第二版前言

相比于2018年第一版，第二版产生了以下更新。

- 编程语言由C++全部替换成Java。
- 对动态规划部分题目进行了大量更新。
- 对知识点进行更细致的划分，基本上覆盖了常见的算法刷题套路。
- 增加了设计篇与模版篇。

说明

- 题解大部分由笔者所写，也有部分参考了各大OJ和书籍的题解。
- 题目分类与编排为笔者完全原创，引用需要声明来源。
- 题目只有题解，几乎没有解析，含解析在加班加点制作中。
- 题目绝大多数来源Leetcode，每个类命名方式为下划线+题号+标题。
- 题目题解遵循这样一个原则：尽量精简但不牺牲可读性；尽量高效但不牺牲可读性。所以并非所有题目都是最优解。
- 作者水平有限，错误疏漏在所难免，请谅解。

- 如果有侵权，请与作者联系。联系方式为邮箱969023014@qq.com。
- 原创不易，你的star是对作者最大的鼓舞。

参考资料

参考了如下视频课程（带*为特别鸣谢）

- 左程云《算法进阶课》
- liuyubobobo*《玩转数据结构》《图论算法精讲》《玩转算法面试》《看得见的算法》
- 小象学院 《BAT Leetcode算法面试》
- 九章算法

参考了如下博客与书籍

- labuladong的算法小抄*
- Leetcode与牛客网题解
- 左程云《程序员代码面试指南》
- 王道《计算机考研上机指南》
- 《算法导论》
- 《算法 第四版》

还有一些参考性较低的资料没有列出，也一并感谢。

题解

1.数据结构篇

1.1栈

1.1.1 计算器类

> 实现一个基本的计算器来计算一个简单的字符串表达式的值。
> 字符串表达式可以包含左括号 （ ，右括号 ），加号 + ，减号 -，非负整数和空格。

```
public class _224基本计算器 {
    public int calculate(String s) {
        Stack<Integer> stack = new Stack<Integer>();
        int operand = 0;
        int result = 0;
        int sign = 1;
        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (Character.isDigit(ch)) {
                operand = 10 * operand + (int) (ch - '0');
            } else if (ch == '+') {
                result += sign * operand;
                sign = 1;
                operand = 0;
            } else if (ch == '-') {
                result += sign * operand;
                sign = -1;
                operand = 0;
            } else if (ch == '(') {
                stack.push(result);
                stack.push(sign);
                sign = 1;
                result = 0;
            } else if (ch == ')') {
                result += sign * operand;
                result *= stack.pop();
                result += stack.pop();
                operand = 0;
            }
        }
        return result + (sign * operand);
    }
}
```

实现一个基本的计算器来计算一个简单的字符串表达式的值。
字符串表达式仅包含非负整数，+， - ，*，/ 四种运算符和空格。
整数除法仅保留整数部分。

```
public class _227基本计算器II {
    public int calculate(String s) {
        Stack<Integer> stack = new Stack<>();
        int num = 0;
        char sign = '+';
        for(int i = 0; i < s.length(); i++){
            char c = s.charAt(i);
            if(Character.isDigit(c))
                num = num * 10 + (c - '0');
            if((!Character.isDigit(c) && c != ' ')|| i == s.length() -
1){
                int pre;
                switch(sign){
                    case '+':
                        stack.push(num);
                        break;
                    case '-':
                        stack.push(-num);
                        break;
                    case '*':
                        pre = stack.pop();
                        stack.push(pre*num);
                        break;
                    case '/':
                        pre = stack.pop();
                        stack.push(pre/num);
                        break;
                }
                sign = c;
                num = 0;
            }
        }
        int res = 0;
        while(!stack.isEmpty())
            res += stack.pop();
        return res;
    }
}
```


1.1.2 单调栈

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。
求在该柱状图中，能够勾勒出来的矩形的最大面积。

```java
public class _84柱状图中最大的矩形 {
    public int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        int n = heights.length;
        int res = 0;
        for(int i=0;i<=n;i++){
            while(!stack.isEmpty() && (i==n || heights[i] <
heights[stack.peek()])){
                int height = heights[stack.pop()];
                int width = stack.isEmpty() ? i : i - 1 -
stack.peek();
                res = Math.max(res, width*height);
            }
            stack.push(i);
        }
        return res;
    }
}
```

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

```java
public class _85最大矩形 {
    public int maximalRectangle(char[][] matrix) {
        int m = matrix.length;
        if(m == 0)
            return 0;
        int n = matrix[0].length;
        int[] height = new int[n];
        int ret = 0;
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                height[j] = matrix[i][j] == '0' ? 0 : height[j] + 1;
            }
            ret = Math.max(ret, maxArea(height));
        }
        return ret;
    }
    private int maxArea(int[] height) {
        int ret = 0;
        Stack<Integer> stack = new Stack<>();
        for(int i = 0; i <= height.length; i++){
            while(!stack.isEmpty() && (i == height.length || height[i]
<= height[stack.peek()])){
                int h = stack.pop();
                int j = stack.isEmpty() ? -1 : stack.peek();
                ret = Math.max(ret, (i-j-1)*height[h]);
            }
            stack.push(i);
```

```
        }
        return ret;
    }
}
```

给定一个循环数组（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。
数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。
如果不存在，则输出 -1。

```
public class _503下一个更大元素II {
    public int[] nextGreaterElements(int[] nums) {
        int n = nums.length;
        int[] ret = new int[n];
        Stack<Integer> stack = new Stack<>();
        for(int i = 2*n-1; i >= 0; i--){
            while(!stack.isEmpty() && stack.peek() <= nums[i%n])
                stack.pop();
            ret[i%n] = stack.isEmpty() ? -1 : stack.peek();
            stack.push(nums[i%n]);
        }
        return ret;
    }
}
```

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

```
class _42接雨水 {
    public int trap(int[] height) {
        Stack<Integer> stack = new Stack<>();
        int ans = 0;
        for (int i = 0; i < height.length; i++) {
            while(!stack.isEmpty() && height[stack.peek()] <=
height[i]) {
                int curIdx = stack.pop();
                if (!stack.isEmpty()) {
                    int stackTop = stack.peek();
                    ans += (Math.min(height[stackTop], height[i]) -
height[curIdx]) * (i - stackTop - 1);
                }
            }
            stack.add(i);
        }
        return ans;
```

```
    }
}
```

1.1.3 最小栈

设计一个支持 push ，pop ，top 操作，并能在常数时间内检索到最小元素的栈。

push(x) —— 将元素 x 推入栈中。
pop() —— 删除栈顶的元素。
top() —— 获取栈顶元素。
getMin() —— 检索栈中的最小元素。

```java
public class _155最小栈 {
    private Stack<Integer> data;
    private Stack<Integer> min;
    public _155最小栈() {
        data = new Stack<>();
        min = new Stack<>();
    }
    public void push(int x) {
        data.push(x);
        if(min.isEmpty() || min.peek() > x)
            min.push(x);
        else
            min.push(min.peek());
    }
    public void pop() {
        data.pop();
        min.pop();
    }
    public int top() {
        return data.peek();
    }
    public int getMin() {
        return min.peek();
    }
}
```


1.1.4 最长有效括号

给定一个只包含 '(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度。

```java
class _32最长有效括号 {
    public int longestValidParentheses(String s) {
        Stack<Integer> stack = new Stack<>();
        stack.push(-1);
```

```
        int max = 0;
        for(int i = 0; i < s.length(); i++){
            if(s.charAt(i) == '('){
                stack.push(i);
            }else{
                stack.pop();
                if(stack.isEmpty()){
                    stack.push(i);
                }else{
                    max = Math.max(max, i - stack.peek());
                }
            }
        }
        return max;
    }
}
```

1.1.5 移除K个数字

给定一个以字符串表示的非负整数 num，移除这个数中的 k 位数字，使得剩下的数字最小。

注意:
num 的长度小于 10002 且 ≥ k。
num 不会包含任何前导零。

```
class _402移掉K位数字 {
    public String removeKdigits(String num, int k) {
        Stack<Integer> stack = new Stack<>();
        for (int i = 0; i < num.length(); i++) {
            int n = num.charAt(i) - '0';
            while (!stack.isEmpty() && stack.peek() > n && k > 0){
                stack.pop();
                k--;
            }
            if (!stack.isEmpty() || n != 0)
                stack.push(n);
        }
        while (!stack.isEmpty() && k > 0){
            stack.pop();
            k--;
        }
        StringBuilder sb = new StringBuilder();
        for (int i : stack)
            sb.append(i);
        return sb.length() == 0 ? "0" : sb.toString();
    }
}
```

1.1.6 最大宽度坡

> 给定一个整数数组 A，坡是元组 (i, j)，其中 i < j 且 A[i] <= A[j]。这样的坡的宽度为 j - i。
> 找出 A 中的坡的最大宽度，如果不存在，返回 0 。

```
class _962最大宽度坡 {
    public int maxWidthRamp(int[] A) {
        Stack<Integer> stack = new Stack<>();
        for(int i=0;i<A.length;i++){
            if(!stack.isEmpty() && A[i] >= A[stack.peek()])
                continue;
            stack.push(i);
        }
        System.out.println(stack);
        int res = 0;
        for(int i=A.length-1;i>res;i--){
            while(!stack.isEmpty() && A[i]>=A[stack.peek()]){
                res = Math.max(res, i-stack.pop());
            }
        }
        return res;
    }
}
```

变式题

> 给你一份工作时间表 hours，上面记录着某一位员工每天的工作小时数。
> 我们认为当员工一天中的工作小时数大于 8 小时的时候，那么这一天就是「劳累的一天」。
> 所谓「表现良好的时间段」，意味在这段时间内，「劳累的天数」是严格 大于「不劳累的天数」。
> 请你返回「表现良好时间段」的最大长度。

```
class _1124表现良好的最长时间段 {
    public int longestWPI(int[] hours) {
        for(int i = 0;i < hours.length;i++)
            hours[i] = hours[i] > 8 ? 1 : -1;
        int[] sum = new int[hours.length+1];
        for(int i = 0;i < hours.length;i++)
            sum[i+1] = sum[i] + hours[i];
        Stack<Integer> stack = new Stack<>();
        for(int i = 0;i < sum.length;i++){
            if(stack.isEmpty() || sum[i] < sum[stack.peek()])
                stack.push(i);
        }
        int res = 0;
        for(int i = sum.length-1;i > res; i--){
            while(!stack.isEmpty() && sum[i] > sum[stack.peek()]){
```

```
                res = Math.max(res, i - stack.pop());
            }
        }
        return res;
    }
}
```

1.2队列

1.2.1 单调队列

给定一个数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
返回滑动窗口中的最大值。

```
public class _239滑动窗口最大值 {
    public int[] maxSlidingWindow(int[] nums, int k) {
        Deque<Integer> queue = new ArrayDeque<>();
        int[] ret = new int[nums.length-k+1];
        int index = 0;
        for(int i = 0; i < nums.length; i++){
            if(i >= k){
                if(nums[i-k] == queue.peekFirst())
                    queue.removeFirst();
            }
            while(!queue.isEmpty() && queue.peekLast() < nums[i])
                queue.removeLast();
            queue.addLast(nums[i]);
            if(i >= k-1)
                ret[index++] = queue.peekFirst();
        }
        return ret;
    }
    //存储下标
    public int[] maxSlidingWindow2(int[] nums, int k) {
        Deque<Integer> queue = new ArrayDeque<>();
        int[] ret = new int[nums.length-k+1];
        for(int i = 0; i < nums.length; i++){
            if(i >= k && i-k == queue.peekFirst())
                queue.removeFirst();
            while(!queue.isEmpty() && nums[queue.peekLast()] <=
nums[i])
                queue.removeLast();
            queue.addLast(i);
            if(i >= k-1)
                ret[i-k+1] = nums[queue.peekFirst()];
        }
        return ret;
```

```
    }
}
```

1.2.2 优先队列

给一非空的单词列表，返回前 k 个出现次数最多的单词。

返回的答案应该按单词出现频率由高到低排序。如果不同的单词有相同出现频率，按字母顺序排序。

```
class _692前K个高频单词 {
    public List<String> topKFrequent(String[] words, int k) {
        List<String> ret = new ArrayList<>(k);
        Map<String,Integer> map = new HashMap<>();
        for(String s : words){
            map.put(s, map.getOrDefault(s, 0)+1);
        }
        Queue<String> queue = new PriorityQueue<>((w1,w2)->
map.get(w1).equals(map.get(w2)) ? w2.compareTo(w1) : map.get(w1) -
map.get(w2));
        for(String s : map.keySet()){
            queue.offer(s);
            if(queue.size() > k)
                queue.poll();
        }
        while (!queue.isEmpty())
            ret.add(queue.poll());
        Collections.reverse(ret);
        return ret;
    }
}
```

给你一个 m x n 的矩阵，其中的值均为非负整数，代表二维高度图每个单元的高度，请计算图中形状最多能接多少体积的雨水。

```
class _407接雨水II {
    public int trapRainWater(int[][] heightMap) {
        Queue<int[]> queue = new PriorityQueue<>
(Comparator.comparingInt(o -> o[2]));
        int m = heightMap.length;
        int n = heightMap[0].length;
        boolean[][] visited = new boolean[m][n];
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(i==0 || i==m-1 || j==0 || j==n-1){
                    queue.offer(new int[]{i,j,heightMap[i][j]});
                    visited[i][j] = true;
```

```
                }
            }
        }
        int res = 0;
        int[][] dir = {{0,1},{0,-1},{1,0},{-1,0}};
        while(!queue.isEmpty()){
            int[] cur = queue.poll();
            for(int i=0;i<4;i++){
                int nx = cur[0] + dir[i][0];
                int ny = cur[1] + dir[i][1];
                if(0 <= nx && nx < m && 0 <= ny && ny < n &&
!visited[nx][ny]){
                    res += Math.max(cur[2],heightMap[nx][ny])-
heightMap[nx][ny];
                    queue.offer(new int[]
{nx,ny,Math.max(cur[2],heightMap[nx][ny])});
                    visited[nx][ny] = true;
                }
            }
        }
        return res;
    }
}
```

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

```
public class _295数据流的中位数 {
    private int count;
    private PriorityQueue<Integer> maxheap;
    private PriorityQueue<Integer> minheap;
    public MedianFinder() {
        count = 0;
        maxheap = new PriorityQueue<>((x, y) -> y - x);
        minheap = new PriorityQueue<>();
    }
    public void addNum(int num) {
        count += 1;
        maxheap.offer(num);
        minheap.add(maxheap.poll());
        if ((count & 1) != 0)
            maxheap.add(minheap.poll());
    }
    public double findMedian() {
        if ((count & 1) == 0) {
            return (double) (maxheap.peek() + minheap.peek()) / 2;
        } else {
            return (double) maxheap.peek();
        }
```

```
    }
}
```

