preL + leftTreeSize + 1, preR, inL + leftTreeSize + 1);  
 return root;  
 }  
}

***//给一个二叉树一个结点，找出中序遍历下一个结点，结点包含左右子结点和指向父结点指针***public class \_6 {  
 public TreeLinkNode GetNext(TreeLinkNode pNode) {  
 if (pNode.right != null) {//如果右子树不为空  
 //获取右节点，然后不断找这个节点的左节点，直至为null，返回  
 TreeLinkNode node = pNode.right;  
 while (node.left != null)  
 node = node.left;  
 return node;  
 } else {//如果右子树为空  
 while (pNode.next != null) { //不断想上找next不为空的节点  
 TreeLinkNode parent = pNode.next;  
 //如果该节点的left是pNode就返回，否则继续找next  
 if (parent.left == pNode)  
 return parent;  
 pNode = pNode.next;  
 }  
 }  
 return null;  
 }  
}

***//输入两棵二叉树A，B，判断B是不是A的子结构，空树不是任意一个树的子结构***public class \_27 {  
 public boolean HasSubtree(TreeNode root1, TreeNode root2) {  
 if (root1 == null || root2 == null) { //如果有一颗树为空，返回false  
 return false;  
 } //返回root1是否是root2的子结构或者递归调用root1的left和right对root2判断  
 return isSubtree(root1, root2) || HasSubtree(root1.left, root2) || HasSubtree(root1.right, root2);  
 }  
 public boolean isSubtree(TreeNode root1, TreeNode root2) {  
 if (root2 == null) return true; //如果root2是空，返回true  
 if (root1 == null) return false; //如果root1是空，返回false  
 if (root1.val == root2.val) { //如果节点值相等，继续判断root1的left和root2的left，以及right  
 return isSubtree(root1.left, root2.left) && isSubtree(root1.right, root2.right);  
 } else { //如果值不相等，返回false  
 return false;  
 }  
 }  
}

***//操作给定二叉树，变换为源二叉树的镜像***public class \_28 {  
 public void Mirror(TreeNode root) {  
 //如果要镜像的节点为空，直接退出  
 if (root == null) return;  
 //交换根节点的左右子节点，然后继续镜像根的左右节点  
 swap(root);  
 Mirror(root.left);  
 Mirror(root.right);  
 }  
 //交换某个根节点的左右节点  
 private void swap(TreeNode root) {  
 TreeNode t = root.left;  
 root.left = root.right;  
 root.right = t;  
 }  
}

***//树 判断一颗二叉树是不是对称***public class \_29 {  
 boolean isSymmetrical(TreeNode pRoot) {  
 //如果树为空，返回true  
 if (pRoot == null) return true;  
 //否则判断树左右节点是否对称  
 return isSymmetrical(pRoot.left, pRoot.right);  
 }  
 boolean isSymmetrical(TreeNode t1, TreeNode t2) {  
 //如果左右节点都是空，返回true  
 if (t1 == null && t2 == null) return true;  
 //如果只有一个为空，返回false  
 if (t1 == null || t2 == null) return false;  
 //能到这里说明两个都不为空，如果值不相等，返回false  
 if (t1.val != t2.val) return false;  
 //递归判断是否对称，因为对称要求最左侧和最右侧相同，所以判断t1.left和t2.right  
 //&& t1.right和t2.left  
 return isSymmetrical(t1.left, t2.right) && isSymmetrical(t1.right, t2.left);  
 }  
}

***//树 从上往下打印二叉树每个节点，同层从左至右***public class \_33 {  
 public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {  
 Queue<TreeNode> queue = new LinkedList<>(); //存放每层节点  
 ArrayList<Integer> ret = new ArrayList<>(); //存放结果的数组  
 queue.add(root); //先把根节点加入队列  
 while (!queue.isEmpty()) { //循环，当队列不为空时  
 int cnt = queue.size(); //计算队列大小  
 while (cnt-- > 0) { //队列元素每次弹出一个  
 TreeNode t = queue.poll();  
 if (t == null)  
 continue;  
 //如果弹出元素不为空，将值放在结果数组，将左右节点入队列  
 ret.add(t.val);  
 queue.add(t.left);  
 queue.add(t.right);  
 }  
 }  
 return ret;  
 }  
}

***//树 二叉树打印成多行，和上题几乎一样***public class \_34 {  
 ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {  
 ArrayList<ArrayList<Integer>> ret = new ArrayList<>(); //存放结果数组  
 Queue<TreeNode> queue = new LinkedList<>(); //存放每层节点  
 queue.add(pRoot); //根节点加入队列  
 while (!queue.isEmpty()) { //循环判断队列是否不为空  
 ArrayList<Integer> list = new ArrayList<>(); //构造内部数组存放每层元素，为了区分何时换行  
 int cnt = queue.size(); //获取队列大小，即当前层元素数  
 while (cnt-- > 0) { //依次弹出每个元素  
 TreeNode node = queue.poll();  
 if (node == null) continue;  
 //如果不为空，将元素值加入内部数组，左右节点入队  
 list.add(node.val);  
 queue.add(node.left);  
 queue.add(node.right);  
 }  
 //当前层被添加完，将非空内部数组加入ret中  
 if (list.size() != 0) ret.add(list);  
 }  
 return ret;  
 }  
}

***//树 之字形打印二叉树，第一行从左到右，第二层从右至左，第三行从左到右，以此类推。***public class \_35 {  
 public ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {  
 ArrayList<ArrayList<Integer>> ret = new ArrayList<>(); //存放结果数组  
 Queue<TreeNode> queue = new LinkedList<>(); //存放每层节点  
 queue.add(pRoot); //将根节点加入队列  
 boolean reverse = false; //标记是否需要翻转该层  
 while (!queue.isEmpty()) {  
 ArrayList<Integer> list = new ArrayList<>();//构造内部数组存放每层元素，为了区分何时换行  
 int cnt = queue.size();//获取队列大小，即当前层元素数  
 while (cnt-- > 0) {//依次弹出每个元素  
 TreeNode node = queue.poll();  
 if (node == null)  
 continue;  
 //如果不为空，将元素值加入内部数组，左右节点入队  
 list.add(node.val);  
 queue.add(node.left);  
 queue.add(node.right);  
 }  
 //到这里是打印完了一层，如果是翻转层，就将list翻转，如果不允许使用Collections  
 //也可以在上面的while增加判断，如果是reverse，就queue.add(right) queue.add(left)  
 if (reverse) //如果需要翻转就翻转内部数组  
 Collections.reverse(list);  
 reverse = !reverse; //更新翻转标记  
 //当前层被添加完，将非空内部数组加入ret中  
 if (list.size() != 0)  
 ret.add(list);  
 }  
 return ret;  
 }  
}

***//树 输入一个整数数组，判断数组是不是某二叉搜索树后序遍历结果***public class \_36 {  
 public boolean VerifySquenceOfBST(int[] sequence) {  
 if (sequence == null || sequence.length == 0) return false; //如果序列是空返回false  
 return verify(sequence, 0, sequence.length - 1);  
 }  
  
 private boolean verify(int[] sequence, int first, int last) {  
 //如果只有一个节点，一定是真  
 if (last - first <= 1) return true;  
 //后序遍历最后一个节点一定是根  
 int rootVal = sequence[last];  
 //cutIndex是左右子树分界点，二叉搜索树左子树一定比根值小，右子树一定比根的值大  
 int cutIndex = first;  
 while (cutIndex < last && sequence[cutIndex] <= rootVal)  
 cutIndex++;  
 //找到右子树第一个节点，判断右子树如果有一个值比根小，就不是二叉搜索树，返回false  
 for (int i = cutIndex; i < last; i++)  
 if (sequence[i] < rootVal)  
 return false;  
 //如果从根的左右子树满足规则，继续找根节点的下一层的根节点，也就是根节点的左右子节点  
 //因为后序遍历是左右根，所以左子树的根一定是cutIndex-1节点，右子树的根是last-1的节点  
 //递归判断这两个根节点是否满足条件，要求同时满足，因此用&&  
 return verify(sequence, first, cutIndex - 1) && verify(sequence, cutIndex, last - 1);  
 }  
}

***//树 一颗二叉树和整数，打印二叉树中结点值和为输入整数的所有路径。路径定义为从根结点开始一直到叶结点经过的结点***public class \_37 {  
 private static ArrayList<ArrayList<Integer>> ret = new ArrayList<>(); //存放结果数组  
 public static ArrayList<ArrayList<Integer>> FindPath(TreeNode root, int target) {  
 backtracking(root, target, new ArrayList<Integer>());  
 return ret;  
 }  
  
 private static void backtracking(TreeNode node, int target, ArrayList<Integer> path) {  
 //如果二叉树为空，直接返回空  
 if (node == null) return;  
 //将根节点值加入路径，目标值减掉根节点值  
 path.add(node.val);  
 target -= node.val;  
 //如果目标值为0并且当前节点是叶子节点，说明找到了一条路径，将path加入ret中  
 if (target == 0 && node.left == null && node.right == null) {  
 ret.add(new ArrayList<>(path));  
 } else { //否则回溯left和right  
 backtracking(node.left, target, path);  
 backtracking(node.right, target, path);  
 }  
 //只有当上面的else分支两个方法都失败才能到这里，  
 //说明当时的node的左右子树都已经没法找到所需的路径，需要回退  
 //因此需要把node的值从path中移除，向node的上一层另一条分支找  
 path.remove(path.size() - 1);  
 }  
}

***//树 一棵二叉搜索树，将树转换成排序的双向链表***public class \_39 {  
 private TreeNode pre = null;  
 private TreeNode head = null;  
  
 public TreeNode Convert(TreeNode root) {  
 inOrder(root);  
 return head;  
 }  
  
 private void inOrder(TreeNode node) {  
 if (node == null) return; //如树为空，直接返回  
 inOrder(node.left); //否则递归左节点  
 node.left = pre; //节点的左节点比它小，放在pre  
 if (pre != null) //将node设置为pre的右节点  
 pre.right = node;  
 pre = node; //更新pre指向node  
 if (head == null) //初始化头结点  
 head = node;  
 inOrder(node.right); //递归右节点  
 }  
}

***//树 输入一棵二叉树，判断该树是否是平衡二叉树，左右子树高度差不超1***public class \_58 {  
 boolean isBalance = true; //标记是否是平衡树  
 public boolean IsBalanced\_Solution(TreeNode root) {  
 lengthOfTree(root);  
 return isBalance;  
 }  
 private int lengthOfTree(TreeNode root) {  
 //如果树为空，返回高度0  
 if (root == null) return 0;  
 //求左子树和右子树高度  
 int left = lengthOfTree(root.left);  
 int right = lengthOfTree(root.right);  
 //如果高度差超过1，设置false  
 if (Math.abs(left - right) > 1) isBalance = false;  
 //最后返回左右子树的高度最大值+1  
 return Math.max(left, right) + 1;  
 }  
}

***//树 两个函数分别序列化和反序列化二叉树***public class \_40 {  
 private String deserializeStr;  
  
 public String Serialize(TreeNode root) {  
 //如果是空，直接返回#  
 if (root == null) return "#";  
 //否则返回值用空格分割，按照根左右顺序  
 return root.val + " " + Serialize(root.left) + " " + Serialize(root.right);  
 }  
  
 public TreeNode Deserialize(String str) {  
 //deserializeStr用于迭代不断变化的剩余需要反序列化的字符串  
 deserializeStr = str;  
 return Deserialize();  
 }  
  
 private TreeNode Deserialize() {  
 //如果需要反序列化的字符串长度为0，说明反序列化完毕，返回  
 if (deserializeStr.length() == 0) return null;  
 //找到节点之间的分隔符位置  
 int index = deserializeStr.indexOf(" ");  
 //从deserializeStr中拿出node值，如果index是-1就代表整个deserializeStr是节点值  
 String node = index == -1 ? deserializeStr : deserializeStr.substring(0, index);  
 //更新取出当前node之后的deserializeStr  
 deserializeStr = index == -1 ? "" : deserializeStr.substring(index + 1);  
 //如果node是#说明是空树，返回空  
 if (node.equals("#")) return null;  
 //转换node为整形值，构造解析出来的值的节点  
 int val = Integer.valueOf(node);  
 TreeNode t = new TreeNode(val);  
 //继续反序列化左右节点  
 t.left = Deserialize();  
 t.right = Deserialize();  
 return t;  
 }  
}

***//树 返回二叉查找树第 K 个结点***public class \_56 {  
 private TreeNode ret;  
 private int cnt = 0;  
  
 public TreeNode KthNode(TreeNode pRoot, int k) {  
 inOrder(pRoot, k); //中序遍历  
 return ret;  
 }  
  
 private void inOrder(TreeNode root, int k) {  
 if (root == null || cnt >= k) return; //如果根为空或者cnt到了k就返回  
 //左根右顺序遍历，每遍历一个节点，就cnt++  
 inOrder(root.left, k);  
 cnt++;  
 if (cnt == k) //找到第k个节点，就把该节点赋值给ret  
 ret = root;  
 inOrder(root.right, k);  
 }  
}

***//树 求二叉树深度，从根到叶依次经过的结点（含根、叶）形成一条路径，最长路径长度为树深度***public class \_57 {  
 public int TreeDepth(TreeNode root) {  
 //如果树为空，返回0，否则就是左右子树深度最大值+1  
 return root == null ? 0 : 1 + Math.max(TreeDepth(root.left), TreeDepth(root.right));  
 }  
}

***//树 求树中两个节点的最低公共祖先***public class \_73 {  
 //针对二叉查找树  
 public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
 if (root == null) return root; //如果树位空，返回该节点  
 //如果节点的值比p和q都大，就找该节点left，如果都小就找right，否则返回该节点  
 if (root.val > p.val && root.val > q.val)  
 return lowestCommonAncestor(root.left, p, q);  
 if (root.val < p.val && root.val < q.val)  
 return lowestCommonAncestor(root.right, p, q);  
 return root;  
 }  
 //针对普通二叉树  
 public TreeNode lowestCommonAncestor2(TreeNode root, TreeNode p, TreeNode q) {  
 //如果节点为p或者q或空，就返回该节点  
 if (root == null || root == p || root == q) return root;  
 //否则继续找该节点的左和右  
 TreeNode left = lowestCommonAncestor2(root.left, p, q);  
 TreeNode right = lowestCommonAncestor2(root.right, p, q);  
 //如果left是空就返回right，right为空就返回left，否则返回root  
 return left == null ? right : right == null ? left : root;  
 }  
}

***//树 给一个二叉树根节点 ，返回它的中序遍历***public class \_94 {  
 public List<Integer> inorderTraversal(TreeNode root) {  
 List<Integer> res = new ArrayList<Integer>();  
 inorder(root, res);  
 return res;  
 }  
  
 public void inorder(TreeNode root, List<Integer> res) {  
 if (root == null) return; //如果树为空返回  
 inorder(root.left, res); //递归左节点  
 res.add(root.val); //值放入res  
 inorder(root.right, res); //递归右节点  
 }  
}

***//树 给一个整数 n ，生成所有n个节点组成且节点值从1到n的不同二叉搜索树***public class \_95 {  
 public List<TreeNode> generateTrees(int n) {  
 if (n == 0) return new LinkedList<TreeNode>();//如果n=0，返回空链表  
 return generateTrees(1, n);  
 }  
 public List<TreeNode> generateTrees(int start, int end) {  
 List<TreeNode> allTrees = new LinkedList<TreeNode>(); //保存所有结果  
 if (start > end) { //如果start超过end，结果加入一个null返回  
 allTrees.add(null);  
 return allTrees;  
 }  
 // 枚举可行根节点，i从start到end  
 for (int i = start; i <= end; i++) {  
 // 获得所有可行的左子树集合  
 List<TreeNode> leftTrees = generateTrees(start, i - 1);  
 // 获得所有可行的右子树集合  
 List<TreeNode> rightTrees = generateTrees(i + 1, end);  
 // 从左子树集合中选出一棵左子树，从右子树集合中选出一棵右子树，拼接到根节点上  
 for (TreeNode left : leftTrees) {  
 for (TreeNode right : rightTrees) {  
 //遍历左右子树，构造节点，节点的left是左子树节点，right是右子树节点  
 TreeNode currTree = new TreeNode(i);  
 currTree.left = left;  
 currTree.right = right;  
 allTrees.add(currTree); //将当前节点加入结果集  
 }  
 }  
 }  
 return allTrees; //返回结果集  
 }  
}

***//树 给一个二叉树根节点，判断其是否是有效的二叉搜索树***public class \_98 {  
 long pre = Long.MIN\_VALUE;  
  
 public boolean isValidBST(TreeNode root) {  
 if (root == null) { //如果树为空，返回true  
 return true;  
 }//递归判断左子树  
 if (!isValidBST(root.left)) {  
 return false;  
 } //如果当前节点小于等于中序遍历的前一节点，返回false  
 if (root.val <= pre) {  
 return false;  
 }//更新前一个节点  
 pre = root.val;  
 //递归判断右子树  
 return isValidBST(root.right);  
 }  
}

***//树 给二叉树根节点，返回层序遍历***public class \_102 {  
 public List<List<Integer>> levelOrder(TreeNode root) {  
 if (root == null) { //如果树为空，返回空数组  
 return new ArrayList<List<Integer>>();  
 }  
 List<List<Integer>> res = new ArrayList<List<Integer>>(); //存放结果  
 LinkedList<TreeNode> queue = new LinkedList<TreeNode>();  
 queue.add(root); //根节点放入队列，然后不断遍历队列  
 while (queue.size() > 0) {  
 //获取当前队列长度，这相当于这一层的节点数  
 int size = queue.size();  
 ArrayList<Integer> tmp = new ArrayList<Integer>();  
 //将队列中的元素都拿出来(也就是获取这一层的节点)，放到临时list中  
 //如果节点的左/右子树不为空，放入队列中  
 for (int i = 0; i < size; ++i) {  
 TreeNode t = queue.remove();  
 tmp.add(t.val);  
 if (t.left != null) {  
 queue.add(t.left);  
 }  
 if (t.right != null) {  
 queue.add(t.right);  
 }  
 }  
 res.add(tmp);//将临时list(每层)加入结果  
 }  
 return res; //返回结果  
 }  
}

***//树 二叉树按先序遍历顺序展为单链表：单链表同样用TreeNode，right指向链表下一结点，左子指针为null。***public class \_114 {  
 //左子树插入到右子树的地方，将原来的右子树接到左子树最右边节点  
 public void flatten(TreeNode root) {  
 while (root != null) {  
 //左子树为 null，直接考虑下一个节点  
 if (root.left == null) {  
 root = root.right;  
 } else {  
 // 找左子树最右边的节点pre  
 TreeNode pre = root.left;  
 while (pre.right != null) {  
 pre = pre.right;  
 }  
 //将原来的右子树接到左子树的最右边节点pre  
 pre.right = root.right;  
 // 将左子树插入到右子树的地方  
 root.right = root.left;  
 root.left = null;  
 // 考虑下一个节点  
 root = root.right;  
 }  
 }  
 }  
}

***//树 二叉树找出最小深度。深度从根节点到最近叶节点的最短路径上的节点数***public class \_111 {  
 public int minDepth(TreeNode root) {  
 if (root == null) { //如果树为空，返回0  
 return 0;  
 }//如果左或右子树为空，返回1  
 if (root.left == null && root.right == null) {  
 return 1;  
 }  
 int min\_depth = Integer.MAX\_VALUE; //初始值最小深度为最大整数  
 if (root.left != null) { //递归求左子树深度并更新最小深度  
 min\_depth = Math.min(minDepth(root.left), min\_depth);  
 } //递归求右子树深度并更新最小深度  
 if (root.right != null) {  
 min\_depth = Math.min(minDepth(root.right), min\_depth);  
 }  
 return min\_depth + 1; //返回最小深度+1，因为根节点深度没统计  
 }  
}

***//树 路径为从树中任意节点出发沿父-子连接到任意节点的序列。路径至少含一节点不一定过根。返回二叉树最大路径和***public class \_124 {  
 int maxSum = Integer.MIN\_VALUE;  
  
 public int maxPathSum(TreeNode root) {  
 maxGain(root);  
 return maxSum;  
 }  
  
 public int maxGain(TreeNode node) {  
 if (node == null) { //如果树为空，返回0  
 return 0;  
 }  
 // 递归计算左右子节点的最大贡献值，只有在贡献值大于 0 时才选取  
 int leftGain = Math.max(maxGain(node.left), 0);  
 int rightGain = Math.max(maxGain(node.right), 0);  
 // 节点的最大路径和等于该节点值与左右子节点的最大贡献值之和  
 int priceNewpath = node.val + leftGain + rightGain;  
 // 更新答案  
 maxSum = Math.max(maxSum, priceNewpath);  
 //返回节点最大贡献值，是节点值+左右中的较大值，因为只能取左右中一条  
 return node.val + Math.max(leftGain, rightGain);  
 }  
}

***//树 返回二叉树前序遍历***public class \_144 {  
 public List<Integer> preorderTraversal(TreeNode root) {  
 List<Integer> res = new ArrayList<Integer>(); //存放结果  
 preorder(root, res);  
 return res;  
 }  
  
 public void preorder(TreeNode root, List<Integer> res) {  
 if (root == null) { //如果节点为空，返回  
 return;  
 }//根左右  
 res.add(root.val); //节点值放入res  
 preorder(root.left, res);//递归左右节点  
 preorder(root.right, res);  
 }  
}

***//树 二叉树的后序遍历***public class \_145 {  
 public List<Integer> postorderTraversal(TreeNode root) {  
 List<Integer> res = new ArrayList<Integer>(); //存放结果  
 postorder(root, res);  
 return res;  
 }  
  
 public void postorder(TreeNode root, List<Integer> res) {  
 if (root == null) {//如果节点为空返回  
 return;  
 }//左右根  
 postorder(root.left, res);//递归左右  
 postorder(root.right, res);  
 res.add(root.val);//节点放入res  
 }  
}

***//树 实现Trie类：Trie()初始化前缀树。void insert(String word)向树插入word。boolean search(String word)如果word在树中返回true***//boolean startsWith(String prefix) 如果已经插入的word前缀之一为prefix返回true  
public class \_208 {  
 //Trie节点包含：指向子节点指针数组children。本题数组长26即英文字母数。  
 // children[0]对应a，children[25]对应z。  
 // 布尔字段 isEnd，表示该节点是否为字符串结尾  
 class Trie {  
 private Trie[] children;  
 private boolean isEnd;  
  
 public Trie() { //构造方法初始化children和isEnd  
 children = new Trie[26];  
 isEnd = false;  
 }  
  
 public void insert(String word) {  
 Trie node = this; //构造节点  
 for (int i = 0; i < word.length(); i++) {  
 char ch = word.charAt(i); //遍历单词每个字符  
 int index = ch - 'a'; //计算该字符的index  
 //如果node的children在index处没节点，就构造一个  
 if (node.children[index] == null) {  
 node.children[index] = new Trie();  
 }//否则将node指向children在index节点，继续向下找  
 node = node.children[index];  
 }  
 node.isEnd = true; //插入完毕将最后一个节点标记为end  
 }  
 //搜索前缀之后判断当前node是否不为空且是end  
 public boolean search(String word) {  
 Trie node = searchPrefix(word);  
 return node != null && node.isEnd;  
 }  
  