汽车从起点出发驶向目的地，该目的地位于出发位置东面 target 英里处。

沿途有加油站，每个 station[i] 代表一个加油站，它位于出发位置东面 station[i][0] 英里处，并且有 station[i][1] 升汽油。

假设汽车油箱的容量是无限的，其中最初有 startFuel 升燃料。它每行驶 1 英里就会用掉 1 升汽油。

当汽车到达加油站时，它可能停下来加油，将所有汽油从加油站转移到汽车中。

为了到达目的地，汽车所必要的最低加油次数是多少？如果无法到达目的地，则返回 -1 。

注意：如果汽车到达加油站时剩余燃料为 0，它仍然可以在那里加油。如果汽车到达目的地时剩余燃料为 0，仍然认为它已经到达目的地。

```java
class _871最低加油次数 {
    public int minRefuelStops(int target, int startFuel, int[][]
stations) {
        Queue<Integer> queue = new PriorityQueue<>((a,b)->b-a);
        int cnt = 0;
        int distance = startFuel;
        for (int[] station : stations) {
            while (!queue.isEmpty() && distance < station[0]){
                distance += queue.poll();
                cnt ++;
            }
            if (distance < station[0])
                return -1;
            queue.offer(station[1]);
        }
        while(!queue.isEmpty() && distance < target){
            distance += queue.poll();
            cnt++;
        }
        return distance >= target ? cnt : -1;
    }
}
```

假设 力扣（LeetCode）即将开始其 IPO。为了以更高的价格将股票卖给风险投资公司，力扣希望在 IPO 之前开展一些项目以增加其资本。 由于资源有限，它只能在 IPO 之前完成最多 k 个不同的项目。帮助 力扣 设计完成最多 k 个不同项目后得到最大总资本的方式。

给定若干个项目。对于每个项目 i，它都有一个纯利润 Pi，并且需要最小的资本 Ci 来启动相应的项目。最初，你有 W 资本。当你完成一个项目时，你将获得纯利润，且利润将被添加到你的总资本中。

总而言之，从给定项目中选择最多 k 个不同项目的列表，以最大化最终资本，并输出最终可获得的最多资本。

```
class _502IPO {
    private PriorityQueue<int[]> project;
    private PriorityQueue<Integer> available;
    public int findMaximizedCapital(int k, int W, int[] Profits, int[]
Capital) {
        project = new PriorityQueue<>((a,b)->a[0]-b[0]);
        available = new PriorityQueue<>((a,b)->b-a);
        for(int i=0; i < Capital.length; i++)
            project.offer(new int[]{Capital[i],Profits[i]});
        while(k > 0){
            while(!project.isEmpty()){
                int[] pair = project.peek();
                if(W >= pair[0]){
                    available.offer(pair[1]);
                    project.poll();
                }
                else
                    break;
            }
            if(!available.isEmpty()){
                W += available.poll();
                k--;
            }else{
                break;
            }
        }
        return W;
    }
}
```


1.3链表

1.3.1 反转链表

反转从位置 m 到 n 的链表。请使用一趟扫描完成反转。
说明:
1 ≤ m ≤ n ≤ 链表长度。

```
public class _92反转链表II {
    ListNode successor = null;
    public ListNode reverseBetween(ListNode head, int m, int n) {
        if(m == 1)
            return reverseN(head, n);
        head.next = reverseBetween(head.next,m-1,n-1);
        return head;
    }
    private ListNode reverseN(ListNode head, int n) {
        if (n == 1) {
            successor = head.next;
            return head;
        }
        ListNode last = reverseN(head.next,n-1);
        head.next.next = head;
        head.next = successor;
        return last;
    }
    //解法2
    public ListNode reverseBetween2(ListNode head, int m, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode pre = dummy;
        for(int i = 1;i < m; i++)
            pre = pre.next;
        ListNode cur = pre.next;
        for(int i = m;i < n; i++){
            ListNode next = cur.next;
            cur.next = next.next;
            next.next = pre.next;
            pre.next = next;
        }
        return dummy.next;
    }
}
```


变式题

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。
k 是一个正整数，它的值小于或等于链表的长度。
如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

```
public class _25K个一组翻转链表 {
    public ListNode reverseKGroup(ListNode head, int k) {
```

```
        if(head == null)
            return null;
        ListNode b = head;
        for(int i = 0; i < k; i++) {
            if(b == null)
                return head;
            b = b.next;
        }
        ListNode newHead = reverse(head, b);
        head.next = reverseKGroup(b, k);
        return newHead;
    }
    private ListNode reverse(ListNode head, ListNode end) {
        ListNode pre = null;
        ListNode cur = head;
        while(cur != end){
            ListNode next = cur.next;
            cur.next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
}
```

1.3.2 快慢指针

> 给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

```
class _19删除链表的倒数第N个节点 {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode fast = head;
        ListNode slow = dummy;
        for(int i=0;i<n;i++)
            fast = fast.next;
        while(fast != null){
            fast = fast.next;
            slow = slow.next;
        }
        slow.next = slow.next.next;
        return dummy.next;
    }
}
```

变式题

> 给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。
> 为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。
> 说明：不允许修改给定的链表。

```
public class _142环形链表II {
    public ListNode detectCycle(ListNode head) {
        ListNode intersect = getIntersection(head);
        if(intersect == null)
            return null;
        ListNode p1 = head;
        ListNode p2 = intersect;
        while(p1 != p2){
            p1 = p1.next;
            p2 = p2.next;
        }
        return p1;
    }
    private ListNode getIntersection(ListNode head){
        ListNode slow = head;
        ListNode fast = head;
        while(fast != null && fast.next != null){
            slow = slow.next;
            fast = fast.next.next;
            if(slow == fast)
                return slow;
        }
        return null;
    }
}
```

1.3.3 链表排序

> 对链表进行插入排序。

```
class _147对链表进行插入排序 {
    public ListNode insertionSortList(ListNode head) {
        ListNode dummy = new ListNode(-1);
        while(head != null){
            ListNode p = dummy;
            while(p.next != null && p.next.val < head.val)
                p = p.next;
            ListNode q = head;
            head = head.next;
            q.next = p.next;
            p.next = q;
        }
```

```
        return dummy.next;
    }
}
```

变式题

> 在 O(n log n) 时间复杂度和常数级空间复杂度下，对链表进行排序。

```
class _148排序链表 {
   public ListNode sortList(ListNode head) {
        if(head == null || head.next == null)
            return head;
        ListNode fast = head.next, slow = head;
        while(fast != null && fast.next != null){
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode mid = slow.next;
        slow.next = null;
        ListNode left = sortList(head);
        ListNode right = sortList(mid);
        ListNode h = new ListNode(0);
        ListNode ret = h;
        while(left != null && right != null){
            if(left.val < right.val){
                h.next = left;
                left = left.next;
                h = h.next;
            }else{
                h.next = right;
                right = right.next;
                h = h.next;
            }
        }
        h.next = left == null ? right : left;
        return ret.next;
    }
}
```


1.3.4 合并链表

> 合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

分治解法

```
class _23合并K个排序链表 {
    public ListNode mergeKLists(ListNode[] lists) {
        if(lists.length == 0)
```

```
            return null;
        if(lists.length == 1)
            return lists[0];
        if(lists.length == 2)
           return mergeTwoLists(lists[0],lists[1]);
        int mid = lists.length/2;
        ListNode[] l1 = new ListNode[mid];
        for(int i = 0; i < mid; i++)
            l1[i] = lists[i];
        ListNode[] l2 = new ListNode[lists.length-mid];
        for(int j = 0; j + mid < lists.length; j++)
            l2[j] = lists[j+mid];
        return mergeTwoLists(mergeKLists(l1),mergeKLists(l2));
    }
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1 == null)
            return l2;
        if(l2 == null)
            return l1;
        if(l1.val < l2.val){
            l1.next = mergeTwoLists(l1.next, l2);
            return l1;
        }else{
            l2.next = mergeTwoLists(l1, l2.next);
            return l2;
        }
    }
}
```

优先队列解法

```
class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        PriorityQueue<ListNode> queue = new PriorityQueue<>
(Comparator.comparingInt(a -> a.val));
        ListNode ret = new ListNode(0);
        for(int i=0;i < lists.length;i++){
            if(lists[i]!= null)
                queue.offer(lists[i]);
        }
        ListNode cur = ret;
        while(!queue.isEmpty()){
            ListNode c = queue.poll();
            cur.next = c;
            cur = cur.next;
            if(c.next != null)
                queue.offer(c.next);
        }
        return ret.next;
```

```
    }
}
```

1.3.5 链表与二叉树

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

```
class _109有序链表转换二叉搜索树 {
    private ListNode node;
    public TreeNode sortedListToBST(ListNode head) {
        int n = 0;
        node = head;
        while(head != null){
            head = head.next;
            n++;
        }
        return toBST(0, n-1);
    }
    private TreeNode toBST(int left, int right){
        if(left > right) return null;
        int m = (left + right) / 2;
        TreeNode left_child = toBST(left, m-1);
        TreeNode father = new TreeNode(node.val);
        node = node.next;
        father.left = left_child;
        father.right = toBST(m+1, right);
        return father;
    }
}
```

1.3.6 去重问题

给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

```
class _83删除排序链表中的重复元素 {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode current = head;
        while (current != null && current.next != null) {
            if(current.next.val == current.val) {
                current.next = current.next.next;
            }else{
                current = current.next;
            }
        }
        return head;
    }
}
```

变式题

> 给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 没有重复出现 的数字。

```
public class _82删除排序链表中的重复元素II {
    public ListNode deleteDuplicates(ListNode head) {
        if(head == null || head.next == null)
            return head;
        ListNode cur = head.next;
        if(cur.val != head.val){
            head.next = deleteDuplicates(head.next);
        }else{
            while(cur != null && cur.val == head.val){
                cur = cur.next;
            }
            head = deleteDuplicates(cur);
        }
        return head;
    }
}
```


1.4二叉树

1.4.1 二叉树属性

1.4.1.1 完全二叉树节点

> 给出一个完全二叉树，求出该树的节点个数。
>
> 说明：
>
> 完全二叉树的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层，则该层包含 1~ 2h 个节点。

普通解法

```
class _222完全二叉树的节点个数 {
    public int countNodes(TreeNode root) {
        return root != null ? 1 + countNodes(root.right) +
countNodes(root.left) : 0;
    }
}
```

高效解法

```
class _222完全二叉树的节点个数 {
    public int countNodes(TreeNode root) {
        if(root == null)
           return 0;
        int left = countLevel(root.left);
        int right = countLevel(root.right);
        return left == right ? countNodes(root.right) + (1<<left) :
countNodes(root.left) + (1<<right);

    }
    private int countLevel(TreeNode root){
        int level = 0;
        while(root != null){
            level++;
            root = root.left;
        }
        return level;
    }
}
```

1.4.1.2 二叉树直径

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

```
class _543二叉树的直径 {
    int ans;
    public int diameterOfBinaryTree(TreeNode root) {
        ans = 1;
```

```
        depth(root);
        return ans - 1;
    }
    public int depth(TreeNode node) {
        if (node == null)
            return 0;
        int L = depth(node.left);
        int R = depth(node.right);
        ans = Math.max(ans, L+R+1);
        return Math.max(L, R) + 1;
    }
}
```

1.4.2 二叉树性质问题

1.4.2.1 对称性

> 给定一个二叉树，检查它是否是镜像对称的。

递归解法

```
class _101对称二叉树 {
    public boolean isSymmetric(TreeNode root) {
        return isSymmetric(root, root);
    }
    private boolean isSymmetric(TreeNode left, TreeNode right){
        if(left == null && right == null)
            return true;
        if(left == null || right == null)
            return false;
        return left.val == right.val
            && isSymmetric(left.left, right.right)
            && isSymmetric(left.right, right.left);
    }
}
```

遍历解法

```
class _101对称二叉树 {
    public boolean isSymmetric(TreeNode root) {
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        queue.add(root);
        while(!queue.isEmpty()){
            TreeNode node1 = queue.poll();
            TreeNode node2 = queue.poll();
            if(node1 == null && node2 == null)
                continue;
```

```
            if(node1 == null || node2 == null)
                return false;
            if(node1.val != node2.val)
                return false;
            queue.offer(node1.left);
            queue.offer(node2.right);
            queue.offer(node1.right);
            queue.offer(node2.left);
        }
        return true;
    }
}
```

1.4.2.2 平衡性

给定一个二叉树，判断它是否是高度平衡的二叉树。
本题中，一棵高度平衡二叉树定义为：
一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过1。

```
class _110平衡二叉树 {
    public boolean isBalanced(TreeNode root) {
        if(root == null)
            return true;
        if(Math.abs(getHeight(root.left)-getHeight(root.right)) > 1)
            return false;
        return isBalanced(root.left) && isBalanced(root.right);
    }
    private int getHeight(TreeNode root){
        if(root == null)
            return 0;
        return Math.max(getHeight(root.left),getHeight(root.right)) +
1;
    }
}
```

1.4.2.3 二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。
节点的右子树只包含大于当前节点的数。
所有左子树和右子树自身必须也是二叉搜索树。

递归解法

```
class _98验证二叉搜索树 {
```

```
    public boolean isValidBST(TreeNode root) {
        return isValidBST(root, null, null);
    }
    private boolean isValidBST(TreeNode root, TreeNode min, TreeNode
max) {
        if(root == null)
            return true;
        if(min != null && root.val <= min.val)
            return false;
        if(max != null && root.val >= max.val)
            return false;
        return isValidBST(root.left,min,root) &&
isValidBST(root.right,root,max);
    }
}
```

中序遍历递归解法

```
class _98验证二叉搜索树 {
    double last = - Double.MAX_VALUE;
    public boolean isValidBST(TreeNode root) {
        if(root == null)
            return true;
        if(isValidBST(root.left)){
            if(last < root.val) {
                last = root.val;
                return isValidBST(root.right);
            }
        }
        return false;
    }
}
```

中序遍历迭代解法

```
class _98验证二叉搜索树 {
    public boolean isValidBST(TreeNode root) {
        Stack<TreeNode> stack = new Stack<>();
        double inorder = - Double.MAX_VALUE;
        TreeNode cur = root;
        while(!stack.isEmpty() || cur != null){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            cur = stack.pop();
            if(cur.val <= inorder)
                return false;
            inorder = cur.val;
```

```
            cur = cur.right;
        }
        return true;
    }
}
```

1.4.2.4 子树

> 给定两个非空二叉树 s 和 t，检验 s 中是否包含和 t 具有相同结构和节点值的子树。s 的一个子树包括 s 的一个节点和这个节点的所有子孙。s 也可以看做它自身的一棵子树。

```
class _572另一个树的子树 {
    public boolean isSubtree(TreeNode s, TreeNode t) {
        if(s == null)
            return t == null;
        if(isSameTree(s,t))
            return true;
        return isSubtree(s.left, t) || isSubtree(s.right, t);
    }
    private boolean isSameTree(TreeNode s, TreeNode t){
        if(s == null && t == null)
            return true;
        if(s == null || t == null)
            return false;
        return s.val == t.val && isSameTree(s.left, t.left) &&
isSameTree(s.right, t.right);
    }
}
```


1.4.3 二叉树遍历

1.4.3.1 前中后序的迭代解法

栈辅助法

```
class _144二叉树的前序遍历 {
    public List<Integer> preorderTraversal(TreeNode root) {
        Stack<TreeNode> stack = new Stack<>();
        List<Integer> list = new ArrayList<>();
        if(root == null)
            return list;
        stack.push(root);
        while(!stack.isEmpty()){
            TreeNode node = stack.pop();
            if(node.right != null)
                stack.push(node.right);
            if(node.left != null)
                stack.push(node.left);
            list.add(node.val);
```

```
        }
        return list;
    }
}
class _94二叉树的中序遍历  {
    public List<Integer> inorderTraversal(TreeNode root) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode cur = root;
        List<Integer> res = new ArrayList<>();
        while(!stack.isEmpty() || cur != null){
            while(cur != null){
                stack.push(cur);
                cur = cur.left;
            }
            cur = stack.pop();
            res.add(cur.val);
            cur = cur.right;
        }
        return res;
    }
}
class _145二叉树的后序遍历 {
    public List<Integer> postorderTraversal(TreeNode root) {
        ArrayList<Integer> ret = new ArrayList<>();
        if (root != null){
            Stack<TreeNode> stack = new Stack<>();
            stack.push(root);
            TreeNode c = null;
            while (!stack.isEmpty()){
                c = stack.peek();
                if (c.left != null && root != c.left && root !=
c.right){
                    stack.push(c.left);
                }else if (c.right != null && root != c.right){
                    stack.push(c.right);
                }else{
                    ret.add(stack.pop().val);
                    root = c;
                }
            }
        }
        return ret;
    }
}
```

Morris遍历法*

```
class _144二叉树的前序遍历 {
    public List<Integer> preorderTraversal(TreeNode head) {
```

```
        List<Integer> ret = new ArrayList<>();
        if (head == null)
            return ret;
        TreeNode cur1 = head;
        TreeNode cur2 = null;
        while (cur1 != null){
            cur2 = cur1.left;
            if (cur2 != null) {
                while (cur2.right != null && cur2.right != cur1) {
                    cur2 = cur2.right;
                } //左子树上最右节点
                if (cur2.right == null) {
                    cur2.right = cur1; //第一次遇到cur1
                    ret.add(cur1.val);
                    cur1 = cur1.left;
                    continue;
                } else { //第二次遇到cur1
                    cur2.right = null;
                }
            }else {
                ret.add(cur1.val);
            }
            cur1 = cur1.right;
        }
        return ret;
    }
}
class _94二叉树的中序遍历 {
    public List<Integer> inorderTraversal(TreeNode head) {
        List<Integer> res = new ArrayList<>();
        if (head == null)
            return res;
        TreeNode cur1 = head;
        TreeNode cur2 = null;
        while (cur1 != null){
            cur2 = cur1.left;
            if (cur2 != null){
                while (cur2.right != null && cur2.right != cur1){
                    cur2 = cur2.right;
                } //左子树上最右节点
                if (cur2.right == null){
                    cur2.right = cur1; //第一次遇到cur1
                    cur1 = cur1.left;
                    continue;
                }else { //第二次遇到cur1
                    cur2.right = null;
                }
            }
            res.add(cur1.val);
```

```
            cur1 = cur1.right;
        }
        return res;
    }
}
```

1.4.3.2 层序遍历

> 给你一个二叉树，请你返回其按 层序遍历 得到的节点值。 （即逐层地，从左到右访问所有节点）。

队列解法

```
class _102二叉树的层序遍历 {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if(root == null)
            return res;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        while(!queue.isEmpty()){
            int size = queue.size();
            List<Integer> list = new ArrayList<>();
            for(int i = 0; i < size; i++){
                TreeNode cur = queue.poll();
                list.add(cur.val);
                if(cur.left != null)
                    queue.offer(cur.left);
                if(cur.right != null)
                    queue.offer(cur.right);
            }
            res.add(list);
        }
        return res;
    }
}
```

DFS解法

```
class _102二叉树的层序遍历 {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> list = new ArrayList<>();
        pre(root,0,list);
        return list;
    }
    private void pre(TreeNode root, int depth, List<List<Integer>>
list){
        if(root == null)
```

```
            return;
        if(depth == list.size())
            list.add(new ArrayList<>());
        list.get(depth).add(root.val);
        pre(root.left, depth+1, list);
        pre(root.right, depth+1, list);
    }
}
```

1.4.4 二叉树的增删改

1.4.4.1 删除二叉搜索树中的节点

给定一个二叉搜索树的根节点 root 和一个值 key，删除二叉搜索树中的 key 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

首先找到需要删除的节点；
如果找到了，删除它。
说明： 要求算法时间复杂度为 O(h)，h 为树的高度。`

```
class _450删除二叉搜索树中的节点 {
    public TreeNode deleteNode(TreeNode root, int key) {
        if(root == null)
            return null;
        if(root.val == key) {
            if(root.left == null)
                return root.right;
            if(root.right == null)
                return root.left;
            TreeNode successor = getMin(root.right);
            successor.right = deleteMin(root.right);
            successor.left = root.left;
            return successor;
        }else if(root.val > key){
            root.left = deleteNode(root.left, key);
        }else{
            root.right = deleteNode(root.right, key);
        }
        return root;
    }
    private TreeNode getMin(TreeNode root){
        while(root.left != null)
            root = root.left;
        return root;
    }
    private TreeNode deleteMin(TreeNode node){
        if (node.left == null)
```

```
            return node.right;
        node.left = deleteMin(node.left);
        return node;
    }
}
```

1.4.4.2 恢复二叉搜索树

> 二叉搜索树中的两个节点被错误地交换。
> 请在不改变其结构的情况下，恢复这棵树。

```
class _99恢复二叉搜索树 {
    private void swap(TreeNode x, TreeNode y){
        int val = x.val;
        x.val = y.val;
        y.val = val;
    }
    public void recoverTree(TreeNode root) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode pre = null, x = null, y = null;
        while(!stack.isEmpty() || root != null){
            while(root != null){
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            if(pre != null && root.val < pre.val){
                y = root;
                if(x == null)
                    x = pre;
            }
            pre = root;
            root = root.right;
        }
        swap(x, y);
    }
}
```

1.4.5 二叉树序列化

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

```
public class _297二叉树的序列化与反序列化 {
    public String serialize(TreeNode root) {
        if (root == null)
            return "#_";
```

```
        return root.val + "_" + serialize(root.left) +
serialize(root.right);
    }
    public TreeNode deserialize(String data) {
        String[] split = data.split("_");
        Queue<String> queue = new LinkedList<>();
        for (String s : split)
            queue.offer(s);
        return preConstruct(queue);
    }
    private TreeNode preConstruct(Queue<String> queue) {
        String value = queue.poll();
        if ("#".equals(value))
            return null;
        TreeNode ret = new TreeNode(Integer.parseInt(value));
        ret.left = preConstruct(queue);
        ret.right = preConstruct(queue);
        return ret;
    }
}
```

1.4.6 二叉树构造

1.4.6.1 不同的二叉搜索树

> 给定一个整数 n，生成所有由 1 ... n 为节点所组成的二叉搜索树。

```
class _95不同的二叉搜索树II {
    public List<TreeNode> generateTrees(int n) {
        if(n == 0)
            return new ArrayList<>();
        return generateTrees(1, n);
    }
    private List<TreeNode> generateTrees(int start, int end){
        List<TreeNode> ret = new ArrayList<>();
        if(start > end){
            ret.add(null);
            return ret;
        }
        for(int i = start; i <= end; i++){
            List<TreeNode> left = generateTrees(start, i-1);
            List<TreeNode> right = generateTrees(i+1, end);
            for(TreeNode l : left){
                for(TreeNode r : right){
                    TreeNode node = new TreeNode(i);
                    node.left = l;
                    node.right = r;
                    ret.add(node);
                }
```

```
            }
        }
        return ret;
    }
}
```

思考题

> 给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种?

```
class _96不同的二叉搜索树 {
    public int numTrees(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= n; ++i)
            for (int j = 1; j <= i; ++j)
              dp[i] += dp[j - 1] * dp[i - j];
        return dp[n];
    }
    //
    public int numTrees2(int n) {
        long res = 1;
        for (int i = 0; i < n; ++i)
            res = res*2*(2*i+1)/(i+2);
        return (int)res;
    }
}
```

$F(i,n) = G(i-1) \cdot G(n-i)$ 表示以i为根的不同二叉搜索树个数

$G(n) = \sum_{i=1}^{n} G(i-1) \cdot G(n-i)$

卡特兰数

$C_0 = 1, C_{n+1} = \frac{2(2n+1)}{n+2} C_n$

1.4.6.2 前序和中序构造二叉搜索树

> 根据一棵树的前序遍历与中序遍历构造二叉树。

```
class _105从前序与中序遍历序列构造二叉树 {
    HashMap<Integer, Integer> map = new HashMap<>();
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        for (int i = 0; i < inorder.length; i++) {
            map.put(inorder[i], i);
        }
```

```
        return buildTree(preorder,inorder,0, preorder.length-
1,0,inorder.length-1);
    }
    private TreeNode buildTree(int[] preorder, int[] inorder, int
pStart, int pEnd, int iStart, int iEnd) {
        if (iStart > iEnd)
            return null;
        TreeNode root = new TreeNode(preorder[pStart]);
        int i = map.get(preorder[pStart]);
        int num = i - iStart;
        root.left =
buildTree(preorder,inorder,pStart+1,pStart+num,iStart,i-1);
        root.right =
buildTree(preorder,inorder,pStart+num+1,pEnd,i+1,iEnd);
        return root;
    }
}
```

变式题

> 根据一棵树的中序遍历与后序遍历构造二叉树。

```
class _106从中序与后序遍历序列构造二叉树 {
    private Map<Integer,Integer> map = new HashMap<>();
    public TreeNode buildTree(int[] inorder, int[] postorder) {
        for(int i=0;i<inorder.length;i++)
            map.put(inorder[i],i);
        return buildTree(inorder, postorder, 0, inorder.length-1, 0,
postorder.length-1);
    }
    private TreeNode buildTree(int[] inorder, int[] postorder, int il,
int ir, int pl, int pr){
        if(pl > pr || il > ir)
            return null;
        int val = postorder[pr];
        TreeNode root = new TreeNode(val);
        int pos = map.get(val);
        root.left = buildTree(inorder, postorder, il, pos-1, pl,
pl+pos-1-il);
        root.right = buildTree(inorder, postorder, pos+1, ir, pl+pos-
il, pr-1);
        return root;
    }
}
```

1.4.7 路径和问题

1.4.7.1 最大路径和

> 给定一个非空二叉树，返回其最大路径和。
> 本题中，路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径至少包含一个节点，且不一定经过根节点。

```
class _124二叉树中的最大路径和 {
    private int res = Integer.MIN_VALUE;
    public int maxPathSum(TreeNode root) {
        dfs(root);
        return res;
    }
    private int dfs(TreeNode root) {
        if(root == null)
            return 0;
        int left = Math.max(0,dfs(root.left));
        int right = Math.max(0,dfs(root.right));
        res = Math.max(res, left + right + root.val);
        return Math.max(left, right) + root.val;
    }
}
```

1.4.7.2 路径和求解

> 给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。
> 说明: 叶子节点是指没有子节点的节点。

```
class _113路径总和II {
    private List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> pathSum(TreeNode root, int sum) {
        if(root == null)
            return res;
        dfs(root, sum, new LinkedList<Integer>());
        return res;
    }
    private void dfs(TreeNode root, int sum, LinkedList<Integer>list){
        if(root.left == null && root.right == null && root.val == sum)
{
            list.addLast(root.val);
            res.add(new LinkedList<>(list));
            list.removeLast();
            return;
        }
        if(root.left != null){
            list.addLast(root.val);
            dfs(root.left, sum-root.val, list);
            list.removeLast();
        }
        if(root.right != null){
            list.addLast(root.val);
```

```
            dfs(root.right, sum-root.val, list);
            list.removeLast();
        }
    }
}
```

变式题

> 给定一个二叉树，它的每个结点都存放着一个整数值。
> 找出路径和等于给定数值的路径总数。
> 路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。
> 二叉树不超过1000个节点，且节点数值范围是 [-1000000,1000000] 的整数。

```
class _437路径总和III{
    public int pathSum(TreeNode root, int sum) {
        if(root == null)
            return 0;
        int res = 0;
        res += path(root, sum);
        res += pathSum(root.left, sum);
        res += pathSum(root.right, sum);
        return res;
    }
    private int path(TreeNode root, int sum){
        int res = root.val == sum ? 1 : 0;
        if(root.left != null)
            res += path(root.left, sum-root.val);
        if(root.right != null)
            res += path(root.right, sum-root.val);
        return res;
    }
}
```

1.4.8 公共祖先问题

> 给定一个二叉搜索树，找到该树中两个指定节点的最近公共祖先。

```
class _235二叉搜索树的最近公共祖先{
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
TreeNode q) {
        if(root == null)
            return null;
        if(p.val < root.val && q.val < root.val)
            return lowestCommonAncestor(root.left,p,q);
        if(p.val > root.val && q.val > root.val)
            return lowestCommonAncestor(root.right,p,q);
        return root;
    }
}
```

变式题

> 给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

```
class _236二叉树的最近公共祖先 {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
TreeNode q) {
        if(root == null || p == root || q == root)
            return root;
        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);
        if(left != null && right != null)
            return root;
        return left == null ? right : left;
    }
}
```