 //搜索前缀，判断是否能搜到最后一个节点  
 public boolean startsWith(String prefix) {  
 return searchPrefix(prefix) != null;  
 }  
  
 //根据前缀搜索节点  
 private Trie searchPrefix(String prefix) {  
 Trie node = this; //构造头节点  
 for (int i = 0; i < prefix.length(); i++) {  
 char ch = prefix.charAt(i); //遍历前缀每个字符  
 int index = ch - 'a'; //计算该字符的index  
 //如果node的children在index处没节点，返回空  
 if (node.children[index] == null) {  
 return null;  
 }//否则继续找下一个节点  
 node = node.children[index];  
 }  
 return node;//查找完毕返回前缀最后一个字符对应节点  
 }  
 }  
}

***//树 非空节点记录值。空节点用#。逗号分隔的序列，验证是否是正确的二叉树前序序列化***public class \_331 {  
 public boolean isValidSerialization(String preorder) {  
 int n = preorder.length(); //获取字符串长度  
 int i = 0;  
 int slots = 1;//遇到空节点消耗一个槽；非空节点消耗一个槽再补充两个槽  
 while (i < n) {//遍历字符串序列  
 if (slots == 0) return false;//如果槽位数是0，返回false  
 if (preorder.charAt(i) == ',') {//分隔符直接跳过  
 i++;  
 } else if (preorder.charAt(i) == '#') {//如果是空位，slot++  
 slots--;  
 i++;  
 } else {//否则是数字，读出到下一个,的所有数字  
 while (i < n && preorder.charAt(i) != ',') { i++; }  
 slots++; // slots = slots - 1 + 2  
 }  
 }  
 return slots == 0;//遍历完判断槽是否位0  
 }  
}

***//树 定一个N叉树，返回层序遍历***public class \_429 {  
 public List<List<Integer>> levelOrder(Node root) {  
 if (root == null) {//如果树为空，返回空数组  
 return new ArrayList<List<Integer>>();  
 }//构造答案数组，是双层整数数组  
 List<List<Integer>> ans = new ArrayList<List<Integer>>();  
 Queue<Node> queue = new ArrayDeque<Node>();//构造队列  
 queue.offer(root);//根节点入队  
 while (!queue.isEmpty()) {//循环判断队列不为空  
 int cnt = queue.size();//计算队列大小，就是某一层的节点数  
 List<Integer> level = new ArrayList<Integer>();//构造存放一层节点的数组  
 for (int i = 0; i < cnt; ++i) {//遍历层内cnt个节点  
 Node cur = queue.poll();//出队一个节点，值加入level  
 level.add(cur.val);//还要将出队节点所有子节点入队  
 for (Node child : cur.children) {  
 queue.offer(child);  
 }  
 }//遍历完一层，将这层的level加入ans  
 ans.add(level);  
 }  
 return ans;//返回ans  
 }  
}

***//树 给二叉树和整数targetSum，求二叉树里节点值和等于targetSum的路径数。不需根开始也不需叶结束，路径必须向下***public class \_437 {  
 public int pathSum(TreeNode root, int targetSum) {  
 if (root == null) {//如果树为空，返回0  
 return 0;  
 }  
 int ret = rootSum(root, targetSum);//从根节点开始找看有多少路径  
 ret += pathSum(root.left, targetSum);//再加上从左右节点找到的路径  
 ret += pathSum(root.right, targetSum);  
 return ret;//最后返回总路径  
 }  
 //从root开始找target一共有多少路径  
 public int rootSum(TreeNode root, int targetSum) {  
 int ret = 0;//记录结果，初始值为0  
 if (root == null) {//如果节点为空，返回0  
 return 0;  
 }  
 int val = root.val;//获取节点值，如果值为target，ret++  
 if (val == targetSum) {  
 ret++;  
 }//遍历左右子树，寻找target-val  
 ret += rootSum(root.left, targetSum - val);  
 ret += rootSum(root.right, targetSum - val);  
 return ret;//返回路径数  
 }  
}

***//树 给一棵二叉树，找出二叉树每一层最大值***public class \_515 {  
 public List<Integer> largestValues(TreeNode root) {  
 if (root == null) return new ArrayList<Integer>();//如果树为空，返回空列表  
 //构造结果数组  
 List<Integer> res = new ArrayList<Integer>();  
 //因为要层序遍历，需要构造队列装每层的节点元素  
 Queue<TreeNode> queue = new ArrayDeque<TreeNode>();  
 queue.offer(root);//将根节点加入队列  
 while (!queue.isEmpty()) {//循环判断队列不为空  
 int len = queue.size();//获取队列元素数  
 int maxVal = Integer.MIN\_VALUE;//将最大值初始化为MIN  
 while (len > 0) {//循环找队列中最大元素  
 len--;//元素数-1  
 TreeNode t = queue.poll();//弹出队首节点  
 maxVal = Math.max(maxVal, t.val);//更新最大值  
 if (t.left != null) {//弹出一个元素就把其左右子节点入队  
 queue.offer(t.left);  
 }  
 if (t.right != null) {  
 queue.offer(t.right);  
 }  
 }  
 res.add(maxVal);//将最后的最大值加入res  
 }  
 return res;//返回res  
 }  
}

***//树 给二叉搜索树，节点值各不相同，转换为累加树使每个节点新值等于原树中大于等于node.val的值之和***public class \_538 {  
 int sum = 0;//记录节点值的和  
 //反序的中序遍历  
 public TreeNode convertBST(TreeNode root) {  
 if (root != null) {//  
 //右根左  
 convertBST(root.right);//遍历右节点  
 sum += root.val;//值加入sum  
 root.val = sum;//更新节点值为sum  
 convertBST(root.left);//遍历左节点  
 }  
 return root;  
 }  
}

***//树 计算二叉树的直径。直径是任意两结点路径长度中的最大值。路径可能也可能不穿过根***public class \_543 {  
 int ans;//记录答案  
 public int diameterOfBinaryTree(TreeNode root) {  
 ans = 1;//初始化为1  
 depth(root);//求二叉树深度  
 return ans - 1;//返回ans-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); // 递归求左子树深度L  
 int R = depth(node.right); // 递归求右子树深度R  
 ans = Math.max(ans, L + R + 1); // L+R再+1和ans比较谁更大，更新ans  
 return Math.max(L, R) + 1; // 返回L和R较大值再+1  
 }  
}

***//树 n叉树返回其节点值前序遍历***public class \_589 {  
 public List<Integer> preorder(Node root) {  
 List<Integer> res = new ArrayList<>();//构造结果数组  
 helper(root, res);  
 return res;  
 }  
  
 public void helper(Node root, List<Integer> res) {  
 if (root == null) {//如果根为空直接返回  
 return;  
 }  
 res.add(root.val);//否则将根节点加入res  
 for (Node ch : root.children) {//递归所有子节点  
 helper(ch, res);  
 }  
 }  
}

***//树 给两棵二叉树。一棵覆盖到另一棵，如果两节点重叠，将节点值相加作新值；否则不为null的节点作新节点***public class \_617 {  
 public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {  
 if (t1 == null) {//如果树1为空，返回树2  
 return t2;  
 }  
 if (t2 == null) {//如果树2为空，返回树1  
 return t1;  
 }//构造合并后的根节点，值是原两树根节点和  
 TreeNode merged = new TreeNode(t1.val + t2.val);  
 //递归合并left和right  
 merged.left = mergeTrees(t1.left, t2.left);  
 merged.right = mergeTrees(t1.right, t2.right);  
 return merged;//返回合并后的树  
 }  
}

# 栈

***//两个栈实现队列,完成 Push 和 Pop***public class \_7 {  
 Stack<Integer> in = new Stack<Integer>(); //入栈  
 Stack<Integer> out = new Stack<Integer>(); //出栈  
 //push方法入栈直接push即可  
 public void push(int node) {  
 in.push(node);  
 }  
 //弹出方法：不断将in的元素依次放入out栈，最后弹出out栈顶元素  
 public int pop() throws Exception {  
 if (out.isEmpty())  
 while (!in.isEmpty())  
 out.push(in.pop());  
 if (out.isEmpty())  
 throw new Exception("queue is empty");  
 return out.pop();  
 }  
}

***//栈 定义栈数据结构，实现能得到栈最小元素的 min 函数***public class \_31 {  
 private Stack<Integer> dataStack = new Stack<>(); //存放原数据序列的栈  
 private Stack<Integer> minStack = new Stack<>(); //保证栈顶存放当前栈内最小的元素  
 //关键是入栈，数据栈直接入栈，最小栈需要比较待入栈元素和当前栈顶元素谁更小，  
 // 放入小值，保证最小栈的栈顶元素一定是当前数据栈内容的最小值，最小栈里至多一个元素  
 public void push(int node) {  
 dataStack.push(node);  
 minStack.push(minStack.isEmpty() ? node : Math.min(minStack.peek(), node));  
 }  
 //弹栈一起弹出  
 public void pop() {  
 dataStack.pop(); minStack.pop();  
 }  
 //ok 查看最顶元素就调用数据栈的peek  
 public int top() {  
 return dataStack.peek();  
 }  
 //ok获取最小元素就调用最小栈的peek  
 public int min() { return minStack.peek(); }  
}

***//栈 字符串表达式s，实现计算器来计算它的值，只包含+-或括号***public class \_224 {  
 public int calculate(String s) {  
 Deque<Integer> ops = new LinkedList<Integer>();//构造栈  
 ops.push(1);//顶元素为初始值+1  
 int sign = 1;//记录符号位  
 int ret = 0; //保存结果  
 int n = s.length(); //计算字符串长度  
 int i = 0;  
 while (i < n) { //i从0到n  
 if (s.charAt(i) == ' ') {//如果字符是空格，跳过  
 i++;  
 } else if (s.charAt(i) == '+') {//如果是+  
 sign = ops.peek(); //符号=栈顶元素  
 i++;  
 } else if (s.charAt(i) == '-') {//如果是-  
 sign = -ops.peek(); //符号=负栈顶元素  
 i++;  
 } else if (s.charAt(i) == '(') {//如果是(  
 ops.push(sign);//将符号入栈  
 i++;  
 } else if (s.charAt(i) == ')') {//如果是)  
 ops.pop();//出栈顶元素  
 i++;  
 } else { //否则是数字  
 long num = 0; //找出所有连续数字计算其值为num  
 while (i < n && Character.isDigit(s.charAt(i))) {  
 num = num \* 10 + s.charAt(i) - '0';  
 i++;  
 }  
 ret += sign \* num; //ret+=符号\*num  
 }  
 }  
 return ret;//返回ret  
 }  
}

***//栈 输入两个整数序列，第一个序列表示栈压入顺序，判断第二序列是否为该栈弹出顺序。假设入栈所有数字不相等。***public class \_32 {  
 public static boolean IsPopOrder(int[] pushSequence, int[] popSequence) {  
 //计算入栈元素数  
 int n = pushSequence.length;  
 //构造一个辅助栈，按照入栈序列把元素放入辅助栈，然后去出栈序列中找这个元素  
 Stack<Integer> stack = new Stack<>();  
 for (int pushIndex = 0, popIndex = 0; pushIndex < n; pushIndex++) {  
 //按照入栈顺序把元素依次放入stack  
 stack.push(pushSequence[pushIndex]);  
 //每次入栈一个元素，就看当前stack栈顶是否是弹栈序列中待找元素，如果是就弹出，弹栈下标++继续找  
 //直到栈顶元素和待弹栈元素不一致，就下一轮循环继续把元素入栈  
 while (popIndex < n && !stack.isEmpty() && stack.peek() == popSequence[popIndex]) {  
 stack.pop();  
 popIndex++;  
 }  
 }  
 return stack.isEmpty();  
 }  
}

***//栈 返回字符流中第一个不重复字符，例如当读出"go" 时，第一个不重复字符是 "g"，当读出“google" 时字符是 "l"***public class \_45 {  
 private int[] cnts = new int[256]; //一共256字符，用cnts代表各字符出现次数  
 private Queue<Character> queue = new LinkedList<>();  
 public void Insert(char ch) {  
 //插入一个字符，先把字符的asc码位置的cnts+1，然后将字符放入队列  
 cnts[ch]++;  
 queue.add(ch);  
 //如果队列不为空并且当前字符已经出现超过1次，就弹出当前字符  
 //保证queue中只保留出现一次的字符，且后插入的放在队尾  
 while (!queue.isEmpty() && cnts[queue.peek()] > 1)  
 queue.poll();  
 }  
 public char FirstAppearingOnce() {  
 //弹出队首元素就是第一个只出现一次的字符  
 return queue.isEmpty() ? '#' : queue.peek();  
 }  
}

***//栈 给一个只包括 '('，')'，'{'，'}'，'['，']' 字符串s ，判断括号是否有效***public class \_20 {  
 private static final Map<Character, Character> map = new HashMap<Character, Character>() {{  
 put('{', '}');put('[', ']');put('(', ')');put('?', '?');  
 }}; //保存各个括号对应关系  
 public boolean isValid(String s) {  
 //如果字符串不为空但第0个元素不是左括号，直接返回false  
 if (s.length() > 0 && !map.containsKey(s.charAt(0))) return false;  
 //构建一个栈，放入一个"?"  
 LinkedList<Character> stack = new LinkedList<Character>() {{ add('?'); }};  
 for (Character c : s.toCharArray()) { //遍历字符串每个字符  
 //如果是左括号，就入栈  
 if (map.containsKey(c)) stack.addLast(c);  
 //如果不是左括号，就移除栈顶的左括号，再map中找到对应的右括号，如果c不是对应右括号，返回false  
 else if (map.get(stack.removeLast()) != c) return false;  
 }  
 return stack.size() == 1; //如果遍历完了，栈大小为1，即只剩下一个?，就返回true  
 }  
}

***//栈 一个只包含 '(' 和 ')' 字符串，找出最长有效（格式正确且连续）括号子串长度***public class \_32 {  
 public int longestValidParentheses(String s) {  
 int maxans = 0; //保存结果  
 Deque<Integer> stack = new LinkedList<Integer>(); //构造一个辅助栈  
 stack.push(-1); //栈先放入-1，所以正常匹配栈不应该为空  
 for (int i = 0; i < s.length(); i++) {//遍历字符串  
 if (s.charAt(i) == '(') { //如果字符串当前字符是左括号  
 stack.push(i); //将字符下标入栈  
 } else { //如果是右括号  
 stack.pop(); //弹栈  
 if (stack.isEmpty()) { //弹栈后如果栈为空，将下标再入栈  
 stack.push(i);  
 } else { //如果栈不为空，比较i-栈顶元素和当前最大值，更新最大值  
 maxans = Math.max(maxans, i - stack.peek());  
 }  
 }  
 }  
 return maxans; //最后返回最大值  
 }  
}

***//栈 一个循环数组nums，返回每个元素下一个更大元素。应该循环搜索下一个更大数。如不存在输出-1***public class \_503 {  
 public int[] nextGreaterElements(int[] nums) {  
 int n = nums.length;//计算数组长度n  
 int[] ret = new int[n]; //构造结果数组  
 Arrays.fill(ret, -1);//先把结果数组都填成-1  
 Deque<Integer> stack = new LinkedList<Integer>();//构造栈  
 //把循环数组「拉直」，即复制前 n−1 个元素拼在原序列后面，相当于循环2轮，如果还没找到最大数那就是没有  
 for (int i = 0; i < n \* 2 - 1; i++) {  
 //到位置i，就将单调栈中所有小于nums[i]下标弹出栈，下一个更大元素即为nums[i]。随后将位置i入栈  
 //循环判断，如果栈不为空，且栈顶元素(下标)对应数小于num[i%n]，则弹出栈顶元素  
 //直到栈顶元素(下标)对应数大于等于num[i%n]，将nums[i % n]赋值给ret的弹出下标  
 while (!stack.isEmpty() && nums[stack.peek()] < nums[i % n]) {  
 ret[stack.pop()] = nums[i % n];  
 }  
 stack.push(i % n);//最后将下标i入栈  
 }  
 return ret;//返回结果  
 }  
}

***//栈 返回解码后字符串。编码规则:k[str]表示str重复k次。字符串没有空格，原始数据不含数字，数字只表示重复次数k***public class \_394 {  
 int ptr;//全局维护指针  
 public String decodeString(String s) {  
 LinkedList<String> stk = new LinkedList<String>();//构造栈维护token  
 ptr = 0;//指针初始指向字符串头  
 while (ptr < s.length()) {//遍历传入字符串  
 char cur = s.charAt(ptr);//获取字符  
 if (Character.isDigit(cur)) {//如果字符是数字，将连续数字都取出  
 String digits = getDigits(s);  
 stk.addLast(digits);//加入最后  
 } else if (Character.isLetter(cur) || cur == '[') {  
 //如果是字母或[，加入最后  
 stk.addLast(String.valueOf(s.charAt(ptr++)));  
 } else {//否则一定是]  
 ++ptr;//跳过右括号  
 LinkedList<String> sub = new LinkedList<String>();  
 while (!"[".equals(stk.peekLast())) {//将栈内元素非[全部出栈加入sub  
 sub.addLast(stk.removeLast());  
 }//出栈序列翻转  
 Collections.reverse(sub);  
 stk.removeLast();// 左括号出栈  
 // 此时栈顶为当前sub对应字符串应出现次数，出栈并解析为整数  
 int repTime = Integer.parseInt(stk.removeLast());  
 StringBuffer t = new StringBuffer();  
 String o = getString(sub);//将sub构造成字符串  
 while (repTime-- > 0) {//重复构造repTime次sub  
 t.append(o);//形成结果t  
 }  
 stk.addLast(t.toString());//最后将构造好的字符串t入栈  
 }  
 }  
 return getString(stk);//拼接链表内所有字符串返回  
 }  
 //从s中提取数字  
 public String getDigits(String s) {  
 StringBuffer ret = new StringBuffer();  
 while (Character.isDigit(s.charAt(ptr))) {  
 //从ptr开始取出所有连续数字加入buffer返回  
 ret.append(s.charAt(ptr++));  
 }  
 return ret.toString();  
 }  
 //拼接链表内所有字符串  
 public String getString(LinkedList<String> v) {  
 StringBuffer ret = new StringBuffer();  
 for (String s : v) {  
 ret.append(s);  
 }  
 return ret.toString();  
 }  
}

***//栈 字符串表达式s，实现计算器来计算它的值。除法仅保留整数，仅包含+-\*/和数字***public class \_227 {  
 public int calculate(String s) {  
 Deque<Integer> stack = new ArrayDeque<Integer>();//构造栈  
 char preSign = '+'; //每个数字前的运算符  
 int num = 0;  
 int n = s.length();//计算字符串长度  
 for (int i = 0; i < n; ++i) {//遍历字符串  
 if (Character.isDigit(s.charAt(i))) {//如果字符是数字，将连续数字全取出并计算num  
 num = num \* 10 + s.charAt(i) - '0';  
 }//如果字符不是数字并且不是空格，或者到了最后一位  
 if (!Character.isDigit(s.charAt(i)) && s.charAt(i) != ' ' || i == n - 1) {  
 switch (preSign) {//判断数字前的运算符，如果是+  
 case '+':  
 stack.push(num);//将num入栈  
 break;  
 case '-':  
 stack.push(-num);//如果是-，将-num入栈  
 break;  
 case '\*':  
 stack.push(stack.pop() \* num);//如果是\*将栈顶出栈\*num再入栈  
 break;  
 default:  
 stack.push(stack.pop() / num);//如果是/将栈顶出栈/num再入栈  
 }//更新前置符号，重置num=0  
 preSign = s.charAt(i);  
 num = 0;  
 }  
 }  
 int ans = 0;  
 while (!stack.isEmpty()) {//将栈内元素累加就是结果  
 ans += stack.pop();  
 }  
 return ans;  
 }  
}

# 递归

***//求斐波那契数列的第 n 项***public class \_8 {  
 //fib[1]是1，从fib[2]开始，fib[i]=fib[i - 1]+fib[i - 2]  
 public int Fibonacci(int n) {  
 int[] fib = new int[40];  
 fib[1] = 1;  
 for (int i = 2; i < fib.length; i++)  
 fib[i] = fib[i - 1] + fib[i - 2];  
 return fib[n];  
 }  
}

***//可以用2\*1 小矩形横或竖着覆盖更大的矩形。用n个2\*1矩形无重叠覆盖2\*n矩形有多少种方法***public class \_9 {  
 public int RectCover(int n) {  
 if (n <= 2) //n小于等于2，只有n种  
 return n;  
 int pre2 = 1, pre1 = 2;  
 int result = 0;  
 //n>2时，result = pre2 + pre1；pre2 = pre1；pre1 = result  
 for (int i = 3; i <= n; i++) {  
 result = pre2 + pre1;  
 pre2 = pre1;  
 pre1 = result;  
 }  
 return result; //最后返回result  
 }  
}

***//一只青蛙一次可以跳1级或2级。该青蛙跳一个n级台阶共多少种***public class \_10 {  
 public int JumpFloor(int n) {  
 if (n <= 2) return n;//和前一题完全一样  
 int pre2 = 1, pre1 = 2; int result = 1;  
 for (int i = 2; i < n; i++) {  
 result = pre2 + pre1;  
 pre2 = pre1;  
 pre1 = result;  
 }  
 return result;  
 }  
}

***//一只青蛙一次可跳1级、2级... 也可跳n级，青蛙跳n级台阶共多少跳法***public class \_11 {  
 public int JumpFloorII(int target) {  
 return (int) Math.pow(2, target - 1); //返回2的n-1次方  
 }  
}

***//递归 求 1+2+3+...+n，要求不能使用乘除法***public class \_69 {  
 public int Sum\_Solution(int n) {  
 int sum = n;  
 //n如果大于0就执行后面的(sum += Sum\_Solution(n - 1))，这部分肯定>0  
 boolean b = (n > 0) && ((sum += Sum\_Solution(n - 1)) > 0);  
 return sum;  
 }  
}

***//递归 求两整数之和，不得使用 +、-、\*、/***public class \_70 {  
 public int Add(int a, int b) {  
 //如果b是0，返回a，否则递归a^b和(a & b) << 1  
 return b == 0 ? a : Add(a ^ b, (a & b) << 1);  
 }  
}

***//递归 数字n代表括号对数，请生成所有可能的且有效的括号组合***public class \_22 {  
 ArrayList[] cache = new ArrayList[100];  
 public List<String> generate(int n) {  
 if (cache[n] != null) { //如果缓存数组n位不为空，返回  
 return cache[n];  
 }  
 ArrayList<String> ans = new ArrayList<String>(); //存放结果数组  
 if (n == 0) ans.add(""); //如果n=0，返回空字符串  
 else {  
 for (int c = 0; c < n; ++c) { //c从0开始  
 for (String left : generate(c)) { //左半部分是generate(c)  
 //右半部分是generate(n-1-c)  
 for (String right : generate(n - 1 - c)) {  
 //再ans中拼接(left)right  
 ans.add("(" + left + ")" + right);  
 }  
 }  
 }  
 }  
 cache[n] = ans; //将ans赋予cache[n]  
 return ans; //返回ans  
 }  
 public List<String> generateParenthesis(int n) {  
 return generate(n);  
 }  
}