1.5并查集

> 班上有 N 名学生。其中有些人是朋友，有些则不是。他们的友谊具有是传递性。如果已知 A 是 B 的朋友，B 是 C 的朋友，那么我们可以认为 A 也是 C 的朋友。所谓的朋友圈，是指所有朋友的集合。
>
> 给定一个 N * N 的矩阵 M，表示班级中学生之间的朋友关系。如果M[i][j] = 1，表示已知第 i 个和 j 个学生互为朋友关系，否则为不知道。你必须输出所有学生中的已知的朋友圈总数。

```
class _547朋友圈 {
    public int findCircleNum(int[][] M) {
        int n = M.length;
        UnionFind uf = new UnionFind(n);
        for(int i = 1; i < n; i++)
            for(int j = 0; j < i; j++)
                if(M[i][j] == 1)
```

```
                    uf.union(i, j);
            return uf.getCount();
        }
    }
    class UnionFind{
        private int[] parent;
        private int[] rank;
        private int count;
        public UnionFind(int size) {
            parent = new int[size];
            rank = new int[size];
            count = size;
            for(int i=0;i<size;i++){
                parent[i] = i;
                rank[i] = 1;
            }
        }
        private int find(int p){
            if(p != parent[p])
                parent[p] = find(parent[p]);
            return parent[p];
        }
        public int getCount(){
            return count;
        }
        public void union(int p, int q){
            int pRoot = find(p);
            int qRoot = find(q);
            if(pRoot == qRoot)
                return;
            if(rank[pRoot] < rank[qRoot])
                parent[pRoot] = qRoot;
            else if(rank[pRoot] > rank[qRoot])
                parent[qRoot] = pRoot;
            else{
                parent[pRoot] = qRoot;
                rank[qRoot] ++;
            }
            count --;
        }
    }
```


变式题

> 给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。
>
> 岛屿总是被水包围，并且每座岛屿只能由水平方向或竖直方向上相邻的陆地连接形成。
>
> 此外，你可以假设该网格的四条边均被水包围。

```java
class _200岛屿数量 {
    class UnionFind{
        private int[] parent;
        private int[] rank;
        public int count = 0;
        public UnionFind(char[][] grid){
            int m = grid.length;
            int n = grid[0].length;
            parent = new int[m*n];
            rank = new int[m*n];
            for(int i=0;i<m;i++){
                for(int j=0;j<n;j++){
                    if(grid[i][j]=='1'){
                        count ++;
                        parent[i*n+j] = i*n+j;
                        rank[i*n+j] = 1;
                    }
                }
            }
        }
        private int find(int p){
            if(p != parent[p])
                parent[p] = find(parent[p]);
            return parent[p];
        }
        public void union(int x, int y){
            int p = find(x);
            int q = find(y);
            if(p == q)
                return;
            if(rank[p]<rank[q])
                parent[p] = q;
            else if(rank[q]<rank[p])
                parent[q] = p;
            else{
                parent[p] = q;
                rank[q] ++;
            }
            count --;
        }

    }
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0)
            return 0;
        int m = grid.length;
        int n = grid[0].length;
        UnionFind uf = new UnionFind(grid);
        for(int i=0;i<m;i++){
```

```
            for(int j=0;j<n;j++){
                if(grid[i][j]=='1'){
                    if(i+1<m && grid[i+1][j]=='1')
                        uf.union(i*n+j,(i+1)*n+j);
                    if(j+1<n && grid[i][j+1]=='1')
                        uf.union(i*n+j,i*n+j+1);
                }
            }
        }
        return uf.count;
    }
}
```

1.6哈希表

给定一个整数数组和一个整数 k，你需要找到该数组中和为 k 的连续的子数组的个数。

```
class _560和为K的子数组 {
    public int subarraySum(int[] nums, int k) {
        int ret = 0;
        int sum = 0;
        Map<Integer,Integer> map = new HashMap<>();
        map.put(0,1);
        for(int i=0;i<nums.length;i++){
            sum += nums[i];
            if(map.containsKey(sum-k))
                ret += map.get(sum-k);
            map.put(sum,map.getOrDefault(sum,0)+1);
        }
        return ret;
    }
}
```

变式题

给你一个整数数组 nums 和一个整数 k。

如果某个 连续 子数组中恰好有 k 个奇数数字，我们就认为这个子数组是「优美子数组」。

请返回这个数组中「优美子数组」的数目。

```
class _1248统计优美子数组 {
    public int numberOfSubarrays(int[] nums, int k) {
        int sum = 0;
        int ret = 0;
        Map<Integer,Integer> map = new HashMap<>();
        map.put(0, 1);
```

```
        for(int i=0;i<nums.length;i++){
            nums[i] &= 1;
            sum += nums[i];
            if(map.containsKey(sum-k))
                ret += map.get(sum-k);
            map.put(sum,map.getOrDefault(sum,0)+1);
        }
        return ret;
    }
}
```

变式题

给定一个二进制数组, 找到含有相同数量的 0 和 1 的最长连续子数组（的长度）。

```
class _525连续数组 {
    public int findMaxLength(int[] nums) {
        HashMap<Integer,Integer> map = new HashMap<>();
        int max = 0;
        int sum = 0;
        map.put(0, -1);
        for(int i = 0; i < nums.length; i++){
            sum += nums[i] == 1 ? 1 : -1;
            if(map.containsKey(sum))
                max = Math.max(max, i-map.get(sum));
            else
                map.put(sum, i);
        }
        return max;
    }
}
```

和谐数组是指一个数组里元素的最大值和最小值之间的差别正好是1。
现在，给定一个整数数组，你需要在所有可能的子序列中找到最长的和谐子序列的长度。

```
class _594最长和谐子序列 {
    public int findLHS(int[] nums) {
        int res = 0;
        Map<Integer, Integer> map = new HashMap<>();
        for(int i = 0;i < nums.length; i++){
            map.put(nums[i],map.getOrDefault(nums[i],0)+1);
        }
        for(int key : map.keySet()){
            if(map.containsKey(key+1)){
                res = Math.max(map.get(key)+map.get(key+1),res);
            }
        }
        return res;
```

```
    }
}
```

给定两个整数，分别表示分数的分子 numerator 和分母 denominator，以字符串形式返回小数。

如果小数部分为循环小数，则将循环的部分括在括号内。

```
class _166分数到小数 {
    public String fractionToDecimal(int numerator, int denominator) {
        if(numerator == 0)
            return "0";
        StringBuilder sb = new StringBuilder();
        if(numerator < 0 ^ denominator < 0)
            sb.append("-");
        long n = Math.abs(Long.valueOf(numerator));
        long d = Math.abs(Long.valueOf(denominator));
        long reminder = n % d;
        if(reminder == 0){
            sb.append(n/d);
            return sb.toString();
        }
        sb.append(n/d);
        sb.append(".");
        Map<Long, Integer> map = new HashMap<>();
        while(reminder != 0){
            if(map.containsKey(reminder)){
                sb.insert(map.get(reminder),"(");
                sb.append(")");
                return sb.toString();
            }
            map.put(reminder, sb.length());
            reminder *= 10;
            sb.append(reminder/d);
            reminder %= d;
        }
        return sb.toString();
    }
}
```

给定四个包含整数的数组列表 A , B , C , D ,计算有多少个元组 (i, j, k, l) ，使得 A[i] + B[j] + C[k] + D[l] = 0。

为了使问题简单化，所有的 A, B, C, D 具有相同的长度 N，且 0 ≤ N ≤ 500 。所有整数的范围在 -228 到 2^28 - 1 之间，最终结果不会超过 2^31 - 1 。

```
class _454四数相加II {
```

```
    public int fourSumCount(int[] A, int[] B, int[] C, int[] D) {
        Map<Integer,Integer> map = new HashMap<>();
        for (int a : A)
            for (int b : B)
                map.put(a + b, map.getOrDefault(a + b, 0) + 1);
        int res = 0;
        for(int c : C)
            for (int d : D)
                if (map.containsKey(-c-d))
                    res += map.get(-c-d);
        return res;
    }
}
```

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

```
class _49字母异位词分组 {
    public List<List<String>> groupAnagrams(String[] strs) {
        HashMap<String,ArrayList<String>> map=new HashMap<>();
        for(String s:strs){
            char[] ch=s.toCharArray();
            Arrays.sort(ch);
            String key=String.valueOf(ch);
            if(!map.containsKey(key))    map.put(key,new ArrayList<>
());
            map.get(key).add(s);
        }
        return new ArrayList(map.values());
    }
}
```

给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。

```
class _149直线上最多的点数{
    public int maxPoints(int[][] points){
        int n = points.length;
        if(n < 2)
            return n;
        int res = 0;
        for(int i = 0; i < n - 1; i++) {
            Map<String, Integer> map = new HashMap<>();
            int duplicate = 0;
            int max = 0;
            for(int j = i+1; j < n; j++) {
                int dy = points[i][1] - points[j][1];
                int dx = points[i][0] - points[j][0];
```

```
                if(dy == 0 && dx == 0) {
                    duplicate ++;
                    continue;
                }
                int g = gcd(dy, dx);
                if(g != 0){
                    dy /= g;
                    dx /= g;
                }
                String key = dy + "/" + dx;
                map.put(key, map.getOrDefault(key, 0)+1);
                max = Math.max(max, map.get(key));
            }
            res = Math.max(res, duplicate + max + 1);
        }
        return res;
    }
    private int gcd(int x, int y){
        return y == 0 ? x : gcd(y, x%y);
    }
}
```

1.7字典树

```
按下述要求实现 StreamChecker 类：

StreamChecker(words)：构造函数，用给定的字词初始化数据结构。
query(letter)：如果存在某些 k >= 1，可以用查询的最后 k个字符（按从旧到新顺序，包括刚刚查询的字母）拼写出给定字词表中的某一字词时，返回 true。否则，返回 false。
```

```
class _1032字符流 {
    class Trie{
        Node root;
        class Node{
            TreeMap<Character, Node> next;
            boolean isWord;
            Node() {
                next = new TreeMap<>();
            }
        }
        public void add(String word){
            Node cur = root;
            for(int i=word.length()-1;i>=0;i--){
                char c = word.charAt(i);
                if(cur.next.get(c) == null)
                    cur.next.put(c, new Node());
                cur = cur.next.get(c);
            }
```

```
            cur.isWord = true;
        }
        Trie() {
            root = new Node();
        }
    }
    private Trie trie;
    public StreamChecker(String[] words) {
        trie = new Trie();
        for(String s: words)
            trie.add(s);
    }
    private StringBuilder sb = new StringBuilder();
    public boolean query(char letter) {
        sb.append(letter);
        Trie.Node cur = trie.root;
        for(int i=sb.length()-1;i>=0;i--){
            char c = sb.charAt(i);
            if(cur.next.containsKey(c)){
                cur = cur.next.get(c);
                if(cur.isWord)
                    return true;
            }
            else
                return false;
        }
        return false;
    }
}
```

实现一个 MapSum 类里的两个方法，insert 和 sum。

对于方法 insert，你将得到一对（字符串，整数）的键值对。字符串表示键，整数表示值。如果键已经存在，那么原来的键值对将被替代成新的键值对。

对于方法 sum，你将得到一个表示前缀的字符串，你需要返回所有以该前缀开头的键的值的总和。

```
class _677键值映射 {
    class Node{
        int value;
        TreeMap<Character,Node> next;
        Node(int value){
            next = new TreeMap<>();
            this.value = value;
        }
        Node(){
            this(0);
```

```
        }
    }
    private Node root;
    public MapSum() {
        root = new Node();
    }
    public void insert(String word, int val) {
        Node cur = root;
        for(int i=0;i<word.length();i++){
            char c = word.charAt(i);
            if(!cur.next.containsKey(c))
                cur.next.put(c, new Node());
            cur = cur.next.get(c);
        }
        cur.value = val;
    }

    public int sum(String prefix) {
        Node cur = root;
        for(int i=0;i<prefix.length();i++){
            char c = prefix.charAt(i);
            if(!cur.next.containsKey(c))
                return 0;
            cur = cur.next.get(c);
        }
        return sum(cur);
    }
    private int sum(Node node){
        int res = node.value;
        for(char c : node.next.keySet()){
            res += sum(node.next.get(c));
        }
        return res;
    }
}
```

1.8线段树

```
给定一个整数数组  nums，求出数组从索引 i 到 j  (i ≤ j) 范围内元素的总和，包含 i,  j 两点。

update(i, val) 函数可以通过将下标为 i 的数值更新为 val，从而对数列进行修改。
```

```
class _307区域和检索数组可修改 {
    SegmentTree tree;
    public NumArray(int[] nums) {
        if(nums.length != 0)
            tree = new SegmentTree(nums);
```

```
    }
    public void update(int i, int val) {
        tree.set(i, val);
    }

    public int sumRange(int i, int j) {
        return tree.query(i, j);
    }
}
class SegmentTree{
    private int[] data;
    private int[] tree;
    SegmentTree(int[] nums){
        data = new int[nums.length];
        for(int i=0;i<nums.length;i++)
            data[i] = nums[i];
        tree = new int[4*nums.length];
        build(0, 0, data.length-1);
    }
    int leftChild(int i){
        return 2*i+1;
    }
    int rightChild(int i){
        return 2*i+2;
    }
    void build(int index, int l, int r){
        if(l == r){
            tree[index] = data[l];
            return;
        }
        int mid = (l+r)/2;
        int left = leftChild(index);
        int right = rightChild(index);
        build(left,l,mid);
        build(right,mid+1,r);
        tree[index] = tree[left] + tree[right];
    }
    public void set(int index, int value){
        data[index] = value;
        set(0,0,data.length-1,index,value);
    }
    private void set(int index, int l, int r, int i, int value){
        if(l == r){
            tree[index] = value;
            return;
        }
        int mid = (l+r)/2;
        int leftIndex = leftChild(index);
        int rightIndex = rightChild(index);
```

```
        if(i >= mid+1)
            set(rightIndex, mid+1, r, i, value);
        else
            set(leftIndex, l, mid, i, value);
        tree[index] = tree[leftIndex] + tree[rightIndex];
    }
    public int query(int ql, int qr){
        return query(0, 0, data.length-1, ql, qr);
    }
    private int query(int index, int l, int r, int ql, int qr){
        if(l == ql && r == qr)
            return tree[index];
        int mid = (l+r)/2;
        int leftIndex = leftChild(index);
        int rightIndex = rightChild(index);
        if(ql >= mid+1)
            return query(rightIndex, mid+1, r, ql, qr);
        else if(qr <= mid)
            return query(leftIndex, l, mid, ql, qr);
        int left = query(leftIndex, l, mid, ql, mid);
        int right = query(rightIndex, mid+1, r, mid+1, qr);
        return left + right;
    }
}
```

2.算法篇

2.1贪心

2.1.1 区间问题

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:

可以认为区间的终点总是大于它的起点。
区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。

```
class _435无重叠区间 {
    public int eraseOverlapIntervals(int[][] intervals) {
        if(intervals.length == 0)
            return 0;
        Arrays.sort(intervals, (a,b)->a[1]-b[1]);
        int count = 1;
        int end = intervals[0][1];
        for(int[] interval : intervals){
            int start = interval[0];
```

```
            if(start >= end){
                count ++;
                end = interval[1];
            }
        }
        return intervals.length - count;
    }
}
```

变式题

> 在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以y坐标并不重要，因此只要知道开始和结束的x坐标就足够了。开始坐标总是小于结束坐标。平面内最多存在104个气球。
>
> 一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

```
class _452用最少数量的箭引爆气球 {
    public int findMinArrowShots(int[][] intervals) {
        if(intervals.length == 0)
            return 0;
        Arrays.sort(intervals, Comparator.comparingInt(a -> a[1]));
        int count = 1;
        int end = intervals[0][1];
        for(int[] interval : intervals){
            int start = interval[0];
            if(start > end){
                count ++;
                end = interval[1];
            }
        }
        return count;
    }
}
```

拓展与补充——区间问题专题

> 给出一个区间的集合，请合并所有重叠的区间。

```
class _56合并区间 {
    public int[][] merge(int[][] intervals) {
        if(intervals == null || intervals.length <= 1)
            return intervals;
        List<int[]> list = new ArrayList<>();
        Arrays.sort(intervals, Comparator.comparingInt(a->a[0]));
```

```
        int i = 0;
        while(i < intervals.length){
            int left = intervals[i][0];
            int right = intervals[i][1];
            while(i < intervals.length -1 && right >= intervals[i+1]
[0]){
                right = Math.max(right, intervals[i+1][1]);
                i++;
            }
            list.add(new int[]{left, right});
            i++;
        }
        return list.toArray(new int[list.size()][2]);
    }
}
```

给定两个由一些闭区间组成的列表，每个区间列表都是成对不相交的，并且已经排序。
返回这两个区间列表的交集。

```
class _986区间列表的交集 {
    public int[][] intervalIntersection(int[][] A, int[][] B) {
        List<int[]> ret = new ArrayList<>();
        int i = 0;
        int j = 0;
        while(i < A.length && j < B.length){
            int a0 = A[i][0];
            int a1 = A[i][1];
            int b0 = B[j][0];
            int b1 = B[j][1];
            if(b1 >= a0 && a1 >= b0)
                ret.add(new int[]{Math.max(a0,b0),Math.min(a1,b1)});
            if(b1 < a1)
                j++;
            else
                i++;
        }
        int[][] arr = new int[ret.size()][2];
        for(int k = 0; k < arr.length; k++)
            arr[k] = ret.get(k);
        return arr;
    }
}
```

Range 模块是跟踪数字范围的模块。你的任务是以一种有效的方式设计和实现以下接口。

addRange(int left, int right) 添加半开区间 [left, right)，跟踪该区间中的每个实数。添加与当前跟踪的数字部分重叠的区间时，应当添加在区间 [left, right) 中尚未跟踪的任何数字到该区间中。
queryRange(int left, int right) 只有在当前正在跟踪区间 [left, right) 中的每一个实数时，才返回 true。
removeRange(int left, int right) 停止跟踪区间 [left, right) 中当前正在跟踪的每个实数。

```
class RangeModule {
    private TreeSet<Interval> set;
    public RangeModule() {
        set = new TreeSet<>();
    }
    public void addRange(int left, int right) {
        Iterator<Interval> iterator = set.tailSet(new Interval(0,
left)).iterator();
        while (iterator.hasNext()){
            Interval interval = iterator.next();
            if(right < interval.left)
                break;
            left = Math.min(left,interval.left);
            right = Math.max(right,interval.right);
            iterator.remove();
        }
        set.add(new Interval(left, right));
    }

    public boolean queryRange(int left, int right) {
        Interval higher = set.higher(new Interval(0, left));
        return higher != null && higher.left <= left && higher.right
>= right;
    }

    public void removeRange(int left, int right) {
        Iterator<Interval> intervals = set.tailSet(new Interval(0,
left)).iterator();
        List<Interval> rerange = new ArrayList<>();
        while (intervals.hasNext()){
            Interval interval = intervals.next();
            if (right < interval.left)
                break;
            if (interval.left < left)
                rerange.add(new Interval(interval.left,left));
            if (interval.right > right)
                rerange.add(new Interval(right,interval.right));
            intervals.remove();
```

```
            }
            set.addAll(rerange);
        }
    }
    class Interval implements Comparable<Interval>{
        int left;
        int right;
        public int compareTo(Interval o) {
            if (this.right == o.right)
                return this.left - o.left;
            return this.right - o.right;
        }
        public Interval(int left, int right) {
            this.left = left;
            this.right = right;
        }
    }
```

2.1.2 跳跃游戏

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

```
class _45跳跃游戏II {
    public int jump(int[] nums) {
        if(nums.length < 2)
            return 0;
        int curMax = nums[0];
        int preMax = nums[0];
        int jump = 1;
        for(int i = 1; i < nums.length; i++){
            if(i > curMax){
                jump++;
                curMax = preMax;
            }
            preMax = Math.max(preMax, nums[i]+i);
        }
        return jump;
    }
}
```

变式题

在一条环路上有 N 个加油站，其中第 i 个加油站有汽油 gas[i] 升。

你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。你从其中的一个加油站出发，开始时油箱为空。

如果你可以绕环路行驶一周，则返回出发时加油站的编号，否则返回 -1。

说明:

如果题目有解，该答案即为唯一答案。
输入数组均为非空数组，且长度相同。
输入数组中的元素均为非负数。

```
class _134加油站 {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int g = 0;
        int total = 0;
        int start = 0;
        for(int i=0;i<gas.length;i++){
            g += gas[i] - cost[i];
            total += gas[i] - cost[i];
            if(g < 0){
                start = i+1;
                g = 0;
            }

        }
        return total >= 0 ? start : -1;
    }
}
```

2.2双指针

2.2.1 快慢指针

编写一个算法来判断一个数 n 是不是快乐数。

「快乐数」定义为：对于一个正整数，每一次将该数替换为它每个位置上的数字的平方和，然后重复这个过程直到这个数变为 1，也可能是 无限循环 但始终变不到 1。如果 可以变为 1，那么这个数就是快乐数。

```
class _202快乐数{
    public boolean isHappy(int n) {
        int slow = n;
        int fast = n;
        do {
            slow = bitSquareSum(slow);
```

```
            fast = bitSquareSum(bitSquareSum(fast));
        }while(fast != slow);
        return slow == 1;
    }
    private int bitSquareSum(int n){
        int ret = 0;
        while(n != 0){
            int mod = n % 10;
            ret += mod * mod;
            n /= 10;
        }
        return ret;
    }
}
```

2.2.2 首尾指针

给定一个包含 n 个整数的数组 nums 和一个目标值 target，判断 nums 中是否存在四个元素 a，b，c 和 d ，使得 a + b + c + d 的值与 target 相等？找出所有满足条件且不重复的四元组。

注意：

答案中不可以包含重复的四元组。

```
class _18四数之和 {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> ret = new ArrayList<>();
        Arrays.sort(nums);
        for(int i = 0; i < nums.length; i++){
            if(i > 0 && nums[i] == nums[i-1])
                continue;
            for(int j = i+1; j < nums.length; j++){
                if(j > i+1 && nums[j] == nums[j-1])
                    continue;
                int m = j+1;
                int n = nums.length-1;
                int sum = nums[i] + nums[j];
                while(m < n){
                    if(sum + nums[m] + nums[n] < target){
                        m++;
                    }else if(sum + nums[m] + nums[n] > target){
                        n--;
                    }else{

 ret.add(Arrays.asList(nums[i],nums[j],nums[m],nums[n]));
                        while(m < n && nums[m] == nums[m+1])
                            m++;
```

```
                    while(m < n && nums[n] == nums[n-1])
                        n--;
                    m++;
                    n--;
                }
            }
        }
        return ret;
    }
}
```

变式题

给定一个包括 n 个整数的数组 nums 和 一个目标值 target。找出 nums 中的三个整数，使得它们的和与 target 最接近。返回这三个数的和。假定每组输入只存在唯一答案。

```
class _16最接近的三数之和 {
    public int threeSumClosest(int[] nums, int target) {
        Arrays.sort(nums);
        int closeSum = nums[0] + nums[1] + nums[2];
        for(int i = 0; i < nums.length; i++){
            int j = i+1;
            int k = nums.length-1;
            while(j < k){
                int sum = nums[i]+nums[j]+nums[k];
                if(Math.abs(sum-target) < Math.abs(closeSum-target))
                    closeSum = sum;
                if(sum > target)
                    k--;
                else if(sum < target)
                    j++;
                else
                    return target;
            }
        }
        return closeSum;
    }
}
```

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

说明：你不能倾斜容器，且 n 的值至少为 2。

```
class _11盛最多水的容器 {
    public int maxArea(int[] height) {
```

```
        int left = 0, right = height.length - 1;
        int res = 0;
        while(left < right){
            int h = Math.min(height[left],height[right]);
            res = Math.max(res, h*(right-left));
            if(h == height[left])
                left++;
            else
                right--;
        }
        return res;
    }
}
```

2.2.3 去重

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素最多出现两次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

```
class _80删除排序数组中的重复项II {
    public int removeDuplicates(int[] nums) {
        if(nums.length < 3)
            return nums.length;
        int i = 2;
        for (int j = 2; j < nums.length;j++)
            if (nums[j] > nums[i-2])
                nums[i++] = nums[j];
        return i;
    }
}
```

思考：题目如果改为最多出现3次，应该怎么修改代码?

2.3递归与回溯

2.3.1 组合问题

给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。
candidates 中的数字可以无限制重复被选取。
说明：
所有数字（包括 target）都是正整数。
解集不能包含重复的组合。

```
public class _39组合总和 {
```

```
    private List<List<Integer>> ret;
    public List<List<Integer>> combinationSum(int[] candidates, int
target) {
        ret = new ArrayList<>();
        helper(candidates, target, new ArrayList<>(),0);
        return ret;
    }
    private void helper(int[] candidates, int target, List<Integer>
list, int start){
        if(target == 0){
            ret.add(new ArrayList<>(list));
            return;
        }
        for(int i=start;i<candidates.length;i++){
            if(target >= candidates[i]){
                list.add(candidates[i]);
                helper(candidates, target-candidates[i],list,i);
                list.remove(list.size()-1);
            }
        }
    }
}
```

变式题

```
给定一个数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字
和为 target 的组合。
candidates 中的每个数字在每个组合中只能使用一次。
说明：

所有数字（包括目标数）都是正整数。
解集不能包含重复的组合。
```

```
public class _40组合总和II {
    public List<List<Integer>> combinationSum2(int[] candidates, int
target) {
        Arrays.sort(candidates);
        helper(candidates, target, 0, new ArrayList<>());
        return ret;
    }
    private List<List<Integer>> ret = new ArrayList<>();
    private void helper(int[] candidates, int target, int start,
ArrayList<Integer>list){
        if(target == 0){
            ret.add(new ArrayList<>(list));
            return;
        }
        for(int i = start; i < candidates.length; i++){
```

```
            if(candidates[i] > target)
                return;
            if(i > start && candidates[i] == candidates[i-1])
                continue;
            list.add(candidates[i]);
            helper(candidates, target-candidates[i], i+1, list);
            list.remove(list.size()-1);
        }
    }
}
```

2.3.2 排列问题

> 给定一个 没有重复 数字的序列，返回其所有可能的全排列。

```
public class _46全排列 {
    private List<List<Integer>> ret;
    public List<List<Integer>> permute(int[] nums) {
        ret = new ArrayList<>();
        helper(nums,0);
        return ret;
    }
    private void helper(int[] nums,int start){
        if(start == nums.length){
            List<Integer> list = new ArrayList<>();
            for(int i : nums)
                list.add(i);
            ret.add(list);
            return;
        }
        for(int i=start;i<nums.length;i++){
            swap(nums,i, start);
            helper(nums,start+1);
            swap(nums,i,start);
        }
    }
    private void swap(int[] nums, int i, int j){
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

变式题

> 给定一个可包含重复数字的序列，返回所有不重复的全排列。

```java
public class _47全排列II {
    private List<List<Integer>> ret;
    private boolean[] used;
    public List<List<Integer>> permuteUnique(int[] nums) {
        Arrays.sort(nums);
        ret = new ArrayList<>();
        used = new boolean[nums.length];
        helper(nums,new ArrayList<>());
        return ret;
    }
    private void helper(int[] nums,List<Integer> list){
        if(list.size() == nums.length){
            ret.add(new ArrayList<>(list));
            return;
        }
        for(int i = 0;i < nums.length; i++){
            if(!used[i]){
                if(i > 0 && nums[i] == nums[i-1] && !used[i-1])
                    continue;
                list.add(nums[i]);
                used[i] = true;
                helper(nums,list);
                list.remove(list.size()-1);
                used[i] = false;
            }
        }
    }
}
```

2.3.3 子集问题

给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。
说明：解集不能包含重复的子集。

```java
public class _78子集 {
    private List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> subsets(int[] nums) {
        dfs(nums, 0, new ArrayList<>());
        return res;
    }
    private void dfs(int[] nums, int start, ArrayList<Integer> list) {
        if(start == nums.length){
            res.add(new ArrayList<>(list));
            return;
        }
        list.add(nums[start]);
        dfs(nums, start+1, list);
        list.remove(list.size()-1);
```

```
            dfs(nums, start+1, list);
        }
    }
```

位运算解法

```
public class _78子集 {
    public List<List<Integer>> subsets(int[] nums) {
        int n = nums.length;
        List<List<Integer>> ret = new ArrayList<>();
        for(int i = 0; i < (1 << n);i++){
            List<Integer> list = new ArrayList<>();
            for(int j = 0; j < n; j++)
                if((i >> j & 1) == 1)
                    list.add(nums[j]);
            ret.add(list);
        }
        return ret;
    }
}
```

变式题

```
给定一个可能包含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。
说明：解集不能包含重复的子集。
```

```
public class _90子集II {
    private List<List<Integer>> res = new ArrayList<>();
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);
        dfs(nums, 0, new ArrayList<>());
        return res;
    }
    private void dfs(int[] nums, int start, ArrayList<Integer> list){
        res.add(new ArrayList<>(list));
        for(int i = start; i < nums.length; i++){
            if(i != start && nums[i] == nums[i-1])
                continue;
            list.add(nums[i]);
            dfs(nums, i+1, list);
            list.remove(list.size()-1);
        }
    }
}
```

2.3.4 N皇后、数独问题

给定一个整数 n，返回所有不同的 n 皇后问题的解决方案。
每一种解法包含一个明确的 n 皇后问题的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。