# 回溯

***//回溯法 无重复元素整数数组candidates和目标target，找出数组中使数字和为target的不同组合，数字可重复。如至少一个数被选次数不同则两组合不同***public class \_39 {  
 public List<List<Integer>> combinationSum(int[] candidates, int target) {  
 List<List<Integer>> ans = new ArrayList<List<Integer>>(); //构造记录答案数组，是一个双层数组  
 List<Integer> combine = new ArrayList<Integer>(); //用来表示结果数组中的每个子数组  
 dfs(candidates, target, ans, combine, 0);  
 return ans;  
 }  
  
 public void dfs(int[] candidates, int target, List<List<Integer>> ans, List<Integer> combine, int idx) {  
 if (idx == candidates.length) { //如果下标达到数组长度，返回  
 return;  
 }  
 if (target == 0) { //如果target=0表明找到了一组，将数字组合加入ans  
 ans.add(new ArrayList<Integer>(combine));  
 return;  
 }  
 dfs(candidates, target, ans, combine, idx + 1); // 继续找id+1  
 // 如果当前数字总和还不到target  
 if (target - candidates[idx] >= 0) { //将当前数字加入combine  
 combine.add(candidates[idx]); //更新target继续找  
 dfs(candidates, target - candidates[idx], ans, combine, idx);  
 combine.remove(combine.size() - 1); //回退时删除最后一个数字  
 }  
 }  
}

***//回溯 判断矩阵是否存在一条含某字符串路径，可从矩阵任一格开始，每步可向上下左右移一格，某格不能进两次***public class \_13 {  
 //next二维数组代表可以移动的方式  
 private final static int[][] next = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};  
 private static int rows; private static int cols;  
 //array为原数组 rows和cols为行列数 str为要找的目标字符串  
 public static boolean hasPath(char[] array, int rows, int cols, char[] str) {  
 if (rows == 0 || cols == 0) return false; //如果行或列为0，返回false  
 \_13.rows = rows;  
 \_13.cols = cols;  
 boolean[][] marked = new boolean[rows][cols]; //构造标记数组，用于回溯  
 char[][] matrix = buildMatrix(array);  
 //两层循环，如果回溯寻找制定字符串成功，返回true，找到最后也没找到就返回false  
 for (int i = 0; i < rows; i++)  
 for (int j = 0; j < cols; j++)  
 if (backtracking(matrix, str, marked, 0, i, j))  
 return true;  
 return false;  
 }  
 private static boolean backtracking(char[][] matrix, char[] str, boolean[][] marked, int pathLen, int r, int c) {  
 if (pathLen == str.length) return true; //如果pathLen已经等于str.length，代表全部字符都找到了，返回true  
 //这一行能通过表明目标字符串的第pathLen个字符找到了，如果r、c越界或者当前搜索的矩阵元素不等于待寻找字符或者当前元素标记过了，返回false  
 if (r < 0 || r >= rows || c < 0 || c >= cols || matrix[r][c] != str[pathLen] || marked[r][c]) {  
 return false;  
 }  
 //将该位置标记为已使用，找目标字符串下一个字符，因此pathLen + 1，可以向四个方向找，因此遍历next  
 //0 -1; 0 1; -1 0; 1 0分别代表行不动，列-1，也就是向下找，以此类推，向上找，向左找，向右找  
 marked[r][c] = true;  
 for (int[] n : next)  
 if (backtracking(matrix, str, marked, pathLen + 1, r + n[0], c + n[1]))  
 return true;  
 marked[r][c] = false;  
 return false;  
 }  
 //将一维数组构造成二维矩阵  
 private static char[][] buildMatrix(char[] array) {  
 char[][] matrix = new char[rows][cols];  
 for (int r = 0, idx = 0; r < rows; r++)  
 for (int c = 0; c < cols; c++)  
 matrix[r][c] = array[idx++];  
 return matrix;  
 }  
}

***//输入数字n，按顺序打印从1到最大n位数。如输入3，则打印1到最大3位数即999***public class \_17 {  
 public static void print1ToMaxOfNDigits(int n) {  
 if (n <= 0) //如果n为负或0，直接返回  
 return;  
 char[] number = new char[n]; //构造n位的空字符数组  
 print1ToMaxOfNDigits(number, 0); //初始digit为0  
 }  
  
 private static void print1ToMaxOfNDigits(char[] number, int digit) {  
 if (digit == number.length) { //递归退出判断，如果digit等于字符数组长度，就打印并返回  
 printNumber(number);  
 return;  
 } //否则遍历给number数组每一位赋值，从0递增到9  
 for (int i = 0; i < 10; i++) {  
 number[digit] = (char) (i + '0');  
 print1ToMaxOfNDigits(number, digit + 1); //递归number数组的下一位  
 }  
 }  
  
 private static void printNumber(char[] number) {  
 //将某个字符数组打印出来，先跳过前面的0  
 int index = 0;  
 while (index < number.length && number[index] == '0')  
 index++;  
 //依次打印每个字符，最后打印换行  
 while (index < number.length)  
 System.out.print(number[index++]);  
 System.out.println();  
 }  
}

***//回溯 一个字符串按字典序打印该字符串字符的所有排列***public class \_41 {  
 private ArrayList<String> ret = new ArrayList<>(); //用来存放最终结果字符串  
  
 public ArrayList<String> Permutation(String str) {  
 if (str.length() == 0) return ret; //如果字符串长度为0，直接返回空数组  
 char[] chars = str.toCharArray(); //字符串转换为字符数组  
 Arrays.sort(chars); //字符数组排序  
 backtracking(chars, new boolean[chars.length], new StringBuilder()); //回溯排序后的字符数组  
 return ret;  
 }  
 //回溯法  
 private void backtracking(char[] chars, boolean[] hasUsed, StringBuilder s) {  
 //如果拼接字符串s长度和chars长度相等，说明拼好了一个结果字符串，放入ret  
 if (s.length() == chars.length) {  
 ret.add(s.toString());  
 return;  
 }  
 for (int i = 0; i < chars.length; i++) {  
 //判断第i个位置字符串是否用过  
 if (hasUsed[i]) continue;  
 //当原始字符串内有重复字符时，这里去重  
 if (i != 0 && chars[i] == chars[i - 1] && !hasUsed[i - 1]) continue;  
 //将第i个字符设置为已使用，将字符拼接到s  
 hasUsed[i] = true;  
 s.append(chars[i]);  
 backtracking(chars, hasUsed, s); //继续向下一个字符回溯  
 //某一条路径走不通了，将s的最后一个字符剔除并将第i位置设置为没用过  
 s.deleteCharAt(s.length() - 1);  
 hasUsed[i] = false;  
 }  
 }  
}

***//回溯法 填充空格解数独。数独部分空格已填入了数字，空白格用 '.' 表示。填充所有.为数字***public class \_37 {  
 //line和columns记录每一行列每个数字出现次数，subboxes记录每个小九宫格中每个数字出现次数  
 private boolean[][] line = new boolean[9][9];  
 private boolean[][] column = new boolean[9][9];  
 private boolean[][][] block = new boolean[3][3][9];  
 private boolean valid = false;  
 private List<int[]> spaces = new ArrayList<int[]>(); //用于记录空白位置  
  
 public void solveSudoku(char[][] board) {  
 for (int i = 0; i < 9; ++i) {  
 for (int j = 0; j < 9; ++j) {  
 if (board[i][j] == '.') { //遍历原数独，遇到.就把ij放入空白数组  
 spaces.add(new int[]{i, j});  
 } else {  
 int digit = board[i][j] - '0' - 1; //否则遇到数字，指定位置记录数字-1出现过  
 line[i][digit] = column[j][digit] = block[i / 3][j / 3][digit] = true;  
 }  
 }  
 }  
 dfs(board, 0); //然后解数独  
 }  
  
 public void dfs(char[][] board, int pos) {  
 if (pos == spaces.size()) { //如果位置达到了空白空间大小，说明解完了，返回  
 valid = true; //用于判断是否解完标志位  
 return;  
 }  
  
 int[] space = spaces.get(pos); //否则解pos位置应该填的数  
 int i = space[0], j = space[1];  
 for (int digit = 0; digit < 9 && !valid; ++digit) {  
 //遍历digit从0到8，并且当前还没解完，判断行、列、对角线都不能出现过该数字  
 if (!line[i][digit] && !column[j][digit] && !block[i / 3][j / 3][digit]) {  
 //将该位置填充为digit，更新行、列、对角线记录情况  
 line[i][digit] = column[j][digit] = block[i / 3][j / 3][digit] = true;  
 board[i][j] = (char) (digit + '0' + 1);  
 dfs(board, pos + 1); //继续回溯pos+1位置  
 //回退时需要更新行、列、对角线记录情况为false  
 line[i][digit] = column[j][digit] = block[i / 3][j / 3][digit] = false;  
 }  
 }  
 }  
}

***//回溯法 两个整数n和k，返回[1, n]中所有可能的 k 个数组合***public class \_77 {  
 private List<List<Integer>> ans = new ArrayList<>(); //存放答案  
  
 public List<List<Integer>> combine(int n, int k) {  
 //因为是1到n，所以start从1开始  
 getCombine(n, k, 1, new ArrayList<>()); //list存放一种结果  
 return ans;  
 }  
  
 public void getCombine(int n, int k, int start, List<Integer> list) {  
 if (k == 0) { //如果k减到0，将结果放入ans并返回，因为下面是不断k-1  
 ans.add(new ArrayList<>(list));  
 return;  
 } //i从start开始到n-k+1，因为i后面至少要有k-1个元素，否则肯定找不到k个元素的组合  
 for (int i = start; i <= n - k + 1; i++) {  
 list.add(i); //将i加入list  
 getCombine(n, k - 1, i + 1, list); //继续找k-1和i+1  
 list.remove(list.size() - 1); //回退时移除最后一个元素  
 }  
 }  
}

***//回溯法 皇后可攻击同行或同列或同斜线。将n皇后放在n×n棋盘上使皇后不能攻击。返回所有解决方案。'Q' 和 '.'代表皇后和空位***public class \_51 {  
 public List<List<String>> solveNQueens(int n) {  
 List<List<String>> solutions = new ArrayList<List<String>>(); //保存结果，双层数组  
 int[] queens = new int[n]; //queens[i]代表第i行皇后防止的位置，因为每一行只一个皇后  
 Arrays.fill(queens, -1); //先把queens填充为-1  
 Set<Integer> columns = new HashSet<Integer>(); //列和对角线Set  
 Set<Integer> diagonals1 = new HashSet<Integer>();  
 Set<Integer> diagonals2 = new HashSet<Integer>();  
 //回溯，传入结果、queens、n、row、三个set  
 backtrack(solutions, queens, n, 0, columns, diagonals1, diagonals2);  
 return solutions;  
 }  
 public void backtrack(List<List<String>> solutions, int[] queens, int n, int row, Set<Integer> columns,  
 Set<Integer> diagonals1, Set<Integer> diagonals2) {  
 if (row == n) { //如果行号=n代表回溯结束，利用queens产生一种结果，将结果放入结果集  
 List<String> board = generateBoard(queens, n);  
 solutions.add(board);  
 } else { //否则i从0到n  
 for (int i = 0; i < n; i++) {  
 if (columns.contains(i)) { //如果列包含i就下一轮循环  
 continue;  
 } //如果对角线包含i，也下一轮循环  
 int diagonal1 = row - i; //对角线1是row-i  
 if (diagonals1.contains(diagonal1)) { continue; }  
 int diagonal2 = row + i;//对角线2是row+i  
 if (diagonals2.contains(diagonal2)) {  
 continue;  
 }  
 queens[row] = i; //通过了上面，就把queens[row] = i  
 columns.add(i); //更新列和对角线set  
 diagonals1.add(diagonal1);  
 diagonals2.add(diagonal2);  
 //继续回溯row+1行  
 backtrack(solutions, queens, n, row + 1, columns, diagonals1, diagonals2);  
 queens[row] = -1; //回退时将queens[row] = -1，列和对角线中移除对应值  
 columns.remove(i);  
 diagonals1.remove(diagonal1);  
 diagonals2.remove(diagonal2);  
 }  
 }  
 }  
 //根据queens和n产生字符串代表二维矩阵 board = [".Q..","...Q","Q...","..Q."]  
 public List<String> generateBoard(int[] queens, int n) {  
 List<String> board = new ArrayList<String>();  
 for (int i = 0; i < n; i++) {  
 char[] row = new char[n]; //每一行是大小n的字符数组  
 Arrays.fill(row, '.'); //先都填充为.  
 row[queens[i]] = 'Q'; //该行的queens[i]位置设置为皇后Q  
 board.add(new String(row)); //将每一行的字符串加入board  
 }  
 return board;  
 }  
}

***//回溯法 数组nums和整数target。向数组每个数前添'+'或'-'，串联所有数，返回运算结果等于target不同表达式数***public class \_494 {  
 int count = 0; //保存表达式数  
  
 public int findTargetSumWays(int[] nums, int target) {  
 //回溯找target，下标从0开始，sum为当前得到的数字和  
 backtrack(nums, target, 0, 0);  
 return count;  
 }  
  
 public void backtrack(int[] nums, int target, int index, int sum) {  
 if (index == nums.length) {//如果下标到了数组长度  
 if (sum == target) { //如果sum=target，说明找到一种表达式，count++  
 count++;  
 }  
 } else {//否则继续回溯index+1、sum+nums[index]和index+1、sum-nums[index]  
 //代表分别给index数字加+和-符号  
 backtrack(nums, target, index + 1, sum + nums[index]);  
 backtrack(nums, target, index + 1, sum - nums[index]);  
 }  
 }  
}

***//回溯法 给一个不含重复数字数组nums返回所有可能的全排列***public class \_46 {  
 public List<List<Integer>> permute(int[] nums) {  
 List<List<Integer>> res = new ArrayList<List<Integer>>(); //存放结果数组，是一个双层数组  
 List<Integer> output = new ArrayList<Integer>(); //找到的一个全排列结果  
 for (int num : nums) { //先把num按顺序加入output  
 output.add(num);  
 }  
 int n = nums.length; //n记录数组总长，回溯退出条件  
 backtrack(n, output, res, 0); //first代表当前找到第几位了  
 return res;  
 }  
  
 public void backtrack(int n, List<Integer> output, List<List<Integer>> res, int first) {  
 if (first == n) { // 所有数都填完了，就把output加入res  
 res.add(new ArrayList<Integer>(output));  
 }  
 for (int i = first; i < n; i++) { //否则从first开始找  
 // 交换first和i  
 Collections.swap(output, first, i);  
 // 继续递归填first+1  
 backtrack(n, output, res, first + 1);  
 // 撤销操作，再次交换first和i  
 Collections.swap(output, first, i);  
 }  
 }  
}

***//回溯法 给一个包含重复数字的序列nums返回所有不重复的全排列***public class \_47 {  
 boolean[] vis;  
 public List<List<Integer>> permuteUnique(int[] nums) {  
 List<List<Integer>> ans = new ArrayList<List<Integer>>(); //保存结果，是双层数组  
 List<Integer> perm = new ArrayList<Integer>(); //存放一个结果  
 vis = new boolean[nums.length]; //用于标记某个数字是否用过  
 Arrays.sort(nums); //先排序  
 backtrack(nums, ans, 0, perm); //回溯  
 return ans;  
 }  
 public void backtrack(int[] nums, List<List<Integer>> ans, int idx, List<Integer> perm) {  
 if (idx == nums.length) { //如果idx达到了数组长度，说明找到了一个组合，加入ans返回  
 ans.add(new ArrayList<Integer>(perm));  
 return;  
 }  
 for (int i = 0; i < nums.length; ++i) { //i从0到num的长度  
 //如果i用过了或者i和i-1相等且i-1没用过，就跳过  
 if (vis[i] || (i > 0 && nums[i] == nums[i - 1] && !vis[i - 1])) {  
 continue;  
 }  
 perm.add(nums[i]); //将元素i加入perm  
 vis[i] = true; //更新i为已使用  
 backtrack(nums, ans, idx + 1, perm); //继续回溯idx+1  
 vis[i] = false; //回退时更新i为未使用  
 perm.remove(idx); //将idx元素从perm移除  
 }  
 }  
}

***//回溯法 由若干括号和字母组成的字符串s，删除最小量无效括号使字符串有效。返回所有可能结果***public class \_301 {  
 private List<String> res = new ArrayList<String>(); //存放结果  
  
 public List<String> removeInvalidParentheses(String s) {  
 //如果左括号数严格小于右括号则表达式无效。因此统计左右括号次数。  
 //遍历到左括号时lremove加1。遍历到右括号时如果lremove不为0就−1；如果为0，rremove加1。得到值就是各自最少删除数  
 int lremove = 0;  
 int rremove = 0;  
 for (int i = 0; i < s.length(); i++) { //计算需要删除的左右括号数  
 if (s.charAt(i) == '(') {  
 lremove++;  
 } else if (s.charAt(i) == ')') {  
 if (lremove == 0) {  
 rremove++;  
 } else {  
 lremove--;  
 }  
 }  
 }//执行删除操作  
 helper(s, 0, lremove, rremove);  
 return res;//返回结果  
 }  
  
 private void helper(String str, int start, int lremove, int rremove) {  
 if (lremove == 0 && rremove == 0) {  
 if (isValid(str)) {  
 res.add(str);  
 }//如果待删除的左右括号数都是0，且str有效就将str加入res并返回  
 return;  
 }//否则从start开始遍历字符串  
 for (int i = start; i < str.length(); i++) {//如果i不是start且i和i-1字符相同，下一轮循环  
 if (i != start && str.charAt(i) == str.charAt(i - 1)) {  
 continue;  
 }  
 // 如果剩余的字符无法满足去掉的数量要求，直接返回  
 if (lremove + rremove > str.length() - i) {  
 return;  
 }  
 // 如果还需要删除左括号且当前字符是(，将第i个字符去掉，继续回溯i，l-1  
 if (lremove > 0 && str.charAt(i) == '(') {  
 helper(str.substring(0, i) + str.substring(i + 1), i, lremove - 1, rremove);  
 }  
 // 如果还需要删除右括号且当前字符是)，将第i个字符去掉，继续回溯i，r-1  
 if (rremove > 0 && str.charAt(i) == ')') {  
 helper(str.substring(0, i) + str.substring(i + 1), i, lremove, rremove - 1);  
 }  
 }  
 }  
  
 //ok 判断某个字符串是否有效  
 private boolean isValid(String str) {  
 int cnt = 0; //计算str的左右括号数，如果相等就返回true  
 for (int i = 0; i < str.length(); i++) {  
 if (str.charAt(i) == '(') {  
 cnt++;  
 } else if (str.charAt(i) == ')') {  
 cnt--;  
 if (cnt < 0) {  
 return false;  
 }  
 }  
 }  
 return cnt == 0;  
 }  
}

***//回溯 一个仅含数字 2-9 的字符串，返回电话号码所有能表示的字母组合***public class \_17 {  
 public List<String> letterCombinations(String digits) {  
 List<String> combinations = new ArrayList<String>(); //记录字符串数组结果  
 if (digits.length() == 0) { //如果字符串数组为空，返回空结果集  
 return combinations;  
 } //构造数字到字母映射表  
 Map<Character, String> phoneMap = new HashMap<Character, String>() {{  
 put('2', "abc");put('3', "def");put('4', "ghi");put('5', "jkl");  
 put('6', "mno");put('7', "pqrs");put('8', "tuv");put('9', "wxyz");  
 }};  
 backtrack(combinations, phoneMap, digits, 0, new StringBuffer());//回溯  
 return combinations;  
 }  
  
 public void backtrack(List<String> combinations, Map<Character, String> phoneMap,  
 String digits, int index, StringBuffer combination) {  
 if (index == digits.length()) {//如果index等于传入字符串长度，表明找到一个可能结果，放入结果集  
 combinations.add(combination.toString());  
 } else { //否则获取index位置数字  
 char digit = digits.charAt(index);  
 String letters = phoneMap.get(digit); //获取该数字对应字母  
 int lettersCount = letters.length(); //计算对应的字母个数  
 for (int i = 0; i < lettersCount; i++) {  
 combination.append(letters.charAt(i));//针对每个字母，将该字母追加到结果字符串  
 backtrack(combinations, phoneMap, digits, index + 1, combination); //递归index+1  
 combination.deleteCharAt(index); //回溯的时候要删除结果的index位置字符  
 }  
 }  
 }  
}

# Dfs-bfs

***//m\*n方格。机器人从(0, 0)开始动，每次能向左右上下移动一格，不能进入行和列坐标的数位之和大于k的格子  
//\* 例如，当k为18时，机器人能进入格(35,37)，因为 3+5+3+7=18。但不能进入(35,38)。问能达到多少格子***public class \_14 {  
 private static final int[][] next = {{0, -1}, {0, 1}, {-1, 0}, {1, 0}};  
 private int cnt = 0; //能到达多少格子  
 private int rows; //行数  
 private int cols; //列数  
 private int threshold; //传入阈值  
 private int[][] digitSum; //存储列坐标数位之和的二维数组  
 //该方法返回能到达多少格子  
 public int movingCount(int threshold, int rows, int cols) {  
 this.rows = rows; this.cols = cols; this.threshold = threshold;  
 initDigitSum();  
 boolean[][] marked = new boolean[rows][cols];  
 dfs(marked, 0, 0);  
 return cnt;  
 }  
 private void dfs(boolean[][] marked, int r, int c) {  
 //如果出界了或者rc位置被用过了，直接返回  
 if (r < 0 || r >= rows || c < 0 || c >= cols || marked[r][c]) return;  
 marked[r][c] = true; //否则说明这个格子即将被使用，标记为被使用  
 //然后判断该格子如果超出限制，直接返回  
 if (this.digitSum[r][c] > this.threshold) return;  
 cnt++; //否则说明这个格子可以达到，格子数增加  
 //向四个方向继续找下一个格子能否达到  
 for (int[] n : next)  
 dfs(marked, r + n[0], c + n[1]);  
 }  
 //该方法根据二维数组值计算出行列坐标数位之和的二维数组，一定对称的  
 private void initDigitSum() {  
 //构造一维坐标和数组，数组长度是行列中的较大值  
 int[] digitSumOne = new int[Math.max(rows, cols)]; //计算一维坐标和数组各元素值  
 for (int i = 0; i < digitSumOne.length; i++) {  
 int n = i;  
 while (n > 0) {  
 digitSumOne[i] += n % 10;  
 n /= 10;  
 }  
 }  
 this.digitSum = new int[rows][cols]; //构造二维坐标和数组，元素ij是一维i+一维j的值  
 for (int i = 0; i < this.rows; i++)  
 for (int j = 0; j < this.cols; j++)  
 this.digitSum[i][j] = digitSumOne[i] + digitSumOne[j];  
 }  
}

***//bfs begin和end转换序列是begin -> s1 -> s2.. -> sk：相邻单词差一个字母。si在wordList中。begin不需在wordList。  
//给两个单词begin和 end和字典wordList，返回从begin到end的最短转换序列单词数。如不存在转换序列返回0***public class \_127 {  
 //给每个单词标号id。创建由 word 到 id 的映射 wordId  
 Map<String, Integer> wordId = new HashMap<String, Integer>();  
 //保存边，edge[idx]是个数组，代表从idx可以一步到达哪些节点的id数组  
 List<List<Integer>> edge = new ArrayList<List<Integer>>();  
 int nodeNum = 0; //记录节点数  
  
 public int ladderLength(String beginWord, String endWord, List<String> wordList) {  
 for (String word : wordList) { //遍历单词列表，为每个单词构造虚拟节点添加边  
 addEdge(word);  
 }  
 addEdge(beginWord); //给开始单词添加边  
 //如果添加完所有节点和边，单词映射表不包含结束单词，肯定没法转换，返回0  
 if (!wordId.containsKey(endWord)) {  
 return 0;  
 }  
 int[] dis = new int[nodeNum]; //存放到指定节点的最短距离  
 Arrays.fill(dis, Integer.MAX\_VALUE); //最开始设置为最大值  
 int beginId = wordId.get(beginWord), endId = wordId.get(endWord); //获取开始id和结束id  
 dis[beginId] = 0; //到达初始节点距离是0，因为不需要变换  
  
 Queue<Integer> que = new LinkedList<Integer>(); //构造队列  
 que.offer(beginId); //起点加入队列  
 while (!que.isEmpty()) { //广度搜索  
 int x = que.poll(); //弹出队首id  
 if (x == endId) { //如果队首id是结束id表明找到了路径，  
 return dis[endId] / 2 + 1; //返回到endId的距离/2+1  
 }  
 for (int it : edge.get(x)) { //否则遍历包含x的所有边的节点id  
 if (dis[it] == Integer.MAX\_VALUE) { //如果还没遇到过  
 dis[it] = dis[x] + 1; //就更新dis[it]=dis[x]+1，因为从x需要再更新一次  
 que.offer(it); //将it节点加入队列  
 }  
 }  
 }  
 return 0; //遍历都没找到，就返回0  
 }  
  
 public void addEdge(String word) {  
 addWord(word); //先把单词加入表中，增加一条空边  
 int id1 = wordId.get(word); //获取单词对应的节点值id1  
 char[] array = word.toCharArray(); //单词转换成字符数组  
 int length = array.length;  
 for (int i = 0; i < length; ++i) { //遍历字符数组  
 char tmp = array[i]; //更新第i位为\*  
 array[i] = '\*';  
 //构造虚拟的新单词  
 String newWord = new String(array);  
 addWord(newWord); //添加新单词  
 int id2 = wordId.get(newWord); //从映射表中获取新单词的id2  
 //id1和id2能互相转换，增加两条边  
 edge.get(id1).add(id2);  
 edge.get(id2).add(id1);  
 array[i] = tmp; //恢复原单词  
 }  
 }  
  
 public void addWord(String word) {  
 //如果映射表不包含该单词，将其加入表中，新增一条空边  
 if (!wordId.containsKey(word)) {  
 wordId.put(word, nodeNum++);  
 edge.add(new ArrayList<Integer>());  
 }  
 }  
}

***//dfs mxn矩阵board由'X'和'O'，找到所有被'X'围绕的'O'用'X'填充。边界'O'不被填充为'X'***public class \_130 {  
 int n, m; //所有不被包围的O都直接或间接与边界O相连  
 public void solve(char[][] board) {  
 n = board.length; //n行m列  
 if (n == 0) return; //如果数组为空，返回  
 m = board[0].length;  
 for (int i = 0; i < n; i++) {  
 //遍历搜索第i行第0列和第m-1列，即左右边界  
 dfs(board, i, 0);  
 dfs(board, i, m - 1);  
 } //遍历搜索第0行第i列和第n-1行，即上下边界  
 //i从1到m-1，因为上下边界搜索的时候不需要搜索四个顶点了  
 //在左右边界搜索的时候已经搜索过了  
 for (int i = 1; i < m - 1; i++) {  
 dfs(board, 0, i);  
 dfs(board, n - 1, i);  
 } //至此已经将和边界相邻的O都改成了A  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < m; j++) {  
 //遍历二维矩阵，将A替换成O，将O替换为X  
 if (board[i][j] == 'A') {  
 board[i][j] = 'O';  
 } else if (board[i][j] == 'O') {  
 board[i][j] = 'X';  
 }  
 }  
 }  
 }  
 public void dfs(char[][] board, int x, int y) {  
 //如果x，y越界或者x,y位置不是O，直接返回  
 if (x < 0 || x >= n || y < 0 || y >= m || board[x][y] != 'O') return;  
 board[x][y] = 'A'; //否则代表是O，将其标记为A  
 //继续搜索x周围的四个节点  
 dfs(board, x + 1, y);  
 dfs(board, x - 1, y);  
 dfs(board, x, y + 1);  
 dfs(board, x, y - 1);  
 }  
}

***//dfs 由'1'（陆地）和 '0'（水）组成二维网格，计算岛屿数。岛屿被水包围且岛屿由相邻1连接。网格四条边被水包围。***public class \_200 {  
 //类似于lc的\_130，只不过那道题边界不算被包围  
 void dfs(char[][] grid, int r, int c) {  
 int nr = grid.length; //nr行nc列  
 int nc = grid[0].length;  
 if (r < 0 || c < 0 || r >= nr || c >= nc || grid[r][c] == '0') {  
 return; //如果rc下标越界或者rc位置是0就返回  
 } //否则将rc位置设为0，继续找相邻上下左右四个点  
 grid[r][c] = '0';  
 dfs(grid, r - 1, c);  
 dfs(grid, r + 1, c);  
 dfs(grid, r, c - 1);  
 dfs(grid, r, c + 1);  
 }  
  
 public int numIslands(char[][] grid) {  
 if (grid == null || grid.length == 0) {//如果数组为空，返回0  
 return 0;  
 }//网格看成无向图，扫描图以1为起点dfs。搜到的1被重标记为0。岛屿数就是dfs次数  
 int nr = grid.length; //nr行nc列  
 int nc = grid[0].length;  
 int num\_islands = 0; //岛屿数  
 for (int r = 0; r < nr; ++r) {  
 for (int c = 0; c < nc; ++c) {  
 //遍历数组，如果rc位置是1，将岛屿数++并从rc开始dfs相邻的1  
 if (grid[r][c] == '1') {  
 ++num\_islands;  
 dfs(grid, r, c);  
 }  
 }  
 }  
 return num\_islands; //返回岛屿数  
 }  
}

***//bfs 基因序列为8字符，字符是A、C、G和T之一。给start和end及基因库bank，变化后基因必须位于bank，返回start变为end最少次数，无法变返回-1***public class \_433 {  
 public int minMutation(String start, String end, String[] bank) {  
 Set<String> cnt = new HashSet<String>();//将基因库去重  
 Set<String> visited = new HashSet<String>();//存放已经遍历的基因列表  
 char[] keys = {'A', 'C', 'G', 'T'};//可能的字符列表  
 for (String w : bank) {//将基因库内容加入cnt去重  
 cnt.add(w);  
 }  
 if (start.equals(end)) {//如果start和end相同，返回0步  
 return 0;  
 }//如果基因库不包含end，返回-1  
 if (!cnt.contains(end)) {  
 return -1;  
 }//BFS要创建队列  
 Queue<String> queue = new ArrayDeque<String>();//建队列  
 queue.offer(start);//将start入队  
 visited.add(start);//将start标记为已使用  
 int step = 1;//初始化步数  
 while (!queue.isEmpty()) {//循环判断队列不为空  
 int sz = queue.size();//获取队列大小  
 for (int i = 0; i < sz; i++) {//遍历队列每个元素  
 String curr = queue.poll();//弹出队首元素  
 for (int j = 0; j < 8; j++) {//基因序列8个字符，每次替换一个  
 for (int k = 0; k < 4; k++) {//k从0到3，因为有四种字符AGCT  
 if (keys[k] != curr.charAt(j)) {//如果基因j位字符和keys[k]不相等  
 StringBuffer sb = new StringBuffer(curr);  
 sb.setCharAt(j, keys[k]);//将基因j位替换位keys[k]构造next基因  
 String next = sb.toString();  
 //如果next基因没被用过并且在基因库中  
 if (!visited.contains(next) && cnt.contains(next)) {  
 if (next.equals(end)) {//如果该基因和end相同，返回步数  
 return step;  
 }//否则将next入队，标记位已使用  
 queue.offer(next);  
 visited.add(next);  
 }  
 }  
 }  
 }  
 }//每次层序遍历步数+1  
 step++;  
 }  
 return -1;//如果最后都没找到end，返回-1  
 }  
}

***//dfs 航线列表tickets，tickets[i]=[fromi,toi]表示出发降落点。对行程排序从JFK开始。按字典序返回最小行程组合***public class \_332 {  
 public List<String> findItinerary(List<List<String>> tickets) {  
 List<String> ans = new LinkedList<>();//构造结果链表  
 if (tickets == null || tickets.size() == 0)//如果票列表为空，返回空数组  
 return ans;  
 Map<String, List<String>> graph = new HashMap<>();//构造图map  
 for (List<String> pair : tickets) {//遍历所有机票  
 //将某个地点能到的所有地点放入graph中  
 List<String> nbr = graph.computeIfAbsent(pair.get(0), k -> new LinkedList<>());  
 nbr.add(pair.get(1));  
 }  
 graph.values().forEach(x -> x.sort(String::compareTo)); //按目的地升序排序  
 visit(graph, "JFK", ans);//从JFK开始dfs遍历图  
 return ans;  
 }  
  
 // DFS方式遍历图，当走到不能走为止，再将节点加入到答案  
 private void visit(Map<String, List<String>> graph, String src, List<String> ans) {  
 List<String> nbr = graph.get(src);//从图中获取src能到的所有dest  
 while (nbr != null && nbr.size() > 0) {//遍历所有dest  
 String dest = nbr.remove(0);//从列表中移除并继续遍历dest  
 visit(graph, dest, ans);  
 }//将src逆序加入  
 ans.add(0, src); // 逆序插入  
 }  
}

***//bfs 2x3板上有5块砖数字1~5,空缺用0。一次移动为选择0与相邻数交换.最终board是[[1,2,3],[4,5,0]]谜被解。给初始board，求解谜最少移动次数，不能解返回-1***public class \_773 {  
 //位置编号为[0 1 2][3 4 5]，则各个位置相邻位置数组如下：0挨着1 3，1挨着0 2 4...  
 int[][] neighbors = {{1, 3}, {0, 2, 4}, {1, 5}, {0, 4}, {1, 3, 5}, {2, 4}};  
 public int slidingPuzzle(int[][] board) {  
 StringBuffer sb = new StringBuffer();//构造序列字符串  
 for (int i = 0; i < 2; ++i) {  
 for (int j = 0; j < 3; ++j) {  
 sb.append(board[i][j]);//遍历初始board获取初始字符串  
 }  
 }  
 String initial = sb.toString();  
 if ("123450".equals(initial)) { return 0; //如果初始字符串就是123450直接返回0不需要移动 }  
 int step = 0; //存储步数  
 Queue<String> queue = new LinkedList<String>();//构造队列  
 queue.offer(initial);//将初始状态加入队列  
 Set<String> seen = new HashSet<String>();//构造该状态是否经历过  
 seen.add(initial);//将初始值加入经历Set  
 while (!queue.isEmpty()) {//当队列不为空时遍历  
 ++step;//步数+1  
 int size = queue.size();  
 for (int i = 0; i < size; ++i) {//依次弹出队列元素  
 String status = queue.poll();  
 for (String nextStatus : get(status)) { //遍历当前状态能达到下一个状态  
 if (!seen.contains(nextStatus)) {//如果该状态没见过  
 if ("123450".equals(nextStatus)) {//如果是123450，直接返回步数  
 return step;  
 }//否则将状态放入队列和seen  
 queue.offer(nextStatus);  
 seen.add(nextStatus);  
 }  
 }  
 }  
 }  
 return -1;//没法转换返回-1  
 }  
 //在status中找出0所在位置x，每个与x相邻位置y，交换status[x]与status[y]，等于一次操作  
 public List<String> get(String status) {  
 List<String> ret = new ArrayList<String>();  
 char[] array = status.toCharArray();//将当前状态转换为字符数组  
 int x = status.indexOf('0');//找到0的位置  
 //从邻居矩阵中获取该位置能交换的位置y，遍历y  
 for (int y : neighbors[x]) {  
 swap(array, x, y);//交换字符串x和y处字符  
 ret.add(new String(array));//将结果放入ret  
 swap(array, x, y);//再换回来进行下次交换  
 }  
 return ret;//返回status一次操作能得到的所有新状态  
 }  
 //交换array的x和y处字符  
 public void swap(char[] array, int x, int y) {  
 char temp = array[x];  
 array[x] = array[y];  
 array[y] = temp;  
 }  
}

***//dfs 修numCourses门课。prerequisites[i]=[ai, bi]表示学ai必须先学bi。判断能否学完所有课程***public class \_207 {  
 List<List<Integer>> edges;  
 int[] visited; //标记某节点是否为搜索过0是未搜索，1是搜索中，2是已完成  
 boolean valid = true; //标记是否存在可能的排序  
  
 public boolean canFinish(int numCourses, int[][] prerequisites) {  
 edges = new ArrayList<List<Integer>>(); //构造numCourses条边  
 for (int i = 0; i < numCourses; ++i) {  
 edges.add(new ArrayList<Integer>());  
 }  
 visited = new int[numCourses]; //构造标记数组，初始大小numCourses  
 for (int[] info : prerequisites) { //遍历依赖数组，用依赖创建边  
 edges.get(info[1]).add(info[0]);  
 }  
 for (int i = 0; i < numCourses && valid; ++i) {  
 if (visited[i] == 0) { //如果第i和点没搜索过，搜索它  
 dfs(i);  
 }  
 }  
 return valid;  
 }  
  
 public void dfs(int u) {  
 visited[u] = 1; //将节点标记为1搜索中  
 for (int v : edges.get(u)) { //获取节点u需要的全部前置课程  
 if (visited[v] == 0) { //如果前置课程v没搜索过，就搜索v  
 dfs(v);  
 if (!valid) { //如果不能找到，直接返回  
 return;  
 }//如果前置课程也搜索中，则不能产生排序标记false返回  
 } else if (visited[v] == 1) {  
 valid = false;  
 return;  
 }  
 }  
 visited[u] = 2;//搜索完了所有前置课程，将u标记为2搜索结束  
 }  
}

# 数学

***//一个double的base和int的exponent，求base的exponent 次方***public class \_16 {  
 public double Power(double base, int exponent) {  
 if (exponent == 0) return 1; //如果指数是0，返回1  
 if (exponent == 1) return base;//如果指数是1，返回base  
 boolean isNegative = false;  
 if (exponent < 0) { //记录指数是否为负值  
 exponent = -exponent;  
 isNegative = true;  
 }  
 //递归每次将指数减半  
 double pow = Power(base \* base, exponent / 2);  
 if (exponent % 2 != 0) //如果指数是奇数，还要再乘一个base  
 pow = pow \* base;  
 return isNegative ? 1 / pow : pow; //如果指数为正，返回pow，否则返回pow倒数  
 }  
}

***//数学 n人围圈。指定m，编号0开始报数到m-1人出去且不回圈中，从下一人继续，直到剩下最后一人求其位置***public class \_67 {  
 public int LastRemaining\_Solution(int n, int m) {  
 if (n == 0) return -1; //如果0个数，返回-1  
 if (n == 1) return 0; //如果1个数，返回下标位0  
 //否则递归n-1和m，结果再+m再对n取余  
 return (LastRemaining\_Solution(n - 1, m) + m) % n;  
 }  
}

***//数学 一个整数，输出该数二进制中 1 的个数***public class \_74 {  
 public static int NumberOf1(int n) {  
 int count = 0;  
 while (n != 0) { //循环判断n不为0  
 n = n & (n - 1); //每次n=n&(n - 1)  
 count++; //count++  
 }  
 return count;  
 }  
}

***//数学 一个整数，输出该数二进制中0的个数。***public class \_75 {  
 public static int findZero(int n) {  
 int count = 0;  
 while (n != 0) { //循环判断n不为0  
 if ((n & 1) != 1) //如果n&1不是1，代表找到了一个0，count++  
 count++;  
 n >>>= 1; //n右移一位  
 }  
 return count;  
 }  
}

***//数组 整数数组nums和整数target，数组找出和为target两个数返回下标。假设只有一个答案。元素在答案里不能重复***public class \_1 {  
 public int[] twoSum(int[] nums, int target) {  
 Map<Integer, Integer> hashtable = new HashMap<Integer, Integer>();  
 for (int i = 0; i < nums.length; ++i) {  
 //key+value=target，找到了key，说明找到了一对数字和为target，返回两个数的下标  
 if (hashtable.containsKey(target - nums[i])) {  
 return new int[]{hashtable.get(target - nums[i]), i};  
 }  
 hashtable.put(nums[i], i); //没找到就把数字和下标放入map  
 }  
 return new int[0];  
 }  
}

***//数学 罗马数字转整数***public class \_13 {  
 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);  
 }}; //映射表只需要记录1-5，因为4=IV也是由1和5组成的，只不过是5的符号-1的符号  
  
 public int romanToInt(String s) {  
 int ans = 0;  
 int n = s.length(); //获取传入字符串长度  
 for (int i = 0; i < n; ++i) { //逐步取出字符串中每位字符，一定是映射表中的一个  
 int value = symbolValues.get(s.charAt(i)); //计算字符对应的值  
 //如果第i位的值小于i+1(它后面一位)，说明要把i位的值减去，否则要加上  
 if (i < n - 1 && value < symbolValues.get(s.charAt(i + 1))) ans -= value;  
 else ans += value;  
 }  
 return ans; //返回结果  
 }  
}

***//数学 被除数dividend和除数 divisor。将两数相除，不用乘法、除法和mod***public class \_29 {  
 public long divide(int dividend, int divisor) {  
 if (dividend == 0) return 0; //如果被除数是0，返回0  
 if (divisor == 1) return dividend; //如果除数是1，返回被除数  
 if (divisor == -1) { //如果除数是-1，返回被除数相反数  
 if (dividend > Integer.MIN\_VALUE) return -dividend;// 只要不是最小整数返回相反数  
 return Integer.MAX\_VALUE;// 是最小的就返回最大整数  
 }  
 long a = dividend; //将被除数和除数转换成长整型  
 long b = divisor;  
 int sign = 1; //两个数相除后的符号位  
 if ((a > 0 && b < 0) || (a < 0 && b > 0)) {  
 sign = -1;  
 }  
 a = a > 0 ? a : -a; //将被除数和除数都转换成正数  
 b = b > 0 ? b : -b;  
 long res = div(a, b); //执行a除以b得到结果  
 //如果符号位是正，就返回res，否则返回-res，记得进行最大值判断  
 if (sign > 0) return res > Integer.MAX\_VALUE ? Integer.MAX\_VALUE : res;  
 return -res;  
 }  
  
 long div(long a, long b) {  
 if (a < b) return 0; //如果a小于b返回0，递归退出条件  
 long count = 1;  
 long tb = b; // 在后面的代码中不更新b  
 while ((tb + tb) <= a) { //循环判断如果a大于等于b的两倍  
 count = count + count; // 最小解翻倍  
 tb = tb + tb; // 当前测试值也翻倍  
 } //跳出循环时tb已经足够大了，此时a不能超过tb两倍，结果+count并递归a-tb和b  
 return count + div(a - tb, b);  
 }  
}

***//LRU LRUCache(int capacity) 以capacity为容量初始LRU。int get(int key) 如果key在缓存返回关键字值否则返回-1。  
//void put(int key, int value)如果key存在则变更value，不存在则插入k-v。如插入导致超容量应逐出最久未使用key***public class \_146 {  
 class LRUCache {  
 class DLinkedNode { //双向链表  
 int key; //存放k和v  
 int value;  
 DLinkedNode prev;//前后指针  
 DLinkedNode next;  
 }  
 //哈希表辅以双向链表实现，哈希表存放key到节点  
 private Map<Integer, DLinkedNode> cache = new HashMap<Integer, DLinkedNode>();  
 private int size; //当前节点数  
 private int capacity; //最大容量  
 private DLinkedNode head, tail; //头尾节点  
 public LRUCache(int capacity) {//构造方法  
 //初始化size、容量。头尾节点，头尾互相指向对方  
 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) {//从缓存中获取指定key的节点  
 DLinkedNode node = cache.get(key);  
 if (node == null) { return -1; }//如果节点为空，返回-1  
 //将node删除再插入到头节点，保证最新访问的在最前面  
 moveToHead(node);  
 return node.value;//返回节点值  
 }  
 //放入  
 public void put(int key, int value) {  
 DLinkedNode node = cache.get(key);//获取key对应的node  
 if (node == null) { // 如果node不存在，根据k和v创建新节点  
 DLinkedNode newNode = new DLinkedNode(key, value);  
 cache.put(key, newNode);// node添加进哈希表  
 addToHead(newNode); // node添加至链表头  
 ++size; //更新size  
 if (size > capacity) {  
 DLinkedNode tail = removeTail();//如果超出容量删除尾节点  
 cache.remove(tail.key);//删除哈希表中对应项  
 --size;//size--  
 }  
 } else {  
 node.value = value; //如果node存在，更新value并移到头部  
 moveToHead(node);  
 }  
 }  
 //节点添加到头，双向链表添加节点要移动四个指针  
 private void addToHead(DLinkedNode node) {  
 //head是虚拟节点，因此让node的pre指向head，node的next指向head的next  
 node.prev = head;  
 node.next = head.next;  
 //head的next的pre指向node，head的next指向node  
 head.next.prev = node;  
 head.next = node;  
 }  
 //删除某节点，要移动两个指针  
 private void removeNode(DLinkedNode node) {  
 node.prev.next = node.next; //node的前节点next指向node的next，跳过node  
 //node的后节点的pre指向node的pre，也跳过node实现删除node  
 node.next.prev = node.prev;  
 }  
 //先将node删除，然后将node添加到头  
 private void moveToHead(DLinkedNode node) {  
 removeNode(node); addToHead(node);  
 }  
 //删除尾节点  
 private DLinkedNode removeTail() {  
 //tail也是虚拟节点，获取tail的pre是当前的尾节点，将其删除  
 DLinkedNode res = tail.prev;  
 removeNode(res);  
 return res;  
 }  
 }  
}

***//数学 返回32位有符号整数x数字反转后结果。如反转后超过范围返回0。如x=123翻转为321***public class \_7 {  
 public int reverse(int x) {  
 int res = 0;  
 while (x != 0) {  
 int tmp = x % 10; //每次取末尾数字  
 //判断是否大于最大32位整数  
 if (res > 214748364 || (res == 214748364 && tmp > 7)) return 0;  
 //判断是否小于最小32位整数  
 if (res < -214748364 || (res == -214748364 && tmp < -8)) return 0;  
 res = res \* 10 + tmp; //更新res和x  
 x /= 10;  
 }  
 return res;  
 }  
}