```java
public class _51N皇后 {
    public List<List<String>> solveNQueens(int n) {
        char[][] board = new char[n][n];
        for (char[] chars : board)
            Arrays.fill(chars,'.');
        col = new boolean[n];
        dia1 = new boolean[2*n-1];
        dia2 = new boolean[2*n-1];
        ret = new ArrayList<>();
        backtrace(n,0,board);
        return ret;
    }
    private void backtrace(int n, int row, char[][] board){
        if (n == row){
            List<String> list = new ArrayList<>(n);
            for (char[] chars : board) {
                list.add(new String(chars));
            }
            ret.add(list);
            return;
        }
        for (int i = 0; i < n; i++) {
            if (!col[i] && !dia1[row+i] && !dia2[row-i+n-1]){
                board[row][i] = 'Q';
                col[i] = true;
                dia1[row+i] = true;
                dia2[row-i+n-1] = true;
                backtrace(n,row+1, board);
                board[row][i] = '.';
                col[i] = false;
                dia1[row+i] = false;
                dia2[row-i+n-1] = false;
            }
        }
    }
    private List<List<String>> ret;
    private boolean[] col;
    private boolean[] dia1;
    private boolean[] dia2;
}
```


变式题

编写一个程序，通过已填充的空格来解决数独问题。

一个数独的解法需遵循如下规则：

数字 1-9 在每一行只能出现一次。
数字 1-9 在每一列只能出现一次。
数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。
空白格用 '.' 表示。

```
class _37解数独 {
    public void solveSudoku(char[][] board) {
        dfs(board, 0, 0);
    }
    private boolean dfs(char[][] board, int x, int y){
        if(x == 9)
            return true;
        if(y == 9)
            return dfs(board, x+1, 0);
        if(board[x][y] != '.')
            return dfs(board, x, y+1);
        for(char i = '1'; i <= '9'; i++){
            board[x][y] = i;
            if(isValid(board,x,y) && dfs(board, x, y+1))
                return true;
            board[x][y] = '.';
        }
        return false;
    }
    private boolean isValid(char[][] board, int x, int y){
        for(int i = 0; i < 9; i++){
            if(i != y && board[x][i] == board[x][y])
                return false;
            if(i != x && board[i][y] == board[x][y])
                return false;
        }
        for(int i = x/3*3; i < x/3*3+3; i++)
            for(int j = y/3*3; j < y/3*3+3; j++)
                if(i != x && j != y && board[x][y] == board[i][j])
                    return false;
        return true;
    }
}
```


2.3.5 复原IP地址

给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式。

```java
public class _93复原IP地址 {
    private List<String> res = new ArrayList<>();
    public List<String> restoreIpAddresses(String s) {
        dfs(s, 0, 0, "");
        return res;
    }
    private void dfs(String s, int start, int n, String temp){
        if(n == 4){
            if(s.length() == start)
                res.add(temp);
            return;
        }
        for(int i = 1;i < 4; i++){
            if(s.length() < i + start)
                return;
            String sub = s.substring(start, i+start);
            int value = Integer.parseInt(sub);
            if(value < 256 && sub.length() == (value + "").length()) {
                dfs(s, start+i, n+1, temp + sub + (n==3 ? "" : "."));
            }
        }
    }
}
```

2.3.6 火柴拼正方形

还记得童话《卖火柴的小女孩》吗？现在，你知道小女孩有多少根火柴，请找出一种能使用所有火柴拼成一个正方形的方法。不能折断火柴，可以把火柴连接起来，并且每根火柴都要用到。输入为小女孩拥有火柴的数目，每根火柴用其长度表示。输出即为是否能用所有的火柴拼成正方形。

```java
public class _473火柴拼正方形 {
    public boolean makesquare(int[] nums) {
        if(nums.length < 4)
            return false;
        int sum = 0;
        for(int num : nums)
            sum += num;
        if(sum % 4 != 0)
            return false;
        Arrays.sort(nums);
        return dfs(nums,nums.length-1,new int[4],sum/4);
    }
    private boolean dfs(int[] nums, int pos, int[] sum, int avg){
        if(pos == -1)
            return sum[0] == avg && sum[1] == avg && sum[2] == avg;
        for(int i = 0; i < 4; i++){
            if(sum[i] + nums[pos] > avg)
```

```
                continue;
            sum[i] += nums[pos];
            if(dfs(nums,pos-1,sum,avg))
                return true;
            sum[i] -= nums[pos];
        }
        return false;
    }
}
```

2.3.7 搜索剪枝

给定一个二维网格 board 和一个字典中的单词列表 words，找出所有同时在二维网格和字典中出现的单词。
单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。
同一个单元格内的字母在一个单词中不允许被重复使用。

```
public class _212单词搜索II {
    public List<String> findWords(char[][] board, String[] words) {
        Trie trie = new Trie();
        for(String s : words)
            trie.add(s);
        m = board.length;
        n = board[0].length;
        visited = new boolean[m][n];
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                String s = String.valueOf(board[i][j]);
                if(trie.isPrefix(s)){
                    visited[i][j] = true;
                    dfs(board, s, trie, i, j);
                    visited[i][j] = false;
                }
            }
        }
        List<String> res = new ArrayList<>(ret);
        res.sort(String::compareTo);
        return res;
    }
    private int m;
    private int n;
    private boolean[][] visited;
    private Set<String> ret = new HashSet<>();
    private int[][] dir = {{1,0},{-1,0},{0,1},{0,-1}};
    private void dfs(char[][] board, String s, Trie trie, int x, int
y) {
        if(trie.contains(s)){
```

```
                ret.add(s);
            }
            for (int[] ints : dir) {
                int nx = x + ints[0];
                int ny = y + ints[1];
                if (0 <= nx && nx < m && 0 <= ny && ny < n && !
visited[nx][ny] && trie.isPrefix(s + board[nx][ny])) {
                    visited[nx][ny] = true;
                    dfs(board, s + board[nx][ny], trie, nx, ny);
                    visited[nx][ny] = false;
                }
            }
        }
    }
    class Trie {
        private class Node{
            public boolean isWord;
            public TreeMap<Character, Node> next;
            public Node(boolean isWord){
                this.isWord = isWord;
                next = new TreeMap<>();
            }
            public Node(){
                this(false);
            }
        }
        private Node root;
        private int size;
        public Trie(){
            root = new Node();
            size = 0;
        }
        public void add(String word){
            Node cur = root;
            for (int i=0;i<word.length();i++){
                char c = word.charAt(i);
                if(cur.next.get(c) == null)
                    cur.next.put(c, new Node());
                cur = cur.next.get(c);
            }
            if(!cur.isWord) {
                cur.isWord = true;
                size++;
            }
        }
        public boolean contains(String word){
            Node cur = root;
            for (int i=0;i<word.length();i++){
                char c = word.charAt(i);
```

```
            if(cur.next.get(c) == null)
                return false;
            cur = cur.next.get(c);
        }
        return cur.isWord;
    }
    public boolean isPrefix(String prefix){
        Node cur = root;
        for (int i=0;i<prefix.length();i++){
            char c = prefix.charAt(i);
            if(cur.next.get(c) == null)
                return false;
            cur = cur.next.get(c);
        }
        return true;
    }
}
```

2.4排序

2.4.1 快速排序思想应用

2.4.1.1 三路快速排序技巧

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

注意:
不能使用代码库中的排序函数来解决这道题。

```
class _75颜色分类 {
    public void sortColors(int[] nums) {
        int zero = -1;
        int two = nums.length;
        for(int i = 0;i < two;){
            if(nums[i] == 1)
                i++;
            else if(nums[i] == 2)
                swap(nums, i, --two);
            else
                swap(nums,++zero, i++);
        }
    }
    private void swap(int[] nums, int x, int y){
        int temp = nums[x];
        nums[x] = nums[y];
        nums[y] = temp;
```

```
    }
}
```

2.4.1.2 Top K

> 在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

```
class _215数组中的第K个最大元素 {
    public int findKthLargest(int[] nums, int k) {
        return findKthLargest(nums, 0, nums.length-1, k);
    }
    private int findKthLargest(int[] nums, int left, int right, int k)
{
        int p = partition(nums, left, right);
        if( k == p-left+1)
            return nums[p];
        if(k < p-left+1)
            return findKthLargest(nums, left, p-1, k);
        return findKthLargest(nums, p+1, right, k-p-1+left);
    }
    private int partition(int[] nums,int left,int right){
        int guard = nums[left];
        int index = left;
        for(int i=left+1;i<=right;i++){
            if(nums[i] > guard){
                index++;
                int temp = nums[i];
                nums[i] = nums[index];
                nums[index] = temp;
            }
        }
        nums[left] = nums[index];
        nums[index] = guard;
        return index;
    }
}
```

变式题

> 输入整数数组 arr ，找出其中最小的 k 个数。不需要有序。

```
class 面40最小的K个数 {
    public int[] getLeastNumbers(int[] arr, int k) {
        if(k == 0)
            return new int[]{};
        int pos = findKthLargest(arr, k);
        int[] ret = new int[pos+1];
```

```
        for(int i = 0; i <= pos; i++)
            ret[i] = arr[i];
        return ret;
    }
    public int findKthLargest(int[] nums, int k) {
        return findKthLargest(nums, 0, nums.length-1, k);
    }
    private int findKthLargest(int[] nums, int left, int right, int k)
{
        int p = partition(nums, left, right);
        if( k == p-left+1)
            return p;
        if(k < p-left+1)
            return findKthLargest(nums, left, p-1, k);
        return findKthLargest(nums, p+1, right, k-p-1+left);
    }
    private int partition(int[] nums,int left,int right){
        int guard = nums[left];
        int index = left;
        for(int i=left+1;i<=right;i++){
            if(nums[i] < guard){
                index++;
                int temp = nums[i];
                nums[i] = nums[index];
                nums[index] = temp;
            }
        }
        nums[left] = nums[index];
        nums[index] = guard;
        return index;
    }
}
```

2.4.2 归并排序思想应用

2.4.2.1 逆序数

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

```
class 面51数组中的逆序对 {
    private int[] temp;
    public int reversePairs(int[] nums) {
        if(nums.length == 0)
            return 0;
        temp = new int[nums.length];
        return mergeCount(nums, 0, nums.length-1);
    }
    private int mergeCount(int[] nums, int left, int right){
```

```
        if(left == right)
            return 0;
        int mid = (left+right)/2;
        int count = mergeCount(nums,left,mid) +
mergeCount(nums,mid+1,right);
        int i = left;
        int j = mid+1;
        int index = left;
        while(i <= mid && j <= right){
            if(nums[i] <= nums[j]){
                count += j-(mid+1);
                temp[index++] = nums[i++];
            }else{
                temp[index++] = nums[j++];
            }
        }
        while(i <= mid){
            temp[index++] = nums[i++];
            count += right-mid;
        }
        while(j <= right){
            temp[index++] = nums[j++];
        }
        for(int k = left; k <= right; k++)
            nums[k] = temp[k];
        return count;
    }
}
```

变式题

给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们就将 (i, j) 称作一个重要翻转对。

你需要返回给定数组中的重要翻转对的数量。

```
class _493翻转对 {
    private int[] temp;
    public int reversePairs(int[] nums) {
        int n = nums.length;
        temp = new int[n];
        return mergeCount(nums, 0, n-1);
    }
    private int mergeCount(int[] nums, int start, int end){
        if(start >= end)
            return 0;
        int mid = start + end >> 1;
        int left = mergeCount(nums, start, mid);
```

```
        int right = mergeCount(nums, mid+1, end);
        int midCount = 0;
        for(int i = start; i <= end; i++)
            temp[i] = nums[i];
        int i = start, j = mid+1;
        while(i <= mid && j <= end){
            if((long)nums[i] > (long)2*nums[j]){
                midCount += mid - i + 1;
                j++;
            }else{
                i++;
            }
        }
        i = start;
        j = mid + 1;
        for(int k = start; k <= end;k++){
            if(i > mid){
                nums[k] = temp[j++];
            }else if(j > end){
                nums[k] = temp[i++];
            }else if(temp[i]<temp[j]){
                nums[k] = temp[i++];
            }else{
                nums[k] = temp[j++];
            }
        }
        return left + right + midCount;
    }
}
```

给定一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

```
public class _315计算右侧小于当前元素的个数 {
    private int[] temp;
    private int[] counter;
    private int[] indexes;
    public List<Integer> countSmaller(int[] nums) {
        List<Integer> res = new ArrayList<>();
        int len = nums.length;
        if (len == 0)
            return res;
        temp = new int[len];
        counter = new int[len];
        indexes = new int[len];
        for (int i = 0; i < len; i++)
            indexes[i] = i;
        mergeAndCountSmaller(nums, 0, len - 1);
```

```
        for (int i = 0; i < len; i++)
            res.add(counter[i]);
        return res;
    }
    private void mergeAndCountSmaller(int[] nums, int l, int r) {
        if (l == r)
            return;
        int mid = l + (r - l) / 2;
        mergeAndCountSmaller(nums, l, mid);
        mergeAndCountSmaller(nums, mid + 1, r);
        if (nums[indexes[mid]] > nums[indexes[mid + 1]])
            mergeOfTwoSortedArrAndCountSmaller(nums, l, mid, r);
    }
    private void mergeOfTwoSortedArrAndCountSmaller(int[] nums, int l,
int mid, int r) {
        for (int i = l; i <= r; i++)
            temp[i] = indexes[i];
        int i = l;
        int j = mid + 1;
        for (int k = l; k <= r; k++) {
            if (i > mid) {
                indexes[k] = temp[j];
                j++;
            } else if (j > r) {
                indexes[k] = temp[i];
                i++;
                counter[indexes[k]] += (r - mid);
            } else if (nums[temp[i]] <= nums[temp[j]]) {
                indexes[k] = temp[i];
                i++;
                counter[indexes[k]] += (j - mid - 1);
            } else {
                indexes[k] = temp[j];
                j++;
            }
        }
    }
}
```

二叉搜索树解法

```
class _315计算右侧小于当前元素的个数 {
    public List<Integer> countSmaller(int[] nums) {
        Integer[] ret = new Integer[nums.length];
        for (int i = 0; i < nums.length; i++)
            ret[i] = 0;
        List<Integer> list = new ArrayList<>();
        TreeNode root = null;
        for (int i = nums.length - 1; i >= 0; i--) {
```