***//数学 一个整数数组nums元素互不相同。返回该数组所有可能子集。不能包含重复子集***public class \_78 {  
 List<Integer> t = new ArrayList<Integer>(); //t存放每种可能的结果  
 List<List<Integer>> ans = new ArrayList<List<Integer>>(); //存最终结果集  
 public List<List<Integer>> subsets(int[] nums) {  
 int n = nums.length; //计算输入数组长度n  
 //mask从0到2^n  
 for (int mask = 0; mask < (1 << n); ++mask) {  
 t.clear(); //先清空  
 for (int i = 0; i < n; ++i) { //i从0到n  
 //mask和2^i与不为0，就把nums[i]加入t  
 if ((mask & (1 << i)) != 0) {  
 t.add(nums[i]);  
 }  
 } //内层循环每次找一个t，将t放入ans中  
 ans.add(new ArrayList<Integer>(t));  
 }  
 return ans; //返回ans  
 }  
}

***//数学 根据逆波兰求表达式值。算符包括+、-、\*、/。每个运算对象可以是整数，也可以是另一个逆波兰式***public class \_150 {  
 public int evalRPN(String[] tokens) {  
 Deque<Integer> stack = new LinkedList<Integer>();//栈存储操作数  
 int n = tokens.length; //计算传入波兰式长度n  
 for (int i = 0; i < n; i++) {//遍历每个字符  
 String token = tokens[i];  
 if (isNumber(token)) {//如果字符是数字直接转换成整形入栈  
 stack.push(Integer.parseInt(token));  
 } else {//否则是表达式，弹出栈顶两个元素  
 int num2 = stack.pop();  
 int num1 = stack.pop();  
 switch (token) {//针对表达式对栈顶两元素计算，结果入栈  
 case "+":  
 stack.push(num1 + num2);  
 break;  
 case "-":  
 stack.push(num1 - num2);  
 break;  
 case "\*":  
 stack.push(num1 \* num2);  
 break;  
 case "/":  
 stack.push(num1 / num2);  
 break;  
 default:  
 }  
 }  
 }  
 return stack.pop();//最后返回栈顶元素就是结果  
 }  
 public boolean isNumber(String token) {//不是+-\*/就是数字  
 return !("+".equals(token) || "-".equals(token) || "\*".equals(token) || "/".equals(token));  
 }  
}

***//数学 一个整数n，判断该是否是 2 的幂次方***public class \_231 {  
 public boolean isPowerOfTwo(int n) {  
 //正数且n & (n - 1) = 0，就是2的幂  
 return n > 0 && (n & (n - 1)) == 0;  
 }  
}

***//数学 颠倒给定32位无符号数二进制位***public class \_190 {  
 public int reverseBits(int n) {  
 int rev = 0; //i从0开始，i<32且n不为0  
 for (int i = 0; i < 32 && n != 0; ++i) {  
 //|=就是位运算的+= rev+=(n&1)<<(31-i0)  
 rev |= (n & 1) << (31 - i);  
 n >>>= 1;//n右移  
 }  
 return rev;//返回rev  
 }  
}

***//数学 两个整数间的汉明距离指两个数对应二进制位不同的位置数。给两个整数x和y返回之间的汉明距离***public class \_461 {//本质相当于找一个二进制数s中1的个数  
 public int hammingDistance(int x, int y) {  
 int s = x ^ y, ret = 0;//让x和y亦或得到s，然后找s的1的个数  
 while (s != 0) {//循环判断s是否到0  
 ret += s & 1;//判断s最后一位如果是1，ret++  
 s >>= 1; //s右移1位，继续判断下一位  
 }  
 return ret;//返回结果  
 }  
}

***//数学 设计HashSet。void add(key)插入key。bool contains(key)是否存在key。void remove(key)将key删除，如果集合没key不做***public class \_705 {  
 class MyHashSet {  
 private static final int BASE = 769;//取一个较大质数作为base用于哈希  
 private LinkedList[] data;  
 //构造方法  
 public MyHashSet() {  
 data = new LinkedList[BASE];//初始化指定大小数组  
 for (int i = 0; i < BASE; ++i) {  
 data[i] = new LinkedList<Integer>(); //数组的每个节点都是个链表  
 }  
 }  
 //添加  
 public void add(int key) {  
 int h = hash(key);//对key进行哈希  
 Iterator<Integer> iterator = data[h].iterator();//获取data指定下标处的链表  
 while (iterator.hasNext()) {//遍历链表，如果已经存在key直接返回  
 Integer element = iterator.next();  
 if (element == key) {  
 return;  
 }  
 }//否则将key加入链表最后一个节点  
 data[h].offerLast(key);  
 }  
 //删除  
 public void remove(int key) {  
 int h = hash(key);//计算key的哈希  
 Iterator<Integer> iterator = data[h].iterator();//获取data指定下标处的链表  
 while (iterator.hasNext()) {//遍历链表，如果元素值为key，就删除链表该节点  
 Integer element = iterator.next();  
 if (element == key) {  
 data[h].remove(element);  
 return;  
 }  
 }  
 }  
 //判断存在  
 public boolean contains(int key) {  
 int h = hash(key);//计算key的哈希值  
 Iterator<Integer> iterator = data[h].iterator();//获取data指定下标处的链表  
 while (iterator.hasNext()) {//遍历链表每个元素，如果存在值为key的元素，返回true  
 Integer element = iterator.next();  
 if (element == key) {  
 return true;  
 }  
 }  
 return false;//遍历结束找不到就返回false  
 }  
 //hash方法，让key对BASE取余  
 private int hash(int key) {  
 return key % BASE;  
 }  
 }  
}

***//数学 设计HashMap：MyHashMap()初始化。void put(int key, int value)插入(key, value)。如key已存在则更新value。  
//int get(int key)返回key映射value；如果不含映射返回-1。void remove(key)如果映射中存在key，则移除key和value***public class \_706 {  
 class MyHashMap {  
 //构造一个内部类pair保存key和value  
 private class Pair {  
 private int key;  
 private int value;  
 public Pair(int key, int value) {  
 this.key = key;  
 this.value = value;  
 }  
 public int getKey() {  
 return key;  
 }  
 public int getValue() {  
 return value;  
 }  
 public void setValue(int value) {  
 this.value = value;  
 }  
 }  
  
 private static final int BASE = 769;//其余和hashSet相同  
 private LinkedList[] data;  
 //初始化的时候链表每个节点都是pair，不是Integer  
 public MyHashMap() {  
 data = new LinkedList[BASE];  
 for (int i = 0; i < BASE; ++i) {  
 data[i] = new LinkedList<Pair>();  
 }  
 }  
 //put也是判断pair  
 public void put(int key, int value) {  
 int h = hash(key);  
 Iterator<Pair> iterator = data[h].iterator();  
 while (iterator.hasNext()) {  
 Pair pair = iterator.next();  
 if (pair.getKey() == key) {  
 pair.setValue(value);  
 return;  
 }  
 }  
 data[h].offerLast(new Pair(key, value));  
 }  
 //get也是判断对  
 public int get(int key) {  
 int h = hash(key);  
 Iterator<Pair> iterator = data[h].iterator();  
 while (iterator.hasNext()) {  
 Pair pair = iterator.next();  
 if (pair.getKey() == key) {  
 return pair.value;  
 }  
 }  
 return -1;  
 }  
 //删除也是删除pair  
 public void remove(int key) {  
 int h = hash(key);  
 Iterator<Pair> iterator = data[h].iterator();  
 while (iterator.hasNext()) {  
 Pair pair = iterator.next();  
 if (pair.key == key) {  
 data[h].remove(pair);  
 return;  
 }  
 }  
 }  
 //哈希方法不变  
 private int hash(int key) {  
 return key % BASE;  
 }  
 }  
}

***//数学 复数用字符串表示"实部+虚部i"，实部是整数，虚部也是整数，两个复数num1和num2，返回表示乘积的字符串***public class \_537 {  
 public String complexNumberMultiply(String num1, String num2) {  
 //将字符串按+或i切割得到 实部 虚部数组  
 String[] complex1 = num1.split("\\+|i");  
 String[] complex2 = num2.split("\\+|i");  
 //解析得到整形的 实部1 虚部1 实部2 虚部2  
 int real1 = Integer.parseInt(complex1[0]);  
 int imag1 = Integer.parseInt(complex1[1]);  
 int real2 = Integer.parseInt(complex2[0]);  
 int imag2 = Integer.parseInt(complex2[1]);  
 //最后结果是​(real1×real2−imag1×imag2)+(real1×imag2+imag1×real2)×i  
 return String.format("%d+%di", real1 \* real2 - imag1 \* imag2, real1 \* imag2 + imag1 \* real2);  
 }  
}

***//数学 字符数组tasks表示任务列表。字母表示不同任务。任务需1时间。任1时间可完成一任务或待命。同任务间有n冷却，计算完成任务最短时间***public class \_621 {  
 public int leastInterval(char[] tasks, int n) {  
 Map<Character, Integer> freq = new HashMap<Character, Integer>();  
 int maxExec = 0; //最多的执行次数  
 for (char ch : tasks) {//遍历任务列表，计算各类任务执行次数  
 int exec = freq.getOrDefault(ch, 0) + 1;  
 freq.put(ch, exec);  
 maxExec = Math.max(maxExec, exec);//得到最后执行次数  
 }  
 int maxCount = 0; //具有最多执行次数的任务数量  
 Set<Map.Entry<Character, Integer>> entrySet = freq.entrySet();  
 for (Map.Entry<Character, Integer> entry : entrySet) {//遍历任务执行次数map  
 int value = entry.getValue();  
 if (value == maxExec) {//计算出最多执行次数的任务数count  
 ++maxCount;  
 }  
 }//计算(maxExec−1)(n+1)+maxCount，和任务长度比较，返回较大值  
 return Math.max((maxExec - 1) \* (n + 1) + maxCount, tasks.length);  
 }  
}

***//数学 停车场有大，中和小车位。实现ParkingSystem类：  
//ParkingSystem(int big, int medium, int small)初始化ParkingSystem类，三个参数对应每种车位数  
//bool addCar(int carType) 检查是否有carType的车位。大，中，小，分别用1，2和3，车只能停对应尺寸***public class \_1603 {  
 class ParkingSystem {  
 int big, medium, small;//记录大、中、小三类车位数  
 //构造方法，初始化大、中、小的值  
 public ParkingSystem(int big, int medium, int small) {  
 this.big = big;  
 this.medium = medium;  
 this.small = small;  
 }  
 //加车  
 public boolean addCar(int carType) {  
 if (carType == 1) {//判断车类型值，如果对应车位数>0就--返回真  
 if (big > 0) {  
 big--;  
 return true;  
 }  
 } else if (carType == 2) {  
 if (medium > 0) {  
 medium--;  
 return true;  
 }  
 } else if (carType == 3) {  
 if (small > 0) {  
 small--;  
 return true;  
 }  
 } //否则返回false  
 return false;  
 }  
 }  
}

# 动态规划

***//匹配包括 '.' 和 '\*' 正则表达式，'.' 表示任一个字符，'\*' 表示它前面字符可出现任意次（含0次）***public class \_20 {  
 public boolean match(char[] str, char[] pattern) { //str是目标字符串，pattern是正则串  
 int m = str.length, n = pattern.length; //获取目标字符串长度m和正则字符串长度n  
 //构造动态规划数组，长度分别是m+1和n+1，默认都是false  
 //dp[i][j]代表长度i的字符串和长度j的模式串是否匹配  
 boolean[][] dp = new boolean[m + 1][n + 1];  
 dp[0][0] = true; //两个字符串都为空肯定匹配，因此00元素为true  
 for (int i = 1; i <= n; i++) //如果目标串为空，模式串有长度  
 //如果模式串i - 1字符为\*，则模式串i和i-2相同，因为\*前面字符可以出现0次  
 if (pattern[i - 1] == '\*')  
 dp[0][i] = dp[0][i - 2];  
 for (int i = 1; i <= m; i++)  
 for (int j = 1; j <= n; j++)  
 //两层循环遍历，如果目标串i和模式串j字符相同，则dp[i][j] = dp[i - 1][j - 1]  
 //如果模式串j字符(pattern[j - 1])为.，能匹配任意单字符，dp[i][j] 也= dp[i - 1][j - 1]  
 if (str[i - 1] == pattern[j - 1] || pattern[j - 1] == '.')  
 dp[i][j] = dp[i - 1][j - 1];  
 //如果模式串j字符是\*，要分情况  
 else if (pattern[j - 1] == '\*')  
 //如果模式串j-1字符和目标串i字符相同或者模式串j-1字符是.(也代表相同)  
 if (pattern[j - 2] == str[i - 1] || pattern[j - 2] == '.') {  
 //则dp[i][j] |= dp[i][j - 1]，即a\*看做a  
 //dp[i][j] |= dp[i - 1][j]，即a\*看做aa  
 //dp[i][j] |= dp[i][j - 2]，即a\*看做空  
 dp[i][j] |= dp[i][j - 1]; dp[i][j] |= dp[i - 1][j]; dp[i][j] |= dp[i][j - 2];  
 } else //如果模式串j-1字符和目标串i字符不相同，只能把a\*看做空  
 dp[i][j] = dp[i][j - 2];  
 return dp[m][n]; //返回结果  
 }  
}

***//动态规划 数字翻译字符串：1译“a”，2译“b”..26译“z”。可能有多种如12258共5种:abbeh，lbeh，aveh，abyh，lyh，计算有多少种译法***public class \_49 {  
 public int numDecodings(String s) {  
 //如果数字字符串为空，返回0种  
 if (s == null || s.length() == 0) return 0;  
 int n = s.length(); //获取字符串长度  
 int[] dp = new int[n + 1]; //构造dp数组，dp[i]代表前i个字符有多少种  
 dp[0] = 1;  
 dp[1] = s.charAt(0) == '0' ? 0 : 1; //如果第一个字符是0，有0种否则1种  
 for (int i = 2; i <= n; i++) {  
 int one = Integer.parseInt(s.substring(i - 1, i)); //获取第i-1个字符，转换整形  
 if (one != 0) //如果这个字符不是0，就先把i-1的值增到i上  
 dp[i] += dp[i - 1];  
 if (s.charAt(i - 2) == '0') //如果i-2字符是0，则不可能通过i-2和i-1这两个字符组装，继续循环  
 continue;  
 int two = Integer.parseInt(s.substring(i - 2, i)); //否则解析出两个字符，如果再26以内，就把i-2值增到i上  
 if (two <= 26)  
 dp[i] += dp[i - 2];  
 }  
 return dp[n];  
 }  
}

***//动态规划 只包含因子 2、3 和 5 的数称作丑数。把 1 当做第一个丑数。求从小到大第N个丑数***public class \_52 {  
 public static int GetUglyNumber\_Solution(int N) {  
 if (N <= 6) return N; //前六个丑数就是本身  
 int i2 = 0, i3 = 0, i5 = 0;  
 //i2、i3、i5代表已经乘了2、3、5的数字下标，例如i2=2，代表dp[0]，dp[1]、dp[2]都已经乘过2了  
 //所以下个丑数如果是通过×2得到，一定是dp[3]\*2，所以当dp[3]\*2是当前最小的丑数时，  
 //下个丑数就是它，i3和i5以此类推  
 int[] dp = new int[N]; dp[0] = 1;  
 for (int i = 1; i < N; i++) {  
 int next2 = dp[i2] \* 2, next3 = dp[i3] \* 3, next5 = dp[i5] \* 5;  
 dp[i] = Math.min(next2, Math.min(next3, next5));  
 if (dp[i] == next2)  
 i2++;  
 if (dp[i] == next3)  
 i3++;  
 if (dp[i] == next5)  
 i5++;  
 }  
 return dp[N - 1];  
 }  
}

***//动态规划 机器人位于mxn左上角，每次向下或右移一步，问总共有多少条不同路径***public class \_62 {  
 public int uniquePaths(int m, int n) {  
 //f(i,j) 表示从左上角走到 (i,j) 的路径数量  
 int[][] f = new int[m][n]; //构造m\*n矩阵  
 for (int i = 0; i < m; ++i) { //最上面一行每个点都只有一种走法  
 f[i][0] = 1;  
 }  
 for (int j = 0; j < n; ++j) { //最左边一列也都只有一种走法  
 f[0][j] = 1;  
 }  
 for (int i = 1; i < m; ++i) {  
 for (int j = 1; j < n; ++j) { //i从1到m，j从1到n遍历  
 //ij可能从[i - 1][j]也可能从[i][j - 1]到达，是二者之和  
 f[i][j] = f[i - 1][j] + f[i][j - 1];  
 }  
 }  
 return f[m - 1][n - 1]; //返回f[m - 1][n - 1]  
 }  
}

***//动态规划 n个骰子仍在地上，求点数和为 s 的概率***public class \_65 {  
 public List<Map.Entry<Integer, Double>> dicesSum(int n) {  
 final int face = 6; //一共有六个面  
 final int pointNum = face \* n; //最高可能出现多少点  
 long[][] dp = new long[n + 1][pointNum + 1];  
 //一个骰子产生各个点数的次数都是1  
 for (int i = 1; i <= face; i++)  
 dp[1][i] = 1;  
 for (int i = 2; i <= n; i++) //骰子个数i从2到n  
 for (int j = i; j <= pointNum; j++) //点数j从i到pointNum，因为i个骰子最小点数为i  
 for (int k = 1; k <= face && k <= j; k++)  
 //i个骰子j点可能来自i-1个骰子j-k点，其中k可能是1-6，六种情况求和就是dp[i][j]  
 dp[i][j] += dp[i - 1][j - k];  
 final double totalNum = Math.pow(6, n); //计算一共的额情况是6的n次方  
 List<Map.Entry<Integer, Double>> ret = new ArrayList<>();  
 //依次计算出各个点数的概率，n个骰子点数i出现次数是dp[n][i]，用dp[n][i] / totalNum  
 for (int i = n; i <= pointNum; i++)  
 ret.add(new AbstractMap.SimpleEntry<>(i, dp[n][i] / totalNum));  
 return ret;  
 }  
}

***//动态规划 给一个字符串s，找到s中最长回文子串***public class \_5 {  
 public String longestPalindrome(String s) {  
 int len = s.length();  
 if (len < 2) return s; //如果字符串长度为1，直接返回  
 int maxLen = 1;  
 int begin = 0;  
 // dp[i][j] 表示 s[i..j] 是否是回文串  
 boolean[][] dp = new boolean[len][len];  
 // 初始化：所有长度为 1 的子串都是回文串，斜对角线代表取出第i个字符这一个字符是回文串  
 for (int i = 0; i < len; i++) { dp[i][i] = true; }  
 char[] charArray = s.toCharArray();  
 for (int L = 2; L <= len; L++) { // 先枚举子串长度  
 for (int i = 0; i < len; i++) { // 枚举左边界  
 // 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得  
 int j = L + i - 1;  
 // 如果右边界越界，退出当前循环  
 if (j >= len) break;  
 //如果i和j字符不相等，设置为false  
 if (charArray[i] != charArray[j]) {  
 dp[i][j] = false;  
 } else {  
 //如果字符相等，继续判断：如果j和i间少于3个元素，i+1=j-1，在对角线上一定为true  
 if (j - i < 3) dp[i][j] = true;  
 else dp[i][j] = dp[i + 1][j - 1]; //否则dp[i][j] = dp[i+1][j-1]  
 }  
 //dp[i][L] == true 表示s[i..L] 是回文，记录回文长度和起始位置  
 if (dp[i][j] && j - i + 1 > maxLen) {  
 maxLen = j - i + 1;  
 begin = i;  
 }  
 }  
 }  
 return s.substring(begin, begin + maxLen); //返回最长子串  
 }  
}

***//动态规划 m\*n棋盘每格有礼物，礼物有价值，从左上每次向右或下移一格，到右下。求拿到最大值***public class \_50 {  
 public static int getMost(int[][] values) {  
 //如果数组为空，返回0  
 if (values == null || values.length == 0 || values[0].length == 0) return 0;  
 int n = values[0].length; //获取列数  
 int[] dp = new int[n];  
 for (int[] value : values) {  
 //遍历每一行数据，内层每一次循环都能得到第j行的最大值  
 dp[0] += value[0];  
 //dp[n]代表遍历到value的那一行的某个点的最大值  
 //因为只能向右或者向下走，所以某个点的路径只能是其上面点到自己或者其左边点到自己  
 //所有到某个点的路径最大值就是上面点或者左面点中的最大值再加上这个点的值  
 //第一轮循环dp[n]是1 11 14 22 代表从左上角到1 10 3 8这几个点的最大值  
 //第二轮循环dp[n]是13 15 24 30 代表从左上角到12 2 9 6这几个点的最大值  
 for (int i = 1; i < n; i++)  
 //dp[i]代表到某个点的上面点的最大值，dp[i-1]代表某个点左边点的最大值  
 //这两个值取max再加当前点的值，就是到当前点的最大值  
 dp[i] = Math.max(dp[i], dp[i - 1]) + value[i];  
 }  
 return dp[n - 1];  
 }  
}

***//动态规划 字符串s和t，计算s子序列中t出现的个数。子序列指删除或不删字符不干扰剩余字符相对位置组成的新串***public class \_115 {  
 public int numDistinct(String s, String t) {  
 int m = s.length(), n = t.length(); //计算s和t的长度m和n  
 if (m < n) { //如果s长度小于t，s肯定不包含t的子序列，返回0  
 return 0;  
 } //dp[i][j] 表示在 s[i:] 子序列中 t[j:] 出现个数  
 int[][] dp = new int[m + 1][n + 1]; //构造dp  
 //j=n 时，t[j:]为空串，空字符串是任何字符串的子序列，因此dp[i][n]=1  
 for (int i = 0; i <= m; i++) {  
 dp[i][n] = 1;  
 }  
 for (int i = m - 1; i >= 0; i--) { //i<m，j<n时  
 char sChar = s.charAt(i); //获取s[i]  
 for (int j = n - 1; j >= 0; j--) {  
 char tChar = t.charAt(j);//获取t[j]  
 //如果s[i]=t[j]，不代表s[i]要和t[j]匹配，这种情况有匹配的权力  
 //如果匹配，则考虑t[j+1:] 作为s[i+1:]子序列，为 dp[i+1][j+1]；  
 //如果不匹配，则考虑t[j:] 作为s[i+1:]子序列，为 dp[i+1][j]。  
 if (sChar == tChar) {  
 dp[i][j] = dp[i + 1][j + 1] + dp[i + 1][j];  
 } else {//s[i]！=t[j]时，s[i]不能和t[j]匹配，只考虑t[j:]作为s[i+1:]子序列，为 dp[i+1][j]  
 dp[i][j] = dp[i + 1][j];  
 }  
 }  
 }  
 return dp[0][0]; //返回dp[0][0]  
 }  
}

***//动态规划 含非负整数的mxn网格，找出从左上到右下路径使路径上数字总和最小。每次只能向下或右移一步***public class \_64 {  
 public int minPathSum(int[][] grid) {  
 if (grid == null || grid.length == 0 || grid[0].length == 0) {  
 return 0; //如果路径为空，返回0  
 }  
 int rows = grid.length, columns = grid[0].length; //获取行列数  
 int[][] dp = new int[rows][columns]; //构造dp数组  
 dp[0][0] = grid[0][0]; //初始化dp[0][0]就是[0][0]数字大小  
 for (int i = 1; i < rows; i++) { //初始化第0列的dp  
 dp[i][0] = dp[i - 1][0] + grid[i][0];  
 } //初始化第0行的dp  
 for (int j = 1; j < columns; j++) {  
 dp[0][j] = dp[0][j - 1] + grid[0][j];  
 }  
 for (int i = 1; i < rows; i++) { //i从1到row，j从1到col  
 for (int j = 1; j < columns; j++) {  
 //找到dp[i - 1][j]和dp[i][j - 1]较小值，再加grid[i][j]就是到[i][j]位置最短路径dp[i][j]  
 dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];  
 }  
 }  
 return dp[rows - 1][columns - 1]; //返回结果  
 }  
}

***//动态规划 机器人位于mxn左上角，每次只能向下或右移一步，有障碍物。求多少条不同路径，障碍物和空位分别1和0***public class \_63 {  
 public int uniquePathsWithObstacles(int[][] obstacleGrid) {  
 int n = obstacleGrid.length, m = obstacleGrid[0].length; //n行m列  
 int[] f = new int[m]; //用一维数组f代表到当前行某列的位置的最多种路径  
 //初始化f[0]，如果obstacleGrid[0][0]不是障碍，f[0]就是1  
 f[0] = obstacleGrid[0][0] == 0 ? 1 : 0;  
 for (int i = 0; i < n; ++i) { //i从0到n  
 for (int j = 0; j < m; ++j) { //j从0到m  
 if (obstacleGrid[i][j] == 1) { //如果遇到障碍，f[j] = 0，继续  
 f[j] = 0;  
 continue;  
 } //第0行的f[j]都是1，因此j从第一行开始迭代，没有障碍f[j] += f[j - 1]  
 if (j - 1 >= 0 && obstacleGrid[i][j - 1] == 0) {  
 f[j] += f[j - 1];  
 }  
 }  
 }  
 return f[m - 1]; //返回最后一行遍历完的f[m - 1]  
 }  
}

***//动态规划 两个单词word1和word2，返回word1转换成word2使用最少操作数，可以插一字符、删字符、换字符***public class \_72 {  
 public int minDistance(String word1, String word2) {  
 int n1 = word1.length(); //计算两个单词各自长度  
 int n2 = word2.length();  
 int[][] dp = new int[n1 + 1][n2 + 1]; //构造dp数组  
 // 第0行，word1为空变成word2最少步数，就是插入操作，因此依次+1  
 for (int j = 1; j <= n2; j++) dp[0][j] = dp[0][j - 1] + 1;  
 // 第0列，是word2为空，需要的最少步数就是删除操作，也依次+1  
 for (int i = 1; i <= n1; i++) dp[i][0] = dp[i - 1][0] + 1;  
 for (int i = 1; i <= n1; i++) { //i从1到n1，j从1到n2  
 for (int j = 1; j <= n2; j++) {  
 if (word1.charAt(i - 1) == word2.charAt(j - 1))//如果i-1和j-1位置字符相同  
 dp[i][j] = dp[i - 1][j - 1];  
 else //如果字符不同，就是dp[i-1][j-1], dp[i][j-1]), dp[i-1][j]的最小值再+1，分别对应替换、删除、插入  
 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]; //返回结果  
 }  
}

***//动态规划 字母A-Z编码：'A'->"1" 'B'->"2"..'Z'->"26"，如"11106"可映射为"AAJF"(1 1 10 6)或"KJF"(11 10 6)，求一个只含数字的串s解码方法的总数***public class \_91 {  
 public int numDecodings(String s) {  
 int n = s.length(); //获取字符串长度  
 int[] f = new int[n + 1]; //构造f数组  
 f[0] = 1; //空字符串有1种，解码出空字符串  
 for (int i = 1; i <= n; ++i) {//i从1到n  
 if (s.charAt(i - 1) != '0') { //如果i-1字符不是0，f[i] += f[i - 1]  
 f[i] += f[i - 1];  
 } //i要>1并且i-2字符不是0并且i-2字符拼上i-1字符这个两位数≤26，f[i] += f[i - 2]  
 if (i > 1 && s.charAt(i - 2) != '0' && ((s.charAt(i - 2) - '0') \* 10 + (s.charAt(i - 1) - '0') <= 26)) {  
 f[i] += f[i - 2];  
 }  
 }  
 return f[n]; //返回f[n]  
 }  
}

***//动态规划 整数数组prices，prices[i]表示股票第i天价。每天可以购买或出售股票。最多只持有一股。返回最大利润***public class \_122 {  
 public int maxProfit(int[] prices) {  
 int n = prices.length; //计算股票天数  
 //dp[i][0]表示第 i 天交易完手里没有股票的最大利润  
 //dp[i][1]表示第 i 天交易完手里持有一支股票的最大利润  
 int[][] dp = new int[n][2];  
 dp[0][0] = 0; //第0天没有股票利润为0  
 dp[0][1] = -prices[0]; //第0天有股票利润为-prices[0]  
 for (int i = 1; i < n; ++i) { //i从1到n  
 //dp[i][0]可能状态为前一天已没有股票即 dp[i−1][0]，或前一天有股票但卖出即dp[i−1][1]+prices[i]  
 dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i]);  
 //dp[i][1]可能状态前一天已有股票即dp[i−1][1]，或前一天没股票将其买入即dp[i−1][0]-prices[i]  
 dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i]);  
 }  
 return dp[n - 1][0]; //返回最后一天手里没有股票的值  
 }  
}

***//动态规划 求由n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***public class \_96 {  
 public int numTrees(int n) {  
 int[] dp = new int[n + 1];  
 dp[0] = 1; //0个节点的树有一种，空树  
 dp[1] = 1; //1个节点的树有一种  
 //i从2到n-1，j从1到i+1，dp[i]+=dp[j - 1]\*dp[i - j]  
 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]; //返回dp[n]  
 }  
}

***//动态规划 一个三角形triangle，找出自顶向下最小路径和。如果位于当前行下标i，下一步可移到下一行的i 或 i+1***public class \_120 {  
 public int minimumTotal(List<List<Integer>> triangle) {  
 int n = triangle.size(); //计算有多少层  
 //f[i][j] 表示从顶部走到位置(i,j)最小路径和  
 int[][] f = new int[n][n];  
 //f[0][0] 是位置(0，0)的值  
 f[0][0] = triangle.get(0).get(0);  
 for (int i = 1; i < n; ++i) {//i从1到n，j从1到n  
 //f[i][0]=f[i−1][0]+c[i][0]，即在第 i 行最左侧时只能从 i−1 行最左侧移动过来  
 f[i][0] = f[i - 1][0] + triangle.get(i).get(0);  
 for (int j = 1; j < i; ++j) {  
 //上一步只能在位置 (i−1,j−1) 或(i−1,j)。在这两个位置中选择小的来转移  
 f[i][j] = Math.min(f[i - 1][j - 1], f[i - 1][j]) + triangle.get(i).get(j);  
 }//f[i][i]=f[i−1][i−1] + c[i][i]，即第 i 行最右侧只能从 i−1 行最右侧移动过来  
 f[i][i] = f[i - 1][i - 1] + triangle.get(i).get(i);  
 }//答案是f[n−1][0] 到f[n−1][n−1] 中的最小值  
 int minTotal = f[n - 1][0];  
 for (int i = 1; i < n; ++i) {  
 minTotal = Math.min(minTotal, f[n - 1][i]);  
 }  
 return minTotal;  
 }  
}

***//动态规划+dfs 字符串s，将s分割成一些子串，使每个子串都是回文。返回所有可能分割方案***public class \_131 {  
 boolean[][] f; //f(i,j)表示s[i..j]是否为回文串  
 //存放结果数组，是一个二层数组  
 List<List<String>> ret = new ArrayList<List<String>>();  
 List<String> ans = new ArrayList<String>(); //存放一种可能的分割方式  
 int n; //保存输入字符串长度  
  
 public List<List<String>> partition(String s) {  
 n = s.length(); //计算字符串长度n  
 f = new boolean[n][n]; //构造f  
 for (int i = 0; i < n; ++i) { //最开始都填充为true  
 Arrays.fill(f[i], true);  
 }  
 //i≥j代表空串或单字符的串，为true  
 //f(i,j)={ True, i≥j  
 //​ f(i+1,j−1)∧(s[i]=s[j]), otherwise  
 for (int i = n - 1; i >= 0; --i) {//i逆向遍历  
 for (int j = i + 1; j < n; ++j) {  
 //j从i+1开始，利用状态方程更新f[i][j]  
 f[i][j] = (s.charAt(i) == s.charAt(j)) && f[i + 1][j - 1];  
 }  
 }  
 dfs(s, 0); //从第0位开始搜索字符串s  
 return ret;  
 }  
  
 public void dfs(String s, int i) {  
 if (i == n) { //如果i=n代表遍历完了原字符串  
 ret.add(new ArrayList<String>(ans)); //将ans放入ret返回  
 return;  
 }  
 for (int j = i; j < n; ++j) {//否则j从i开始  
 if (f[i][j]) { //如果i到j是回文串，就将i到j子串加入ans  
 ans.add(s.substring(i, j + 1));  
 dfs(s, j + 1); //继续回溯j+1  
 ans.remove(ans.size() - 1);//回退时删除ans最后一个串  
 }  
 }  
 }  
}

***//动态规划 数组第i个元素是股票第i天价格。计算最大利润。最多可完成两笔交易。必须在再购买前出售之前的股票***public class \_123 {  
 public int maxProfit(int[] prices) {  
 int n = prices.length; //计算股票天数  
 //只可能有四种状态：  
 //①只买了一次；②买一次卖一次；③完成一笔交易下买了第二次；④卖出了第二次。  
 //初始值分别如下  
 int buy1 = -prices[0], sell1 = 0; int buy2 = -prices[0], sell2 = 0;  
 for (int i = 1; i < n; ++i) { //i从1到n  
 //buy1在第 i天可以不进行任何操作，也可以在未进行操作下以prices[i]买入股票  
 buy1 = Math.max(buy1, -prices[i]);  
 //sell1在第 i天可以不进行任何操作，也可以在buy1下以 prices[i]卖出股票  
 sell1 = Math.max(sell1, buy1 + prices[i]);  
 //buy2在第 i天可以不进行任何操作，也可以在sell1下以prices[i]买入股票  
 buy2 = Math.max(buy2, sell1 - prices[i]);  
 //sell2在第 i天可以不进行任何操作，也可以在buy2下以 prices[i]卖出股票  
 sell2 = Math.max(sell2, buy2 + prices[i]);  
 }  
 return sell2; //最后返回sell2  
 }  
}

***//动态规划 字符串s和wordDict作字典。判断是否可以用字典单词拼出s。不要求字典单词全用，字典单词可重复用***public class \_139 {  
 public boolean wordBreak(String s, List<String> wordDict) {  
 Set<String> wordDictSet = new HashSet(wordDict); //单词列表去重  
 //dp[i]表示s前 i 个字符s[0..i−1] 是否能被拆分成若干个字典中的单词  
 boolean[] dp = new boolean[s.length() + 1];  
 dp[0] = true; //空字符串可以被拆分成单词表里的词  
 for (int i = 1; i <= s.length(); i++) {//遍历s的每个字符  
 //dp[i]=dp[j] && check(s[j..i−1])  
 //check(s[j..i−1]) 表示s[j..i−1] 是否出现在字典中  
 for (int j = 0; j < i; j++) { //因此j从0到i  
 //如果dp[i]=dp[j] && check(s[j..i−1])为真，dp[i]就是true，继续找下一个i  
 if (dp[j] && wordDictSet.contains(s.substring(j, i))) {  
 dp[i] = true;  
 break;  
 }  
 }  
 }  
 return dp[s.length()]; //返回dp[s.length()]  
 }  
}

***//动态规划 整数数组prices，prices[i]是股票第i天价。计算最大利润。最多完成k笔交易。再次买前必须出售之前股票***public class \_188 {  
 public int maxProfit(int k, int[] prices) {  
 if (prices.length == 0) { return 0; }//如果股票数组为空，返回0  
 int n = prices.length;//获取股票数组大小n  
 //因为一共n天，最多进行n/2次有效交易，同一天不断买入卖出没有意义  
 k = Math.min(k, n / 2);  
 //buy[i][j]表示进行 j 笔交易且持有股票的最大利润  
 int[][] buy = new int[n][k + 1];  
 //sell[i][j]表示进行 j 笔交易且当前不持有股票的最大利润  
 int[][] sell = new int[n][k + 1];  
 buy[0][0] = -prices[0];//第0天持有一只股票收益是-prices[0]  
 sell[0][0] = 0; //第0天手上没有股票收益是0  
 for (int i = 1; i <= k; ++i) {  
 //第0天不可能进行过任何交易，所有buy[0][1..k]和sell[0][1..k]设置为小值代表不合法  
 buy[0][i] = sell[0][i] = Integer.MIN\_VALUE / 2;  
 }  
 for (int i = 1; i < n; ++i) {//i从1到n，j从1到k  
 //buy[i][0]代表第i天进行过0次交易且有股票，可能在i-1天就有股票即buy[i - 1][0]  
 //也可能i-1天没有股票，i天买入，即sell[i - 1][0] - prices[i]  
 buy[i][0] = Math.max(buy[i - 1][0], sell[i - 1][0] - prices[i]);  
 for (int j = 1; j <= k; ++j) {  
 //buy[i][j]考虑当前持有的股票如果是第i天买入，那第i−1天不持有股票，对应sell[i−1][j]且扣除prices[i]  
 //如果不是第i天买入，那么第 i−1天持有股票，对应buy[i−1][j]因此：  
 //buy[i][j]=max{buy[i−1][j],sell[i−1][j]−price[i]}  
 buy[i][j] = Math.max(buy[i - 1][j], sell[i - 1][j] - prices[i]);  
 //sell[i][j]如果是第i天卖出，那么i−1天持有股票，对应buy[i−1][j−1]且增加prices[i]；  
 //如果不是第i天卖出，那么第i−1天不持有股票，对应sell[i−1][j]。因此：  
 //sell[i][j]=max{sell[i−1][j],buy[i−1][j−1]+price[i]}  
 sell[i][j] = Math.max(sell[i - 1][j], buy[i - 1][j - 1] + prices[i]);  
 }  
 }  
 // n天结束后，不持有股票利润一定优于持有股票利润，而交易数不是越多越好，因此答案为sell[n−1][0..k]中最大值  
 return Arrays.stream(sell[n - 1]).max().getAsInt();  
 }  
}

***//动态规划 整数数组nums，找出数组中乘积最大的非空连续子数组，返回所对应乘积***public class \_152 {  
 public int maxProduct(int[] nums) {  
 int length = nums.length; //计算数组长度  
 //因为存在负数，所以要同时更新最大和最小  
 int[] maxF = new int[length];//maxF(i)表示第i个元素结尾的乘积最大子数组的乘积  
 int[] minF = new int[length];//minF(i)表示第i个元素结尾的乘积最小子数组的乘积  
 //将nums分别复制到maxF和minF，代表每个数字不和前面任何数相乘的最大值和最小值，即等于数本身  
 System.arraycopy(nums, 0, maxF, 0, length);  
 System.arraycopy(nums, 0, minF, 0, length);  
 for (int i = 1; i < length; ++i) {  
 //i为0时，最大值和最小值都是它本身，因此i从1到l，更新maxF[i]和minF[i]  
 //maxF[i]是maxF[i-1]\*ai minF[i-1]\*ai ai这三个数中的max  
 //minF[i]是maxF[i-1]\*ai minF[i-1]\*ai ai这三个数中的min  
 maxF[i] = Math.max(maxF[i - 1] \* nums[i], Math.max(nums[i], minF[i - 1] \* nums[i]));  
 minF[i] = Math.min(minF[i - 1] \* nums[i], Math.min(nums[i], maxF[i - 1] \* nums[i]));  
 }//最后找出maxF数组中的最大值就是答案  
 int ans = maxF[0];  
 for (int i = 1; i < length; ++i) {  
 ans = Math.max(ans, maxF[i]);  
 }  
 return ans;  
 }  
}

***//动态规划 MxN二维网格从左上到右下，如果健康点降至0或以下立死，房间数代表健康点变化，每次向右或下，计算确保到右下所需最低初始健康点***public class \_174 {  
 public int calculateMinimumHP(int[][] dungeon) {  
 //n行m列  
 int n = dungeon.length, m = dungeon[0].length;  
 //  
 //从右下往左上动态规划，dp[i][j]表示从坐标(i,j)到终点所需最小生命值  
 int[][] dp = new int[n + 1][m + 1];  
 for (int i = 0; i <= n; ++i) {//最开始都设置为MAX  
 Arrays.fill(dp[i], Integer.MAX\_VALUE);  
 }//和右下角相邻的两个点值是1，只需保证到这两个点时至少有1点生命  
 dp[n][m - 1] = dp[n - 1][m] = 1;  
 for (int i = n - 1; i >= 0; --i) {  
 for (int j = m - 1; j >= 0; --j) {//i从n-1到0，j从m-1到0  
 //先获取i,j能到的两个点i+1,j和i,j+1的最小生命  
 int minn = Math.min(dp[i + 1][j], dp[i][j + 1]);  
 //用上面的最小生命减去i,j损耗的生命就是i,j点所需的生命，  
 //如果是负值或0代表该点无论多少生命都能通关，但生命最少为1，所以要和1比较取较大值  
 dp[i][j] = Math.max(minn - dungeon[i][j], 1);  
 }  
 }  
 return dp[0][0];//返回dp[0][0]  
 }  
}

***//动态规划 每房有一定现金，邻房同一晚被偷会报警。给代表每个房金额非负整数数组，计算不报警能偷到最高金额***public class \_198 {  
 public int rob(int[] nums) {  
 if (nums == null || nums.length == 0) {//如果数组为空，返回0  
 return 0;  
 }  
 int length = nums.length;  
 if (length == 1) { //如果只有一家，偷他家就是最高收益  
 return nums[0];  
 }//dp[i]表示前 i 间屋能偷到的最高总金额  
 int[] dp = new int[length]; //构造dp  
 dp[0] = nums[0]; //第0家能偷到的最高金额就是nums[0]  
 //只有两间屋，选金额较高的房屋偷窃  
 dp[1] = Math.max(nums[0], nums[1]);  
 for (int i = 2; i < length; i++) {//i从2开始  
 //偷第i间屋就不能偷第i−1间屋，总金额为dp[i-2]+nums[i],不偷第i间，总金额为dp[i-1]  
 dp[i] = Math.max(dp[i - 2] + nums[i], dp[i - 1]);  
 }  
 return dp[length - 1];//返回dp[i - 1]  
 }  
}

***//动态规划 给一个代表环形房屋金额的非负整数数组，相邻房屋同一晚被闯入会报警，计算不触警报下能偷到最高金额***public class \_213 {  
 public int rob(int[] nums) {  
 int length = nums.length; //计算数组长度  
 if (length == 1) { //如果只有一间，返回nums[0]  
 return nums[0];  
 } else if (length == 2) {//两间房返回价值更高的一间  
 return Math.max(nums[0], nums[1]);  
 }//返回从0偷到len-2或者1偷到len-1的更大值  
 return Math.max(robRange(nums, 0, length - 2), robRange(nums, 1, length - 1));  
 }  
  
 public int robRange(int[] nums, int start, int end) {  
 //计算dp[start]和dp[start+1]分别是first和second  
 int first = nums[start], second = Math.max(nums[start], nums[start + 1]);  
 for (int i = start + 2; i <= end; i++) { //i从start+2开始到end  
 int temp = second;  
 //只需存储前两间房屋最高总金额，first是i-2间，second是i-1间  
 //i可以从first过来再加nums[i]，也可以从second过来不偷nums[i]  
 second = Math.max(first + nums[i], second);  
 first = temp; //之前的second变成现在的first，因为i后移一位  
 }  
 return second;//返回second  
 }  
}

***//动态规划 由 '0' 和 '1' 组成的二维矩阵找到只包含 '1' 的最大正方形返回面积***public class \_221 {  
 public int maximalSquare(char[][] matrix) {  
 int maxSide = 0;  
 if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {  
 return maxSide; //如果矩阵为空，返回0  
 } //计算行列  
 int rows = matrix.length, columns = matrix[0].length;  
 //dp(i,j)表示(i,j)为右下角且只包含1的正方形边长最大值  
 int[][] dp = new int[rows][columns];//构造dp，初始值都是0  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < columns; j++) {//遍历行列  
 if (matrix[i][j] == '1') {//如果ij是1  
 //如果i或j有一个为 0，则(i,j)为右下角的最大正方形的边长只能是1  
 if (i == 0 || j == 0) {  
 dp[i][j] = 1;  
 } else {//dp(i,j)值由上方、左方和左上三个相邻dp决定，dp(i,j)等于三个相邻位置元素中最小值加1  
 dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) + 1;  
 } //找到一个dp[i][j]就更新最大边长  
 maxSide = Math.max(maxSide, dp[i][j]);  
 }  
 }  
 }//返回最大面积  
 return maxSide \* maxSide;  
 }  
}

***//动态规划 一个整数数组nums，找到其最长严格递增子序列长度，子序列不需要连续***public class \_300 {  
 public int lengthOfLIS(int[] nums) {  
 if (nums.length == 0) {//如果数组为空，返回长度0  
 return 0;  
 }//dp[i]为前i个元素，以第i个数字结尾的最长上升子序列长度  
 int[] dp = new int[nums.length];  
 dp[0] = 1;//只有第0个元素时，长度为1  
 int maxans = 1;  
 for (int i = 1; i < nums.length; i++) {//i从1到len  
 dp[i] = 1; //最开始认为dp[i]=1，即只考虑第i个元素  
 for (int j = 0; j < i; j++) {//j从0到i  
 //如果num[j]<num[i]，那么可以把j元素加进来  
 if (nums[i] > nums[j]) {  
 //dp[i]=max(dp[j])+1  
 dp[i] = Math.max(dp[i], dp[j] + 1);  
 }  
 } //每找到一个dp[i]就更新最大长度  
 maxans = Math.max(maxans, dp[i]);  
 }  
 return maxans;  
 }  
}

***//动态规划 整数n ，返回和为n的完全平方数最少数量***public class \_279 {  
 public int numSquares(int n) {  
 int[] f = new int[n + 1];//f[i]表示最少要多少个平方数来表示i  
 for (int i = 1; i <= n; i++) {//i从1到n  
 int minn = Integer.MAX\_VALUE; //定义最小个数  
 //f[i]= 1+ sqrt(i) |min| j=1 f[i−j^2]  
 for (int j = 1; j \* j <= i; j++) {//遍历所有小于sqrt(i)的j  
 //找到f[i - j \* j]的最小值  
 minn = Math.min(minn, f[i - j \* j]);  
 }//f[i]=最小值+1  
 f[i] = minn + 1;  
 }  
 return f[n];//返回f[n]  
 }  
}