```
            root = insert(root, new TreeNode(nums[i]), ret, i);
        }
        return Arrays.asList(ret);
    }
    public TreeNode insert(TreeNode root, TreeNode node, Integer[]
ret, int i) {
        if (root == null) {
            root = node;
            return root;
        }
        if (root.val >= node.val) {
            root.count++;
            root.left = insert(root.left, node, ret, i);
        } else {
            ret[i] += root.count + 1;
            root.right = insert(root.right, node, ret, i);
        }
        return root;
    }
}
class TreeNode {
    int val;
    int count;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
        count = 0;
    }
}
```

2.5动态规划

2.5.1 序列型问题

2.5.1.1 正则匹配

```
给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹
配。

'.' 匹配任意单个字符
'*' 匹配零个或多个前面的那一个元素
所谓匹配，是要涵盖 整个 字符串 s的，而不是部分字符串。

说明:
s 可能为空，且只包含从 a-z 的小写字母。
p 可能为空，且只包含从 a-z 的小写字母，以及字符 . 和 *。
```

```
class _10正则表达式匹配 {
```

```
    public boolean isMatch(String s, String p) {
        int sLen = s.length(), pLen = p.length();
        boolean[][] dp = new boolean[sLen+1][pLen+1];
        dp[0][0] = true;
        for(int i = 0; i <= sLen; i++) {
            for(int j = 1; j <= pLen; j++) {
                if(p.charAt(j-1) == '*') {
                    dp[i][j] = dp[i][j-2] || (i > 0 && (s.charAt(i-1)
== p.charAt(j-2) ||
                            p.charAt(j-2) == '.') && dp[i-1][j]);
                }else {
                    dp[i][j] = i > 0 && (s.charAt(i-1) == p.charAt(j-
1) || p.charAt(j-1) == '.')
                            && dp[i-1][j-1];
                }
            }
        }
        return dp[sLen][pLen];
    }
}
```

变式题

> 给定一个字符串 (s) 和一个字符模式 (p) ，实现一个支持 '?' 和 '*' 的通配符匹配。
>
> '?' 可以匹配任何单个字符。
> '*' 可以匹配任意字符串（包括空字符串）。

```
class _44通配符匹配 {
    public boolean isMatch(String s, String p) {
        int m = s.length();
        int n = p.length();
        boolean[][] dp = new boolean[m + 1][n + 1];
        dp[0][0] = true;
        for (int i = 1; i <= n; i++) {
            dp[0][i] = dp[0][i - 1] && p.charAt(i - 1) == '*';
        }
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (s.charAt(i - 1) == p.charAt(j - 1) || p.charAt(j -
1) == '?') {
                    dp[i][j] = dp[i - 1][j - 1];
                } else if (p.charAt(j - 1) == '*') {
                    dp[i][j] = dp[i][j - 1] || dp[i - 1][j];
                }
            }
        }
        return dp[m][n];
```

```
    }
}
```

2.5.1.2 回文串

> 给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

```
class _5最长回文子串 {
    public String longestPalindrome(String s) {
        int n = s.length();
        if(n == 0)
            return "";
        int left = 0;
        int right = 0;
        boolean[][] dp = new boolean[n][n];
        for (int i = n - 1; i >= 0; i--) {
            for (int j = i; j < n; j++) {
                dp[i][j] = s.charAt(i) == s.charAt(j) && (j - i < 2 ||
dp[i + 1][j - 1]);
                if (dp[i][j] &&  j - i > right - left) {
                    left = i;
                    right = j;
                }
            }
        }
        return s.substring(left, right+1);
    }
}
```

变式题

> 给定一个字符串s，找到其中最长的回文子序列。可以假设s的最大长度为1000。

```
class _516最长回文子序列 {
    public int longestPalindromeSubseq(String s) {
        int n = s.length();
        int[][] dp = new int[n][n];
        for(int i = 0; i < n; i++)
            dp[i][i] = 1;
        for(int i = n-1; i >= 0; i--){
            for(int j = i+1; j < n; j++){
                if(s.charAt(i) == s.charAt(j))
                    dp[i][j] = dp[i+1][j-1] + 2;
                else
                    dp[i][j] = Math.max(dp[i+1][j],dp[i][j-1]);
            }
        }
        return dp[0][n-1];
```

```
    }
}
```

2.5.1.3 编辑距离

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符
删除一个字符

```
class _72编辑距离 {
    public int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();
        int[][] dp = new int[m+1][n+1];
        for(int i=0;i<=n;i++)
            dp[0][i] = i;
        for(int j=0;j<=m;j++)
            dp[j][0] = j;
        for(int i=1;i<=word1.length();i++){
            for(int j=1;j<=word2.length();j++){
                if(word1.charAt(i-1)==word2.charAt(j-1))
                    dp[i][j] = dp[i-1][j-1];
                else
                    dp[i][j] = Math.min(dp[i-1][j],Math.min(dp[i][j-
1],dp[i-1][j-1]))+1;
            }
        }
        return dp[m][n];
    }
}
```

2.5.1.4 最长公共子序列

给定两个字符串 text1 和 text2，返回这两个字符串的最长公共子序列的长度。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。
例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。两个字符串的「公共子序列」是这两个字符串所共同拥有的子序列。

若这两个字符串没有公共子序列，则返回 0。

```
class _1143最长公共子序列 {
    public int longestCommonSubsequence(String a, String b) {
```

```
        int m = a.length();
        int n = b.length();
        int[][] dp = new int[m+1][n+1];
        int[][] path = new int[m+1][n+1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (a.charAt(i-1) == b.charAt(j-1)){
                    dp[i][j] = dp[i-1][j-1] + 1;
                    path[i][j] = 1;
                }else if(dp[i-1][j] > dp[i][j-1]) {
                    dp[i][j] = dp[i-1][j];
                    path[i][j] = 2;
                }else{
                    dp[i][j] = dp[i][j-1];
                    path[i][j] = 3;
                }
            }
        }
        /* for debug only
        System.out.println(dp[m][n]);
        StringBuilder sb = new StringBuilder(dp[m][n]);
        while (m != 0 && n != 0) {
            if (path[m][n] == 1) {
                sb.append(a.charAt(m-1));
                m--;
                n--;
            } else if (path[m][n] == 2) {
                m--;
            } else {
                n--;
            }
        }
        System.out.println(sb.reverse());*/
        return dp[m][n];
    }
}
```

2.5.2 背包问题

2.5.2.1 01背包

> 给定一个只包含正整数的非空数组。是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
>
> 注意:
> 每个数组中的元素不会超过 100
> 数组的大小不会超过 200

```
public class _416分割等和子集 {
```

```
    public boolean canPartition(int[] nums) {
        int sum = 0;
        for(int n : nums)
            sum += n;
        if(sum % 2 != 0)
            return false;
        int c = sum/2;
        boolean[] dp = new boolean[c+1];
        for(int i = 0; i <= c; i++)
            dp[i] = (nums[0] == i);
        for(int i = 1; i < nums.length; i++)
            for(int j = c; j >= nums[i]; j--)
                dp[j] |= dp[j-nums[i]];
        return dp[c];
    }
}
```

变式题

在计算机界中，我们总是追求用有限的资源获取最大的收益。
现在，假设你分别支配着 m 个 0 和 n 个 1。另外，还有一个仅包含 0 和 1 字符串的数组。
你的任务是使用给定的 m 个 0 和 n 个 1，找到能拼出存在于数组中的字符串的最大数量。每个 0 和 1 至多被使用一次。

```
public class _474一和零 {
    public int findMaxForm(String[] strs, int m, int n) {
        int[][] dp = new int[m+1][n+1];
        for(String s : strs){
            int zeros = 0;
            int ones = 0;
            for(char c : s.toCharArray()){
                if(c == '0')
                    zeros ++;
                else
                    ones ++;
            }
            for(int i = m; i >= zeros; i--){
                for(int j = n; j >= ones; j--){
                    dp[i][j] = Math.max(dp[i][j],dp[i-zeros][j-
ones]+1);
                }
            }
        }
        return dp[m][n];
    }
}
```

2.5.2.2 完全背包

> 给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

```
public class _322零钱兑换 {
    public int coinChange(int[] coins, int amount) {
        int max = amount + 1;
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, max);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++)
            for (int coin : coins)
                if (coin <= i)
                    dp[i] = Math.min(dp[i], dp[i - coin] + 1);
        return dp[amount] > amount ? -1 : dp[amount];
    }
}
```

变式题

> 给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

```
public class _518零钱兑换II {
    public int change(int amount, int[] coins) {
        int[] dp = new int[amount + 1];
        dp[0] = 1;
        for (int coin : coins)
            for (int x = coin; x < amount + 1; ++x)
                dp[x] += dp[x - coin];
        return dp[amount];
    }
}
```

变式题

> 给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。

```
public class _377组合总和IV {
    public int combinationSum4(int[] nums, int target) {
        int[] dp = new int[target+1];
        dp[0] = 1;
        for(int j = 0; j <= target; j++)
            for (int num : nums)
                if (j >= num)
                    dp[j] += dp[j - num];
        return dp[target];
    }
}
```

2.5.2.3 多重背包

```
public class MultiBag {
    public int getMaxProfit(int money, int[] v, int[] w, int[] k) {
        List<Integer> value = new ArrayList<>();
        List<Integer> weight = new ArrayList<>();
        for (int i = 0; i < v.length; i++) {
            for (int j = 1; j <= k[i]; j <<= 1) {
                value.add(j * v[i]);
                weight.add(j * w[i]);
                k[i] -= j;
            }
            if (k[i] > 0){
                value.add(k[i] * v[i]);
                weight.add(k[i] * w[i]);
            }
        }
        int[] dp = new int[money+1];
        for (int i = 0; i < value.size(); i++)
            for (int j = money; j >= value.get(i); j--)
                dp[j] = Math.max(dp[j],dp[j-
value.get(i)]+weight.get(i));
        return dp[money];
    }
}
```

2.5.3 博弈问题

给定一个表示分数的非负整数数组。 玩家1从数组任意一端拿取一个分数，随后玩家2继续从剩余数组任意一端拿取分数，然后玩家1拿，……。
每次一个玩家只能拿取一个分数，分数被拿取之后不再可取。直到没有剩余分数可取时游戏结束。最终获得分数总和最多的玩家获胜。
给定一个表示分数的数组，预测玩家1是否会成为赢家。你可以假设每个玩家的玩法都会使他的分数最大化。

```
public class _486预测赢家 {
```

```
    //解法1
    public boolean PredictTheWinner(int[] nums) {
        int n = nums.length;
        int[][] dp = new int[n][n]; //dp表示先手最多赢多少
        int sum = 0;
        for(int i : nums)
            sum += i;
        for(int i = n-1; i >= 0; i--){
            for(int j = i; j < n; j++) {
                if(j == i){
                    dp[i][j] = nums[i];
                    continue;
                }
                if(j == i+1){
                    dp[i][j] = Math.max(nums[i],nums[j]);
                    continue;
                }
                dp[i][j] = Math.max(nums[i]+Math.min(dp[i+2]
[j],dp[i+1][j-1]),
                        nums[j]+Math.min(dp[i+1][j-1],dp[i][j-2]));
            }
        }
        return dp[0][n-1] >= sum - dp[0][n-1];
    }
    //解法2
    public boolean PredictTheWinnerFirstAndSecond(int[] nums) {
        int n = nums.length;
        int[][] f = new int[n][n];
        int[][] s = new int[n][n];
        for(int i = n-1; i >= 0; i--){
            f[i][i] = nums[i];
            for(int j = i+1; j < n; j ++){
                f[i][j] = Math.max(nums[i]+s[i+1][j],nums[j]+s[i][j-
1]);
                s[i][j] = Math.min(f[i+1][j],f[i][j-1]);
            }
        }
        return f[0][n-1] >= s[0][n-1];
    }
    //递归解法：存在重复计算
    public boolean PredictTheWinnerRecursion(int[] nums) {
        return first(nums,0,nums.length-1) >=
second(nums,0,nums.length-1);
    }
    private int first(int[] nums, int l, int r) {
        if(l == r)
            return nums[l];
        return
Math.max(nums[l]+second(nums,l+1,r),nums[r]+second(nums,l,r-1));
```

```
    }
    private int second(int[] nums, int l, int r) {
        if(l == r)
            return 0;
        return Math.min(first(nums, l+1, r),first(nums, l, r-1));
    }
}
```

2.5.4 买卖股票问题

给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。
如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。
注意：你不能在买入股票前卖出股票。

```
class _121买卖股票的最佳时机 {
    //dp解法
    public int maxProfit0(int[] prices) {
        int n = prices.length;
        if( n == 0)
            return 0;
        int[][] dp = new int[n][2];
        dp[0][1] = -prices[0];
        for(int i = 1; i < n; i++){
            dp[i][0] = Math.max(dp[i-1][0],dp[i-1][1]+prices[i]);
            dp[i][1] = Math.max(dp[i-1][1],-prices[i]);
        }
        return dp[n-1][0];
    }
    //dp优化
    public int maxProfit1(int[] prices) {
        int n = prices.length;
        int dp0 = 0;
        int dp1 = Integer.MIN_VALUE;
        for(int i = 0; i < n; i++){
            dp0 = Math.max(dp0,dp1+prices[i]);
            dp1= Math.max(dp1,-prices[i]);
        }
        return dp0;
    }
    //子序和解法
    public int maxProfit3(int[] prices) {
        int n = prices.length;
        if(n < 2)
            return 0;
        int[] arr = new int[n-1];
        for(int i = 0; i < n-1; i++)
            arr[i] = prices[i+1]-prices[i];
```

```
        int res = maxSubArray(arr);
        return Math.max(res, 0);
    }
    public int maxSubArray(int[] nums) {
        int dp = nums[0];
        int ret = nums[0];
        for(int i = 1; i < nums.length; i++){
            if (dp > 0)
                dp += nums[i];
            else
                dp = nums[i];
            ret = Math.max(ret, dp);
        }
        return ret;
    }
    //波峰波谷解法
    public int maxProfit4(int[] prices) {
        int minPrice = Integer.MAX_VALUE;
        int maxProfit = 0;
        for(int i=0;i<prices.length;i++){
            if(prices[i]<minPrice)
                minPrice = prices[i];
            else if(prices[i]-minPrice>maxProfit)
                maxProfit = prices[i]-minPrice;
        }
        return maxProfit;
    }
}
```

变式题

```
给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。
设计一个算法来计算你所能获取的最大利润。你可以尽可能地完成更多的交易（多次买卖一支股票）。
注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
```