***//动态规划 n个气球都有数存在nums。戳破所有气球。戳破i个气球可获得nums[i-1]\*nums[i]\*nums[i+1]值。如果i-1或i+1越界就是1。求最大值***public class \_312 {  
 public int maxCoins(int[] nums) {  
 int n = nums.length;//计算数组长度  
 //dp[i][j]表示填满开区间 (i,j) 得到的最多硬币数  
 int[][] rec = new int[n + 2][n + 2];  
 int[] val = new int[n + 2];//扩展原数组两边，各增加一个1  
 val[0] = val[n + 1] = 1;  
 for (int i = 1; i <= n; i++) {//通过nums[i-1]给val[i]赋值  
 val[i] = nums[i - 1];  
 }  
 for (int i = n - 1; i >= 0; i--) {//i从n-1开始  
 //如果i≥j−1代表i到j的区间没有数字，dp[i][j]不合法就是0，因此j从i+2开始  
 for (int j = i + 2; j <= n + 1; j++) {  
 //dp[i][j]= val[i]×val[k]×val[j]+dp[i][k]+dp[k][j]这个表达式的k从i+1到j-1的最大值  
 for (int k = i + 1; k < j; k++) {  
 int sum = val[i] \* val[k] \* val[j];  
 sum += rec[i][k] + rec[k][j];  
 rec[i][j] = Math.max(rec[i][j], sum);  
 }  
 }  
 }  
 return rec[0][n + 1];//返回[0][n-1]就是最大值  
 }  
}

***//动态规划 河流分若干格，每格可能有石子青蛙可跳石子。石子位置列表stones，判断青蛙能否最后一步跳至最后块石子  
//开始青蛙在第一块石子，假定第一步只能跳1单位。如果上一步跳k，接下来只能选k - 1、k 或 k + 1***public class \_403 {  
 public boolean canCross(int[] stones) {  
 int n = stones.length;//求石子数组长度  
 //dp[i][k]表示青蛙能否到石子编号i且上次跳距离k的状态  
 boolean[][] dp = new boolean[n][n];  
 dp[0][0] = true;//第0个石子能到达，为true  
 //遍历判断当第i石子与i−1石子距离超过i时，必定无法到终点，返回false  
 for (int i = 1; i < n; ++i) {  
 if (stones[i] - stones[i - 1] > i) {  
 return false;  
 }  
 }  
 for (int i = 1; i < n; ++i) {//i从1到n  
 //对于第i个石子，枚举所有j（即上次在的石子编号），因此j从i-1到0  
 for (int j = i - 1; j >= 0; --j) {  
 //计算两个石子间的距离k  
 int k = stones[i] - stones[j];  
 if (k > j + 1) { //如果距离大于j+1，肯定没法跳break  
 break;  
 }//从j石头过来跳了k距离，这次可以跳k-1、k或k+1，所以  
 //dp[j][k - 1]、dp[j][k]、dp[j][k + 1]有一个为真即可  
 dp[i][k] = dp[j][k - 1] || dp[j][k] || dp[j][k + 1];  
 //当找到一个dp[n−1][k]为真时，就知道青蛙可以到达终点  
 if (i == n - 1 && dp[i][k]) {  
 return true;  
 }  
 }  
 }  
 return false;//没找到dp[n−1][k]为真，代表无法跳到终点  
 }  
}

***//动态规划 整数数组coins表示不同面额硬币；amount表示总金额。返回可凑成总金额最少币数。如不能返回-1，硬币无限***public class \_322 {  
 public int coinChange(int[] coins, int amount) {  
 int max = amount + 1;  
 int[] dp = new int[amount + 1];//F(i)为组成金额 i 所需最少硬币数  
 Arrays.fill(dp, max);//首先将dp所有元素赋值为mount+1代表非法数  
 dp[0] = 0;//组成0元需要0个硬币  
 for (int i = 1; i <= amount; i++) {//i从1到amount  
 for (int j = 0; j < coins.length; j++) {//遍历所有面值硬币  
 //F(i)=j从0到n-1下式的最小值F(i−c\_j)+1，c\_j代表第 j 枚硬币面值  
 if (coins[j] <= i) {  
 dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);  
 }  
 }  
 }//如果dp[amount]是非法数代表无法组成，返回-1，否则返回dp[amount]  
 return dp[amount] > amount ? -1 : dp[amount];  
 }  
}

***//动态规划 如果二叉树直接相连节点同一天被劫将报警。给二叉树，返回不触警报能盗取最高金额***public class \_337 {  
 //f(o)表示选择o节点下，o节点子树被选择的节点最大权值和；g(o)表示不选择o节点下子树被选择的节点最大权值和  
 Map<TreeNode, Integer> f = new HashMap<TreeNode, Integer>();  
 Map<TreeNode, Integer> g = new HashMap<TreeNode, Integer>();  
  
 public int rob(TreeNode root) {  
 dfs(root);//dfs树 //返回f和g在root对应值的较大值  
 return Math.max(f.getOrDefault(root, 0), g.getOrDefault(root, 0));  
 }  
 public void dfs(TreeNode node) {  
 if (node == null) { return; }//如果节点位空直接返回  
 dfs(node.left);//dfs左右节点  
 dfs(node.right);  
 //节点被选中下子树被选中的最大权值和为 l 和 r 不被选中的最大权值和相加再加当前值，f(o)=g(l)+g(r)+node(o)，默认值0代表空节点  
 f.put(node, node.val + g.getOrDefault(node.left, 0) + g.getOrDefault(node.right, 0));  
 //节点不被选中，左右孩子可以选也可不选，对o某个孩子x，它对o的贡献是x被选或不被选的较大值。g(o)=max{f(l),g(l)}+max{f(r),g(r)}  
 g.put(node, Math.max(f.getOrDefault(node.left, 0), g.getOrDefault(node.left, 0)) +  
 Math.max(f.getOrDefault(node.right, 0), g.getOrDefault(node.right, 0)));  
 }  
}

***//动态规划 只含正整数的非空数组nums。判断是否可将数组分成两个子集，两个子集元素和相等***public class \_416 {  
 public boolean canPartition(int[] nums) {  
 int n = nums.length;//计算数组长度  
 if (n < 2) {//如果长度小于2，返回false  
 return false;  
 }  
 int sum = 0, maxNum = 0;//最大元素maxNum  
 for (int num : nums) {//计算整个数组元素和sum，求maxNum  
 sum += num;  
 maxNum = Math.max(maxNum, num);  
 }//如果sum是奇数，不可能分成元素和相等的两个子集，返回false  
 if (sum % 2 != 0) {  
 return false;  
 }//如果sum是偶数，令target= sum/2  
 int target = sum / 2;  
 if (maxNum > target) {  
 return false;//如果数组中最大数超过target，也不能分割成功  
 }//dp[i][j]表示从数组[0,i]选若干数，是否有一种方案使被选数和等于j  
 boolean[][] dp = new boolean[n][target + 1];  
 for (int i = 0; i < n; i++) {//j为0时肯定是true，即不选任何数  
 dp[i][0] = true;  
 }//从num的0-0下标选数，j只能是nums[0]，因此dp[0][nums[0]]=true  
 dp[0][nums[0]] = true;  
 for (int i = 1; i < n; i++) {//i从1到n  
 int num = nums[i];//获取nums[i]的值  
 for (int j = 1; j <= target; j++) {//j从1到target  
 if (j >= num) {//j≥nums[i]，当前数nums[i]可选可不选，只要一个为true即可  
 dp[i][j] = dp[i - 1][j] | dp[i - 1][j - num];  
 } else {//如果j<nums[i]，则无法选当前数nums[i]，因此dp[i][j]=dp[i−1][j]  
 dp[i][j] = dp[i - 1][j];  
 }  
 }  
 }  
 return dp[n - 1][target];//返回dp[n - 1][target]  
 }  
}

***//动态规划 非负整数数组nums和整数m，将数组分成m个非空连续子数组。使m个子数组各自和的最大值最小***public class \_410 {  
 public int splitArray(int[] nums, int m) {  
 int n = nums.length; //计算数组长度  
 //f[i][j]表示将数组前i个数分为j段能得到的最大连续子数组和最小值  
 int[][] f = new int[n + 1][m + 1];//因此f是[n+1]和[m+1]  
 for (int i = 0; i <= n; i++) { //先把f数组填充成MAX  
 Arrays.fill(f[i], Integer.MAX\_VALUE);  
 } //sub(i+1)表示数组nums下标从0到i内数的和  
 int[] sub = new int[n + 1]; //因此sub大小是n+1，sub[0]是0  
 for (int i = 0; i < n; i++) { //计算sub数字每个位置的值  
 sub[i + 1] = sub[i] + nums[i];  
 }  
 f[0][0] = 0; //0个字符分成0段，和的最小值是0  
 for (int i = 1; i <= n; i++) {//i从1到n  
 //j不能大于i，即长i的数组被分成超过i段是不可能的  
 for (int j = 1; j <= Math.min(i, m); j++) {//因此j从1到i和m的较小值  
 //枚举k，前k个数被分为 j−1 段，第k+1到第i个数为第j段，k从0到i-1  
 //此时j段子数组中和的最大值，就是f[k][j−1] 与 sub(k+1,i)中较大值  
 //再用这个较大值和f[i][j]判断谁更小，就更新f[i][j]  
 for (int k = 0; k < i; k++) {  
 f[i][j] = Math.min(f[i][j], Math.max(f[k][j - 1], sub[i] - sub[k]));  
 }  
 }  
 }  
 return f[n][m];//最后返回f[n][m]  
 }  
}

***//动态规划 字符串表示出勤记录，字符标记当天情况'A'缺勤，'L'迟到，'P'到场。如同时满足：A数<2，最多连2个L则奖励，n表示出勤天。返回能获奖的情况数***public class \_552 {  
 public int checkRecord(int n) {  
 final int MOD = 1000000007;//用来取余数  
 //dp[i][j][k]表示前i天有j个‘A’且结尾连续k个‘L’的奖励出勤记录数，0≤j≤1，0≤k≤2  
 int[][][] dp = new int[n + 1][2][3]; // 长度，A 的数量，结尾连续 L 的数量  
 dp[0][0][0] = 1;//i=0没有出勤记录，‘A’数和结尾连续‘L’数是0，dp[0][0][0]=1  
 //1≤i≤n时，dp[i][][]从dp[i−1][][]转移得到，需要考虑第 i 天出勤记录  
 for (int i = 1; i <= n; i++) {//i从1到n  
 //如果第i天是‘P’，则和前i−1天相比，‘A’数不变，结尾连续‘L’数清零，有  
 //dp[i][j][0] = dp[i][j][0] + 2 ∑ k=0 dp[i−1][j][k]  
 for (int j = 0; j <= 1; j++) {//j从0到1  
 for (int k = 0; k <= 2; k++) {//k从0到2  
 dp[i][j][0] = (dp[i][j][0] + dp[i - 1][j][k]) % MOD;  
 }  
 }  
 //如果第i天是‘A’，则和前i−1天相比‘A’数加1，结尾连续‘L’数清零，要求前i−1天记录中  
 //‘A’数必须为0，，因此有dp[i][1][0] = dp[i][1][0]+ 2 ∑ k=0 dp[i−1][0][k]  
 for (int k = 0; k <= 2; k++) { dp[i][1][0] = (dp[i][1][0] + dp[i - 1][0][k]) % MOD; }  
 //如果第i天是‘L’，则和前i−1天相比‘A’数不变，结尾连续‘L’数加1，要求前i−1天结尾连续‘L’数不超1  
 //因此对0≤j≤1 和 1≤k≤2，有dp[i][j][k] = dp[i][j][k]+dp[i−1][j][k−1]  
 for (int j = 0; j <= 1; j++) {  
 for (int k = 1; k <= 2; k++) {  
 dp[i][j][k] = (dp[i][j][k] + dp[i - 1][j][k - 1]) % MOD;  
 }  
 }  
 }  
 int sum = 0;  
 for (int j = 0; j <= 1; j++) {//j从0到1，k从0到2  
 for (int k = 0; k <= 2; k++) { //将dp[n][j][k]求和返回  
 sum = (sum + dp[n][j][k]) % MOD;  
 }  
 }  
 return sum;  
 }  
}

***//动态规划 整数数组prices，prices[i]表示i天股价；fee是手续费。可无限交易每笔交易(买又卖)付手续费。手里最多持一股票。返回利润最大***public class \_714 {  
 public int maxProfit(int[] prices, int fee) { //同lc的\_122，方程多减一个free  
 int n = prices.length; int[][] dp = new int[n][2]; dp[0][0] = 0; dp[0][1] = -prices[0];  
 for (int i = 1; i < n; ++i) { //这里多减一个fee  
 dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][1] + prices[i] - fee);  
 dp[i][1] = Math.max(dp[i - 1][1], dp[i - 1][0] - prices[i]);  
 }  
 return dp[n - 1][0];  
 }  
}

***//动态规划 车从位置0开始速度为+1，车也可负向行驶。按加速'A'和倒车'R'组的序列行驶。'A'时position+=speed speed\*=2  
//'R'时：如果速度为正，那么speed = -1 否则 speed = 1 位置不变，给目标target，返回到达目标的最短指令序列长度***public class \_818 {  
 public int racecar(int target) {  
 //dp[x] 表示到位置 x 最短指令长度  
 int[] dp = new int[target + 3];  
 Arrays.fill(dp, Integer.MAX\_VALUE);  
 dp[0] = 0;//到位置0指令数为0  
 dp[1] = 1;//到位置1指令数为1，A即可  
 dp[2] = 4;//到位置2指令数为4，AARA:0->1->3->3->2  
 //目标值t从3到target  
 for (int t = 3; t <= target; ++t) {  
 //找到t左边连续0个数，再用32减掉，就是高位1的位置  
 int k = 32 - Integer.numberOfLeadingZeros(t);  
 if (t == (1 << k) - 1) {//如果t恰好是2^k-1，就通过k次A即可到达  
 dp[t] = k;//因此dp[t] = k  
 continue;  
 }//否则t比2^k-1小但肯定大于2^{k-1}，先向右到2^{k-1}，然后R再j次A到2^{k-1}-2^j，到  
 //dp[t - (1 << (k - 1)) + (1 << j)]位置  
 for (int j = 0; j < k - 1; ++j)//因此j从0到k-2遍历求出dp[t-(1 << (k - 1))+(1 << j)]+k-1+j+2最小值  
 dp[t] = Math.min(dp[t], dp[t - (1 << (k - 1)) + (1 << j)] + k - 1 + j + 2);  
 if ((1 << k) - 1 - t < t)//如果2t大于2^{k}-1，使用指令A^kR，长度为 k+1，求dp[(1 << k)-1 - t]+k+1的较小值  
 dp[t] = Math.min(dp[t], dp[(1 << k) - 1 - t] + k + 1);  
 }  
 return dp[target];//返回dp[target]  
 }  
}

***//动态规划 两字符串text1和text2，返回最长公共子序列长。不存在返回0。子序列由原串不变字符相对顺序下删某些字符也可不删除后组成***public class \_1143 {  
 public int longestCommonSubsequence(String text1, String text2) {  
 int m = text1.length(), n = text2.length();//计算两个字符串长度m和n  
 //dp[i][j]表示text1[0:i]和text2[0:j]最长公共子序列长度  
 int[][] dp = new int[m + 1][n + 1];  
 //下标从1开始算，i从1到m，j从1到n  
 for (int i = 1; i <= m; i++) {  
 char c1 = text1.charAt(i - 1);//获取串1的字符c1  
 for (int j = 1; j <= n; j++) {  
 //获取串2字符c2  
 char c2 = text2.charAt(j - 1);  
 if (c1 == c2) {//如果字符相同，从dp[i-1][j-1]值+1得到  
 dp[i][j] = dp[i - 1][j - 1] + 1;  
 } else {//字符不相同，dp[i-1][j]或dp[i][j-1]中的较大值  
 dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);  
 }  
 }  
 }  
 return dp[m][n];//返回dp[m][n]  
 }  
}

# 堆

***//堆 得到数据流中的中位数***public class \_44 {  
 /\* 大顶堆，存储左半边元素 \*/  
 private PriorityQueue<Integer> left = new PriorityQueue<>((o1, o2) -> o2 - o1);  
 /\* 小顶堆，存储右半边元素，并且右半边元素都大于左半边 \*/  
 private PriorityQueue<Integer> right = new PriorityQueue<>();  
 private int N = 0; /\* 当前数据流读入的元素个数 \*/  
 public void Insert(Integer val) {  
 /\* 插入要保证两个堆存于平衡状态 \*/  
 if (N % 2 == 0) {  
 /\* N为偶数插到右半边。右半边要大于左半边，但新插入元素不一定比左半边元素大  
 \* 因此先将元素插入左半边，然后取出堆顶元素即为最大元素，插入右半边 \*/  
 left.add(val); right.add(left.poll());  
 } else { //N为奇数把数字插左半边，先插右半边把堆顶元素弹到左边，保证左都小于右  
 right.add(val);  
 left.add(right.poll());  
 }  
 N++; //增大N  
 }  
 public Double GetMedian() {  
 //如果N是偶数，就返回两个堆顶元素的平均值  
 if (N % 2 == 0) return (left.peek() + right.peek()) / 2.0;  
 //如果是奇数，返回右半边的堆顶元素，因为第一个数插入右半边，所以右边比左边多一个数  
 else return (double) right.peek();  
 }  
}

***//堆 一个整数数组nums和整数k，返回出现频率前k高元素***public class \_347 {  
 public int[] topKFrequent(int[] nums, int k) {  
 Map<Integer, Integer> occurrences = new HashMap<Integer, Integer>();//记录各数字出现次数  
 for (int num : nums) {//num作为key，如果已经有值了就+1，否则default为0  
 occurrences.put(num, occurrences.getOrDefault(num, 0) + 1);  
 }  
 // int[]第一元素代表数值，第二元素代表该值次数，按次数构造小顶堆  
 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()) {  
 //遍历出现次数map，获取数字num及次数count  
 int num = entry.getKey(), count = entry.getValue();  
 if (queue.size() == k) {//如果堆大小达到k  
 if (queue.peek()[1] < count) {//如果栈顶元素次数小于count  
 queue.poll();//就弹出栈顶元素，将新元素放入  
 queue.offer(new int[]{num, count});  
 }  
 } else {//如果堆大小没到k，直接放入元素  
 queue.offer(new int[]{num, count});  
 }  
 }  
 int[] ret = new int[k];//构造返回数组  
 for (int i = 0; i < k; ++i) {//将小顶堆内的num加入结果集合  
 ret[i] = queue.poll()[0];  
 }  
 return ret;  
 }  
}

***//堆 给整数数组nums和整数k，返回数组中第k大的元素***public class \_215 {  
 public int findKthLargest(int[] nums, int k) {  
 int heapSize = nums.length; //计算数组大小  
 buildMaxHeap(nums, heapSize); //用数组构造大顶堆  
 for (int i = nums.length - 1; i >= nums.length - k + 1; --i) {  
 swap(nums, 0, i);//i从最后一个到n-k+1  
 --heapSize; //堆size--  
 maxHeapify(nums, 0, heapSize); //从0到堆size  
 }  
 return nums[0]; //返回Nums[0]  
 }  
  
 public void buildMaxHeap(int[] a, int heapSize) {  
 //i从heapSize/2到0，调整堆  
 for (int i = heapSize / 2; i >= 0; --i) {  
 maxHeapify(a, i, heapSize);  
 }  
 }  
  
 public void maxHeapify(int[] a, int i, int heapSize) {  
 //l是2i+1，r是2i+2，最大下标初始化为i  
 int l = i \* 2 + 1, r = i \* 2 + 2, largest = i;  
 //如果l不越界并且a[l] > a[largest]，更新最大下标为l  
 if (l < heapSize && a[l] > a[largest]) {  
 largest = l;  
 }//如果r不越界并且a[r] > a[largest]，更新最大下标为r  
 if (r < heapSize && a[r] > a[largest]) {  
 largest = r;  
 }  
 if (largest != i) {//如果更新过large  
 swap(a, i, largest);//交换i和large位置元素  
 maxHeapify(a, largest, heapSize);//继续调整large  
 }  
 }  
  
 public void swap(int[] a, int i, int j) {  
 int temp = a[i];  
 a[i] = a[j];  
 a[j] = temp;  
 }  
}

***//堆 返回数组中最小的 K 个数***public class \_43 {  
 public ArrayList<Integer> GetLeastNumbers\_Solution(int[] nums, int k) {  
 if (k > nums.length || k <= 0) //如果k大于数组长度或者k<=0，返回空数组  
 return new ArrayList<>();  
 //否则构造大顶堆，o2-o1就是大顶堆  
 PriorityQueue<Integer> maxHeap = new PriorityQueue<>((o1, o2) -> o2 - o1);  
 for (int num : nums) { //遍历数组，将元素放入大顶堆，如果size超过k就移除堆顶元素(最大元素)  
 maxHeap.add(num);  
 if (maxHeap.size() > k)  
 maxHeap.poll();  
 }  
 return new ArrayList<>(maxHeap); //最后返回大顶堆内的k个元素  
 }  
}

# 二分查找

***//二分查找 求某数字在排序数组中出现的次数***public class \_55 {  
 public int GetNumberOfK(int[] nums, int K) {  
 //因为是有序的，所以用二分查找找到≤K的第一个下标  
 int first = binarySearch(nums, K);  
 //二分查找到≤K+1的第一个下标，这两个下标之差就是K出现的次数  
 int last = binarySearch(nums, K + 1);  
 //如果first是数组长度，代表没有小于k的数，返回0，如果first位置不是K，也返回0，否则返回差值  
 return (first == nums.length || nums[first] != K) ? 0 : last - first;  
 }  
  
 private int binarySearch(int[] nums, int K) {  
 int l = 0, h = nums.length; //初始化l和h  
 while (l < h) { //退出循环条件  
 int m = l + (h - l) / 2; //计算l和h中点  
 if (nums[m] >= K) h = m; //如果该位置数字大于等于k，更新h  
 else l = m + 1; //小于k时更新l，所以最后得到小于k的最大元素下标或第一个K下标  
 }  
 return l;  
 }  
}

***//二分查找 整数数组nums升序值互不同nums在k旋转，如[0,1,2,4,5,6,7]在3旋为[4,5,6,7,0,1,2]，旋转后数组找target下标，不存在返-1***public class \_33 {  
 public int search(int[] nums, int target) {  
 int n = nums.length; //如果数组为空，返回-1  
 if (n == 0) {  
 return -1;  
 }//如果数组只有1个元素，通过比较该值是否为target决定返回下标0还是-1  
 if (n == 1) {  
 return nums[0] == target ? 0 : -1;  
 } //初始化左右指针位置  
 int l = 0, r = n - 1;  
 while (l <= r) { //二分循环  
 int mid = (l + r) / 2; //计算中间位置  
 if (nums[mid] == target) { //如果mid位置元素=target，返回mid下标  
 return mid;  
 } //如果0小于mid位置值，说明0到mid有序  
 if (nums[0] <= nums[mid]) { //如果target在0到mid范围，就要更新r为mid-1  
 if (nums[0] <= target && target < nums[mid]) {  
 r = mid - 1;  
 } else { //否则l是mid+1  
 l = mid + 1;  
 }  
 } else { //如果mid到n-1有序且target在这个范围内，就把l更新为mid+1  
 if (nums[mid] < target && target <= nums[n - 1]) {  
 l = mid + 1;  
 } else { //否则r更新为mid-1  
 r = mid - 1;  
 }  
 }  
 }  
 return -1; //最后如果没找到就返回-1  
 }  
}

***//二分查找 非递减排列整数数组nums和目标值target。找目标值在数组中开始位置和结束位置***public class \_34 {  
 public int[] searchRange(int[] nums, int target) {  
 //找到第一个≥target下标  
 int leftIdx = binarySearch(nums, target, true);  
 //找到第一个大于target下标再-1  
 int rightIdx = binarySearch(nums, target, false) - 1;  
 //如果left下标≤right下标并且right不月结并且left和right位置值确实都是target，就返回left和right  
 if (leftIdx <= rightIdx && rightIdx < nums.length && nums[leftIdx] == target && nums[rightIdx] == target) {  
 return new int[]{leftIdx, rightIdx};  
 } //否则返回-1 -1  
 return new int[]{-1, -1};  
 }  
  
 //二分查找  
 public int binarySearch(int[] nums, int target, boolean lower) {  
 //lower为true，则查找第一个大于等于target下标，否则找第一个大于target的下标  
 int left = 0, right = nums.length - 1, ans = nums.length; //初始化左右  
 while (left <= right) { //二分循环  
 int mid = (left + right) / 2; //计算mid  
 //如果mid值比target大或者lower是true且mid值等于target，要继续找  
 if (nums[mid] > target || (lower && nums[mid] >= target)) {  
 right = mid - 1; //更新right为mid-1  
 ans = mid; //下标更新为mid  
 } else { //否则更新left为mid+1  
 left = mid + 1;  
 }  
 }  
 return ans; //返回找到的下标  
 }  
}

***//二分查找 排序数组和目标值，在数组中找目标值索引。如果不存在，返回它将被按序插入位置***public class \_35 {  
 public int searchInsert(int[] nums, int target) {  
 int n = nums.length;  
 int left = 0, right = n - 1, ans = n; //初始化左右下标  
 while (left <= right) { //二分循环  
 int mid = ((right - left) >> 1) + left; //计算mid，和(right + left) >> 1一样  
 if (target <= nums[mid]) { //如果mid值大于等于target，更新右边界且ans=mid  
 ans = mid;  
 right = mid - 1;  
 } else { //否则更新左边界  
 left = mid + 1;  
 }  
 }  
 return ans; //返回找到的下标或者需要插入的下标  
 }  
}

***//二分查找 非负整数x ，返回x算术平方根 。结果保留整数部分***public class \_69 {  
 public int mySqrt(int x) {  
 int l = 0, r = x, ans = -1; //二分从0找到x  
 while (l <= r) { //二分循环判断条件  
 int mid = l + (r - l) / 2; //求mid  
 if ((long) mid \* mid <= x) { //如果mid平方≤x，更新ans和l  
 ans = mid;  
 l = mid + 1;  
 } else { //否则更新r  
 r = mid - 1;  
 }  
 }  
 return ans; //最后返回ans  
 }  
}

***//二分查找 长度n的元素值互不相同数组预先升序，经1到n次旋转后得到输入数组nums。找出数组中最小元素***public class \_153 {  
 public int findMin(int[] nums) {  
 int low = 0; //标准二分查找，直到low和high相遇  
 int high = nums.length - 1;  
 while (low < high) {  
 int pivot = low + (high - low) / 2;  
 if (nums[pivot] < nums[high]) { high = pivot; }  
 else { low = pivot + 1; }   
 }  
 return nums[low];  
 }  
}

***//二分查找 判断mxn矩阵是否存在目标值。矩阵每行从左到右升序。每行第一个整数大于前一行最后整数***public class \_74 {  
 public boolean searchMatrix(int[][] matrix, int target) {  
 int rowIndex = binarySearchFirstColumn(matrix, target);  
 if (rowIndex < 0) { //如果行号小于0，说明左上角元素都比target大，返回false  
 return false;  
 }  
 return binarySearchRow(matrix[rowIndex], target); //否则再二分查找这一行  
 }  
 //二分查找第一列元素小于等于target的最大值  
 public int binarySearchFirstColumn(int[][] matrix, int target) {  
 //低和高边界分别是0和m-1，因为有m行，所以一列有m个元素，就是matrix.length-1  
 int low = -1, high = matrix.length - 1;  
 while (low < high) { //二分循环  
 int mid = (high - low + 1) / 2 + low; //计算mid  
 //因为找第一列，所以matrix[mid][0]  
 if (matrix[mid][0] <= target) {  
 low = mid; //如果mid[0]小于等于target，更新low  
 } else { //否则更新high  
 high = mid - 1;  
 }  
 }  
 return low; //返回找到的行号  
 }  
 //标准的二分查找  
 public boolean binarySearchRow(int[] row, int target) {  
 int low = 0, high = row.length - 1;  
 while (low <= high) {  
 int mid = (high - low) / 2 + low;  
 if (row[mid] == target) {  
 return true;  
 } else if (row[mid] > target) {  
 high = mid - 1;  
 } else {  
 low = mid + 1;  
 }  
 }  
 return false;  
 }  
}

***//二分查找 n个元素升序数组nums和目标值target，搜索nums中的target下标，不存在就返回-1***public class \_704 {  
 //标准二分查找  
 public int search(int[] nums, int target) {  
 int left = 0, right = nums.length - 1;//得到l和r  
 while (left <= right) {//循环判断  
 int mid = (right - left) / 2 + left;//计算mid  
 int num = nums[mid];//获取mid处数值  
 if (num == target) {//如果等于target，返回下标  
 return mid;  
 } else if (num > target) {//如果mid处值大于target，更新r=mid-1  
 right = mid - 1;  
 } else {//否则l=mid+1  
 left = mid + 1;  
 }  
 }  
 return -1;//循环结束没找到就返回-1  
 }  
}

# 滑动窗口

***//滑动窗口 输出所有和为 S 的连续正数序列***public class \_61 {  
 public ArrayList<ArrayList<Integer>> FindContinuousSequence(int sum) {  
 ArrayList<ArrayList<Integer>> ret = new ArrayList<>(); //存放结果  
 int start = 1, end = 2; //起始和结束值  
 int curSum = 3; //当前的连续数和  
 while (end < sum) { //循环，end是结束元素，start是起始元素  
 if (curSum > sum) { //如果当前连续序列和大于预期，就减掉一个起始值，把起始值右移一位  
 curSum -= start;  
 start++;  
 } else if (curSum < sum) { //如果当前连续序列和小于预期，就增加一个终止值，把终止值右移一位  
 end++;  
 curSum += end;  
 } else { //如果当前连续序列和等于预期，就说明找到了一个连续数组，将该数组放入结果集  
 ArrayList<Integer> list = new ArrayList<>();  
 for (int i = start; i <= end; i++)  
 list.add(i);  
 ret.add(list);  
 curSum -= start; start++; end++; curSum += end; //将数组整体右移一个数  
 }  
 }  
 return ret;  
 }  
}

***//滑动窗口 字符串s和一些长度相同单词words。找出s恰好由words中所有单词串联形成子串起始位置。不考虑单词顺序***public class \_30 {  
 public List<Integer> findSubstring(String s, String[] words) {  
 List<Integer> res = new ArrayList<Integer>(); //存放结果数组  
 //m是单词个数，n是每个单词长度，ls是字符串长度  
 int m = words.length, n = words[0].length(), ls = s.length();  
 //将s分为单词组，每个单词大小均n。划分方法有n种，即先删去前i（i=0∼n−1）个字母，将剩下字母划分  
 //因为删去前n和字母和删0个字母对后面的单词来说效果一样，相当于少得到一个单词  
 for (int i = 0; i < n; i++) { //所以i从0到n-1 对于n种划分得到的所有窗口列表遍历  
 if (i + m \* n > ls) { //如果i+mn超过ls退出  
 break;  
 } //differ表示窗口中单词频次和words中单词频次之差  
 Map<String, Integer> differ = new HashMap<String, Integer>();  
 for (int j = 0; j < m; j++) { //一个窗口包含s中前m个单词，因此j从0到m  
 //依次取出m个长n的字符串(单词)  
 String word = s.substring(i + j \* n, i + (j + 1) \* n);  
 //将单词放入differ，记录各个单词在当前窗口出现的次数  
 differ.put(word, differ.getOrDefault(word, 0) + 1);  
 } //遍历words中的单词，将其加入differ  
 for (String word : words) {  
 //differ中某个单词的value为正，代表当前子串多了这个单词，  
 //为负代表子串还缺少words中这个单词，为0代表正好匹配了words中这个单词个数  
 //如果differ所有key的value都为0，代表找到了一条子串由words单词构成  
 differ.put(word, differ.getOrDefault(word, 0) - 1);  
 if (differ.get(word) == 0) {  
 differ.remove(word);  
 }  
 }//移动窗口，start代表窗口起始点，每次start+n(即右移一个单词)  
 for (int start = i; start < ls - m \* n + 1; start += n) {  
 if (start != i) { //start!=i代表移动过窗口，右侧加个单词，左侧出个单词，要更新differ  
 //进来的单词是 start+(m-1)\*n 到 start+m\*n  
 String word = s.substring(start + (m - 1) \* n, start + m \* n);  
 //进来一个单词，要给word的value+1  
 differ.put(word, differ.getOrDefault(word, 0) + 1);  
 if (differ.get(word) == 0) { //如果该单词次数为0，将其移除  
 differ.remove(word);  
 } //出去的单词是start-n 到 start  
 word = s.substring(start - n, start);  
 //出去的单词，要给word的value-1  
 differ.put(word, differ.getOrDefault(word, 0) - 1);  
 if (differ.get(word) == 0) { //如果该单词次数为0，将其移除  
 differ.remove(word);  
 }  
 } //每移动一次窗口判断differ是否为空，如果空就把start加入res，因为从该位置开始的子串符合要求  
 if (differ.isEmpty()) {  
 res.add(start);  
 }  
 }  
 }  
 return res;//最后返回结果  
 }  
}

***//滑动窗口 给一个数组和窗口大小，找出所有滑动窗口里数值最大值***public class \_64 {  
 public ArrayList<Integer> maxInWindows(int[] num, int size) {  
 ArrayList<Integer> ret = new ArrayList<>(); //存放结果数组  
 //如果滑动窗口超出数组长度或者小于等于1，直接返回原数组  
 if (size > num.length || size < 1) return ret;  
 //构建一个大顶堆 o2-o1是大顶堆  
 PriorityQueue<Integer> heap = new PriorityQueue<>((o1, o2) -> o2 - o1);  
 //将最开始的size个数放入堆，取出堆顶元素放入结果集  
 for (int i = 0; i < size; i++)  
 heap.add(num[i]);  
 ret.add(heap.peek());  
 //以此平移窗口，每次向右移动一位，更新大顶堆元素，将堆顶元素放入结果集  
 for (int i = 0, j = i + size; j < num.length; i++, j++) {  
 heap.remove(num[i]); //滑动窗口，移除i元素，加入j元素，将大顶堆堆顶元素放入结果集  
 heap.add(num[j]);  
 ret.add(heap.peek());  
 }  
 return ret;  
 }  
}

***//滑动变长窗口 字符串s和t。返回s中涵盖t所有字符的最小子串。如果s中不存在则返回空字符串***public class \_76 {  
 Map<Character, Integer> ori = new HashMap<Character, Integer>();//保存t各字符出现次数  
 Map<Character, Integer> cnt = new HashMap<Character, Integer>(); //保存当前窗口各字符出现次数  
  
 public String minWindow(String s, String t) {  
 int tLen = t.length(); //得到字符串t长度  
 for (int i = 0; i < tLen; i++) { //获取t各字符次数  
 char c = t.charAt(i);  
 ori.put(c, ori.getOrDefault(c, 0) + 1);  
 }  
 int l = 0, r = -1; //用于延伸现有窗口的r指针，和用于收缩窗口的l指针  
 //len保存当前满足条件结果子串最短长度，ansL为最终子串左下标，ansR为最终子串右下标  
 int len = Integer.MAX\_VALUE, ansL = -1, ansR = -1;  
 int sLen = s.length(); //计算s字符串长度  
 while (r < sLen) { //循环，判断右指针r不越界  
 ++r; //右指针++，让r从0开始找字符直到r位置  
 if (r < sLen && ori.containsKey(s.charAt(r))) { //如果在t中有s[r]字符  
 //就把cnt中s[r]字符出现的次数++  
 cnt.put(s.charAt(r), cnt.getOrDefault(s.charAt(r), 0) + 1);  
 }  
 //r滑动一位，就判断当前子串是否包含t的全部字符，不包含就继续++r更新cnt  
 while (check() && l <= r) { //循环判断当前窗口是否包含t全部字符且l<=r  
 //如果当前左右指针包含的字符串长度小于len，更新len、ansL、ansR，先更新结果  
 if (r - l + 1 < len) {  
 len = r - l + 1;  
 ansL = l;  
 ansR = l + len;  
 }  
 //如果t包含s[l]字符，就将窗口中该字符次数-1，因为接下来要l++把这个字符踢出窗口了  
 if (ori.containsKey(s.charAt(l))) {  
 cnt.put(s.charAt(l), cnt.getOrDefault(s.charAt(l), 0) - 1);  
 }  
 ++l; //踢出l字符继续判断窗口是否满足条件  
 }  
 } //当右指针遍历完的时候，len就达到了最短，返回s.substring(ansL, ansR)  
 return ansL == -1 ? "" : s.substring(ansL, ansR);  
 }  
  
 public boolean check() {  
 Iterator iter = ori.entrySet().iterator();  
 while (iter.hasNext()) { //遍历ori每个元素  
 //获取字符串t每个字符及出现次数  
 Map.Entry entry = (Map.Entry) iter.next();  
 Character key = (Character) entry.getKey();  
 Integer val = (Integer) entry.getValue();  
 //如果窗口内字符串的字符出现次数小于t的字符次数，返回false  
 if (cnt.getOrDefault(key, 0) < val) {  
 return false;  
 }  
 }  
 return true; //返回true代表当前窗口字符串能找到t的全部字符甚至多出部分字符  
 }  
}

***//滑动窗口 两个字符串s和p，找到s中所有p的异位词子串，返回子串起始索引列表***public class \_438 {  
 public List<Integer> findAnagrams(String s, String p) {  
 int sLen = s.length(), pLen = p.length();//获取s和p的长度  
 if (sLen < pLen) {//如果s比p短，肯定找不到，返回空数组  
 return new ArrayList<Integer>();  
 }  
 List<Integer> ans = new ArrayList<Integer>();//建立结果数组  
 int[] sCount = new int[26];//构造s和p中各字符出现次数表  
 int[] pCount = new int[26];  
 for (int i = 0; i < pLen; ++i) {  
 ++sCount[s.charAt(i) - 'a'];//计算sCount和pCount各位值  
 ++pCount[p.charAt(i) - 'a'];  
 }//如果s和p所有字符出现次数都相同，ans中添加0，说明从s起始下标就可匹配  
 if (Arrays.equals(sCount, pCount)) {  
 ans.add(0);  
 }//i从0到sl-pl，如果s剩余的字符数少于p的长度，肯定没法匹配异位词  
 for (int i = 0; i < sLen - pLen; ++i) {  
 //每移动一步，将s该位置字符次数-1，窗口右侧字符次数+1，更新sCount  
 --sCount[s.charAt(i) - 'a'];  
 ++sCount[s.charAt(i + pLen) - 'a'];  
 //如果sCount和pCount相等，将i+1下标加入结果集  
 if (Arrays.equals(sCount, pCount)) {  
 ans.add(i + 1);  
 }  
 }  
 return ans;//返回结果集  
 }  
}

# 回文串

***//回文串 一个字符串，验证是否是回文，只考虑字母和数字，忽略字母大小写***public class \_125 {  
 public boolean isPalindrome(String s) {  
 int n = s.length(); //获取字符串长度n  
 int left = 0, right = n - 1; //左右指针  
 while (left < right) { //循环  
 while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {  
 ++left; //从左遍历找到字符或数字字符  
 }  
 while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {  
 --right; //从右边找到字符或数字字符  
 }  
 if (left < right) {  
 if (Character.toLowerCase(s.charAt(left)) != Character.toLowerCase(s.charAt(right))) {  
 return false; //如果两个字符不相同，返回false  
 } //否则同时更新左右指针  
 ++left; --right;  
 }  
 }  
 return true; //如果遍历退出，说明是回文串，返回true  
 }  
}

***//回文串 给一个包含大小写字母字符串s，返回通过这些字母构造成的最长回文串***public class \_409 {  
 public int longestPalindrome(String s) {  
 //统计每个字符出现多少次，其实用不了128，128比较方便  
 int[] count = new int[128];  
 int length = s.length();//获取字符串长度  
 for (int i = 0; i < length; ++i) {  
 char c = s.charAt(i);//遍历字符串每个字符  
 count[c]++;//字符的asc码处值++  
 }  
 int ans = 0;//最长回文串长度  
 for (int v : count) {//遍历count数组  
 ans += v / 2 \* 2; // v/2\*2就是把所有双数字符用来构成回文  
 //针对剩余的所有奇数个字符，最多只能再加上一个，  
 if (v % 2 == 1 && ans % 2 == 0) {  
 ans++;//因此如果v是奇数并且ans是偶数，就给ans++  
 }  
 }  
 return ans;//返回ans  
 }  
}

***//回文串 判断整数x是否位回文整数***public class \_9 {  
 //应该用双指针指向首尾，不断判断字符是否相等并靠近指针  
 public boolean isPalindrome(int x) {  
 String reversedStr = (new StringBuilder(x + "")).reverse().toString();  
 return (x + "").equals(reversedStr);  
 }  
}

***//回文串 字符串s，统计其中回文子串数(注意子串连续)。有不同开始结束位置的内容相同子串被视作不同子串***public class \_647 {  
 public int countSubstrings(String s) {  
 int n = s.length(), ans = 0;//计算原字符串长度n，初始化ans=0  
 for (int i = 0; i < 2 \* n - 1; ++i) {//长n字符串有2n-1个回文中心[li,ri]  
 int l = i / 2, r = i / 2 + i % 2;//li=⌊i/2⌋，ri=li+(i%2)  
 //以[li,ri]为中心向外依次判断是否有回文子串，l和r不越界，如果l和r字符相等就扩l和r，ans++  
 //直到越界或者l和r处字符不相等退出循环，继续找下一个回文中心进行扩展  
 while (l >= 0 && r < n && s.charAt(l) == s.charAt(r)) {  
 --l; ++r; ++ans;  
 }  
 }  
 return ans;//返回ans  
 }  
}

# 贪心

***//贪心 非负整数数组nums，最初位于第一位置。每个元素代表该位置可跳最大长度。求最少跳跃次数到数组最后位置***public class \_45 {  
 public int jump(int[] nums) {  
 //记录当前反向查找已经到达的位置  
 int position = nums.length - 1;  
 int steps = 0;  
 while (position > 0) { //如果还没到第0个位置就循环找  
 //从第0个位置开始找能到达position且离position最远的位置  
 for (int i = 0; i < position; i++) {  
 //如果找到了就把position = i，steps++，跳出内循环  
 if (i + nums[i] >= position) {  
 position = i;  
 steps++;  
 break;  
 }  
 }  
 }  
 return steps; //最后返回step  
 }  
}

***//贪心 每个孩子一块饼干。孩子i胃口值g[i]，饼干j尺寸s[j]。如果s[j]>=g[i]孩子满足。输出能满足孩子的最大值***public class \_455 {  
 public int findContentChildren(int[] g, int[] s) {  
 Arrays.sort(g);//孩子胃口升序排序  
 Arrays.sort(s);//饼干大小升序排序  
 int numOfChildren = g.length, numOfCookies = s.length;//计算孩子数和饼干数  
 int count = 0;//保存满足的最多孩子数  
 //i从0到孩子数，j从0到饼干数遍历，注意外层循环j每次不一定比上一次大1，因为内层也会j++  
 //如果外层循环j跳跃了多个，说明部分小饼干没法用，不能满足胃口最小的孩子被跳过了  
 for (int i = 0, j = 0; i < numOfChildren && j < numOfCookies; i++, j++) {  
 //如果j小于饼干数且孩子胃口大于饼干，就j++找下一块更大饼干  
 while (j < numOfCookies && g[i] > s[j]) {  
 j++;  
 }//如果j小于饼干数，说明找到了满足g[i]胃口最小的饼干就count++  
 if (j < numOfCookies) {  
 count++;  
 }  
 }  
 return count;  
 }  
}

***//贪心 一个非空字符串s，最多删一个字符判断是否能成回文字符串***public class \_680 {  
 public boolean validPalindrome(String s) {  
 int low = 0, high = s.length() - 1;//初始化低位和高位指针  
 while (low < high) {//循环  
 char c1 = s.charAt(low), c2 = s.charAt(high);//判断l和h处字符如果相等，就靠近l和h继续判断  
 if (c1 == c2) {  
 ++low;  
 --high;  
 } else {//字符如果不相等，可以删除左或右字符，判断l到h-1或者l+1到h是否有一个是回文串  
 return validPalindrome(s, low, high - 1) || validPalindrome(s, low + 1, high);  
 }  
 }  
 return true;  
 }  
 //判断s从low到high的内容是否是回文串  
 public boolean validPalindrome(String s, int low, int high) {  
 //不断移动l和h，只要两个位置字符串不相等，就返回false，相等就靠近  
 for (int i = low, j = high; i < j; ++i, --j) {  
 char c1 = s.charAt(i), c2 = s.charAt(j);  
 if (c1 != c2) {  
 return false;  
 }  
 }  
 return true;//便利完毕都能匹配，返回true  
 }  
}

***//贪心 每杯水5。顾客一次买一杯付5、10或20。数组bills，bills[i]是第i顾客付的账，判断能否给每个顾客正确找零，开始没有零钱***public class \_860 {  
 public boolean lemonadeChange(int[] bills) {  
 int five = 0, ten = 0;//变量five和ten表示手中的5元和10元钞票数  
 for (int bill : bills) {//遍历每次付款  
 if (bill == 5) {//如果付5，不找零，且five++  
 five++;  
 } else if (bill == 10) {//如果付10  
 if (five == 0) {//如果当前没有5原，返回false  
 return false;  
 }//否则five--，ten++  
 five--; ten++;  
 } else {//如果付20  
 if (five > 0 && ten > 0) {//如果同时有5和10，就找零5和10  
 five--; ten--;  
 } else if (five >= 3) {//否则如果有不少于3张5，找零3个5  
 five -= 3;  
 } else {//否则返回false  
 return false;  
 }  
 }  
 }  
 return true;//遍历结束返回true  
 }  
}