```
class _122买卖股票的最佳时机II {
    //贪心算法
    public int maxProfit(int[] prices) {
        int res = 0;
        for(int i = 1; i < prices.length; i++)
            if(prices[i] > prices[i-1])
                res += prices[i]-prices[i-1];
        return res;
    }
    //dp解法
    public int maxProfit2(int[] prices) {
        int n = prices.length;
```

```
        if(n == 0)
            return 0;
        int[][] dp = new int[n][2];
        dp[0][1] = - prices[0];
        for(int i = 1; i < n; i++){
            dp[i][0] = Math.max(dp[i-1][1]+prices[i],dp[i-1][0]);
            dp[i][1] = Math.max(dp[i-1][1],dp[i-1][0]-prices[i]);
        }
        return dp[n-1][0];
    }
    //dp优化
    public int maxProfit3(int[] prices) {
        int n = prices.length;
        int dp0 = 0;
        int dp1 = Integer.MIN_VALUE;
        for(int i = 0;i < n; i++){
            int temp = dp0;
            dp0 = Math.max(dp0,dp1+prices[i]);
            dp1 = Math.max(dp1,temp-prices[i]);
        }
        return dp0;
    }
}
```

变式题

> 给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。
> 设计一个算法来计算你所能获取的最大利润。你最多可以完成 两笔 交易。
> 注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

```
class _123买卖股票的最佳时机III {
    //dp
    public int maxProfit(int[] prices) {
        int n = prices.length;
        if(n < 2)
            return 0;
        int[][][] dp = new int[n][3][2];
        dp[0][1][1] = - prices[0];
        dp[0][2][1] = - prices[0];
        for(int i = 1; i < prices.length; i++){
            for(int j = 1; j < 3; j++){
                dp[i][j][0] = Math.max(dp[i-1][j][0],dp[i-1][j]
[1]+prices[i]);
                dp[i][j][1] = Math.max(dp[i-1][j][1],dp[i-1][j-1][0]-
prices[i]);
            }
        }
        return dp[n-1][2][0];
```

```
    }
    //dp优化
    public int maxProfit2(int[] prices) {
        int n = prices.length;
        int dp10 = 0;
        int dp11 = Integer.MIN_VALUE;
        int dp20 = 0;
        int dp21 = Integer.MIN_VALUE;
        for(int price : prices){
            dp20 = Math.max(dp20,dp21+price);
            dp21 = Math.max(dp21,dp10-price);
            dp10 = Math.max(dp10,dp11+price);
            dp11 = Math.max(dp11,-price);
        }
        return dp20;
    }
}
```

变式题

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。

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

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

```
class _309最佳买卖股票时机含冷冻期{
  //dp解法
  public int maxProfit(int[] prices) {
        int n = prices.length;
        if(n < 2)
            return 0;
        int[][] dp = new int[n][3];
        dp[0][1] = -prices[0];
        for(int i = 1; i < n; i++){
            dp[i][0] = Math.max(dp[i-1][0],dp[i-1][2]);
            dp[i][1] = Math.max(dp[i-1][1],dp[i-1][0]-prices[i]);
            dp[i][2] = dp[i-1][1] + prices[i];
        }
        return Math.max(dp[n-1][0],dp[n-1][2]);
  }
  //dp优化
  public int maxProfit(int[] prices) {
        int n = prices.length;
        int dp0 = 0;
        int dp1 = Integer.MIN_VALUE;
```

```
        int dp2 = 0;
        for(int i=0;i<n;i++){
            int temp0 = dp0;
            int temp1 = dp1;
            dp0 = Math.max(dp2,dp0);
            dp1 = Math.max(temp0-prices[i],dp1);
            dp2 = temp1+prices[i];
        }
        return Math.max(dp0, dp2);
    }
}
```

变式题

> 给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。
>
> 设计一个算法来计算你所能获取的最大利润。你最多可以完成 k 笔交易。
>
> 注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

```
class _188买卖股票的最佳时机IV {
    public int maxProfit(int k, int[] prices) {
        int n = prices.length;
        if(k > n/2)
            return maxProfitInfinite(prices);
        int[][][] dp = new int[n][k+1][2];
        for(int i = 0; i < n; i++){
            for(int j = 1; j <= k; j++){
                if(i == 0){
                    dp[i][j][0] = 0;
                    dp[i][j][1] = - prices[i];
                    continue;
                }
                dp[i][j][0] = Math.max(dp[i-1][j][0],dp[i-1][j]
[1]+prices[i]);
                dp[i][j][1] = Math.max(dp[i-1][j][1],dp[i-1][j-1][0]-
prices[i]);
            }
        }
        return dp[n-1][k][0];
    }
    public int maxProfitInfinite(int[] prices) {
        int res = 0;
        for(int i = 1; i < prices.length; i++)
            if(prices[i] > prices[i-1])
                res += prices[i]-prices[i-1];
        return res;
    }
```

```
}
```

变式题

> 给定一个整数数组 prices，其中第 i 个元素代表了第 i 天的股票价格 ；非负整数 fee 代表了交易股票的手续费用。
>
> 你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。
>
> 返回获得利润的最大值。
>
> 注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

```java
class _714买卖股票的最佳时机含手续费 {
    public int maxProfit(int[] prices, int fee) {
        int n = prices.length;
        int dp0 = 0;
        int dp1 = Integer.MIN_VALUE;
        for(int i = 0; i < n; i++){
            int temp = dp0;
            dp0 = Math.max(dp0,dp1+prices[i]);
            dp1 = Math.max(dp1,temp-prices[i]-fee);
        }
        return dp0;
    }
}
```

2.5.5 最大子序和

> 给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

```java
class _53最大子序和 {
    public int maxSubArray(int[] nums) {
        int dp = nums[0];
        int ret = nums[0];
        for(int i = 1; i < nums.length; i++){
            if (dp > 0)
                dp += nums[i];
            else
                dp = nums[i];
            ret = Math.max(ret, dp);
        }
        return ret;
    }
}
```

分治解法

```
public class Solution{
    public int maxSubArray(int[] nums) {
        return maxSubArray(nums, 0, nums.length-1);
    }
    private int maxSubArray(int[] nums, int left, int right) {
        if(left == right)
            return nums[left];
        int mid = (left + right)/2;
        int leftSum = maxSubArray(nums, left, mid);
        int rightSum = maxSubArray(nums, mid+1, right);
        int sum = 0;
        int crossLeft = Integer.MIN_VALUE;
        int crossRight = Integer.MIN_VALUE;
        for(int i = mid+1; i <= right; i++){
            sum += nums[i];
            crossRight = Math.max(crossRight, sum);
        }
        sum = 0;
        for(int i = mid; i >= left; i--) {
            sum += nums[i];
            crossLeft = Math.max(crossLeft, sum);
        }
        return
Math.max(Math.max(leftSum,rightSum),crossLeft+crossRight);
    }
}
```

变式题

给定一个正整数和负整数组成的 N × M 矩阵，编写代码找出元素总和最大的子矩阵。
返回一个数组 [r1, c1, r2, c2]，其中 r1, c1 分别代表子矩阵左上角的行号和列号，
r2, c2 分别代表右下角的行号和列号。若有多个满足条件的子矩阵，返回任意一个均可。

```
public class MaxMatrix {
    public int[] getMaxMatrix(int[][] matrix) {
        int max = Integer.MIN_VALUE;
        int n = matrix.length;
        int m = matrix[0].length;
        int[] arr = new int[m];
        int[] ret = new int[4];
        for (int i = 0; i < n; i++) {
            Arrays.fill(arr,0);
            for (int j = i; j < n; j++) {
                for (int k = 0; k < m; k++) {
                    arr[k] += matrix[j][k];
```

```
                }
                int[] result = maxSubArrayRange(arr);
                if(result[2] > max){
                    max = result[2];
                    ret[0] = i;
                    ret[1] = result[0];
                    ret[2] = j;
                    ret[3] = result[1];
                }
            }
        }
        return ret;
    }
    //最大子序和模版，返回3个值的数组：左端点，右端点，最大子序和
    public int[] maxSubArrayRange(int[] nums) {
        int[] ret = new int[3];
        ret[2] = nums[0];
        int dp = nums[0];
        int begin = 0;
        for (int i = 1; i < nums.length; i++) {
            if(dp > 0)
                dp += nums[i];
            else{
                dp = nums[i];
                begin = i;
            }
            if(dp > ret[2]){
                ret[2] = dp;
                ret[1] = i;
                ret[0] = begin;
            }
        }
        return ret;
    }
}
```

2.5.6 最长上升子序列

> 给定一个无序的整数数组，找到其中最长上升子序列的长度。

```
public class _300最长递增子序列 {
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;
        if(n == 0)
            return 0;
        int[] dp = new int[n];
        int res = 1;
```

```
        for(int i = 0; i < n; i++){
            dp[i] = 1;
            for(int j = 0; j < i; j++){
                if(nums[i] > nums[j])
                    dp[i] = Math.max(dp[i],dp[j]+1);
            }
            res = Math.max(dp[i], res);
        }
        return res;
    }
    //二分查找优化，O(nlog(n))
    public int lengthOfLISEffective(int[] nums) {
        int[] res = new int[nums.length];
        int len = 0;
        for (int num: nums) {
            int idx = Arrays.binarySearch(res, 0, len, num);
            idx = idx < 0? -idx - 1: idx;
            res[idx] = num;
            if (idx == len) {
                len++;
            }
        }
        return len;
    }
}
```

变式题

给定一些标记了宽度和高度的信封，宽度和高度以整数对形式 (w, h) 出现。当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。请计算最多能有多少个信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。
说明:
不允许旋转信封。

```
public class _354俄罗斯套娃信封问题 {
    public int maxEnvelopes(int[][] envelopes) {
        Arrays.sort(envelopes, (a,b)->a[0]-b[0] == 0 ? b[1]-a[1] :
a[0]-b[0]);
        int[] arr = new int[envelopes.length];
        int res = 0;
        for (int[] i : envelopes) {
            int idx = Arrays.binarySearch(arr, 0, res, i[1]); //查看
Arrays源码说明
            idx = idx < 0 ? - idx - 1: idx;
            arr[idx] = i[1];
            if (idx == res)
                res++;
```

```
        }
        return res;
    }
}
```

变式题

> N位同学站成一排，音乐老师要请其中的(N-K)位同学出列，使得剩下的K位同学不交换位置就能排成合唱队形。
> 合唱队形是指这样的一种队形：设K位同学从左到右依次编号为1，2，…，K，他们的身高分别为T1，T2，…，TK，T1 < T2 < … < Ti , Ti > Ti+1 > … > TK (1 <= i <= K)。
> 你的任务是，已知所有N位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

```
public class SingQueue{
    public int minimumRemove(int[] arr) {
        int n = arr.length;
        int[] left = new int[n];
        int[] right = new int[n];
        int ans = n;
        for (int i = 0; i < n; i++) {
            left[i] = 1;
            for (int j = 0; j < i; j++) {
                if (arr[i] > arr[j]){
                    left[i] = Math.max(left[i], left[j]+1);
                }
            }
        }
        for (int i = n-1; i >= 0; i--) {
            right[i] = 1;
            for (int j = i+1; j < n; j++) {
                if (arr[i] > arr[j]){
                    right[i] = Math.max(right[i], right[j]+1);
                }
            }
        }
        for (int i = 0; i < n; i++)
            ans = Math.min(ans,n-left[i]-right[i]+1);
        return ans;
    }
}
```


2.5.7 正方形问题

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

```
public class _221最大正方形 {
    public int maximalSquare(char[][] matrix) {
```

```
        int maxSide = 0;
        if (matrix == null || matrix.length == 0 || matrix[0].length
== 0) {
            return maxSide;
        }
        int rows = matrix.length, columns = matrix[0].length;
        int[][] dp = new int[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                if (matrix[i][j] == '1') {
                    if (i == 0 || j == 0) {
                        dp[i][j] = 1;
                    } else {
                        dp[i][j] = Math.min(Math.min(dp[i - 1][j],
dp[i][j - 1]), dp[i - 1][j - 1]) + 1;
                    }
                    maxSide = Math.max(maxSide, dp[i][j]);
                }
            }
        }
        return maxSide * maxSide;
    }
}
```

变式题

> 给你一个 m * n 的矩阵，矩阵中的元素不是 0 就是 1，请你统计并返回其中完全由 1 组成的 正方形 子矩阵的个数。

```
public class _1277统计全为1的正方形子矩阵 {
    public int countSquares(int[][] matrix) {
        int m = matrix.length;
        int n = matrix[0].length;
        int[][] dp = new int[m][n];
        int res = 0;
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                if(matrix[i][j] == 1){
                    if(i == 0 || j == 0)
                        dp[i][j] = 1;
                    else{
                        dp[i][j] = Math.min(dp[i-1][j-
1],Math.min(dp[i][j-1],dp[i-1][j]))+1;
                    }
                    res += dp[i][j];
                }
            }
        }
```

```
        return res;
    }
}
```

2.5.8 单词拆分问题

> 给定一个非空字符串 s 和一个包含非空单词列表的字典 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。
>
> 说明：
> 拆分时可以重复使用字典中的单词。
> 你可以假设字典中没有重复的单词。

```
public class _139单词拆分 {
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> wordDictSet = new HashSet<>(wordDict);
        boolean[] dp = new boolean[s.length() + 1];
        dp[0] = true;
        for (int i = 1; i <= s.length(); i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && wordDictSet.contains(s.substring(j, i)))
{
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.length()];
    }
}
```

广度优先遍历解法

```
class _139单词拆分 {
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> set = new HashSet<>(wordDict);
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(0);
        boolean[] visited = new boolean[s.length()];
        while(!queue.isEmpty()){
            int start = queue.poll();
            System.out.println(queue);
            if(start == s.length())
                return true;
            if(visited[start])
                continue;
            for(int i=start+1;i<=s.length();i++){
                String sub = s.substring(start, i);
```

```
                if(wordDict.contains(sub)){
                    queue.offer(i);
                }
            }
            visited[start] = true;
        }
        return false;
    }
}
```

变式题

给定一个非空字符串 s 和一个包含非空单词列表的字典 wordDict，在字符串中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这些可能的句子。
说明：

分隔时可以重复使用字典中的单词。
你可以假设字典中没有重复的单词。

```
public class _140单词拆分II {
    private HashMap<Integer, List<String>> map = new HashMap<>();
    public List<String> wordBreak(String s, List<String> wordDict) {
        Set<String> dict = new HashSet<>(wordDict);
        return backtrace(s,dict,0);
    }
    private List<String> backtrace(String s, Set<String> dict, int
start) {
        List<String> ret = new ArrayList<>();
        if (start == s.length()){
            ret.add("");
            return ret;
        }
        if (map.containsKey(start))
            return map.get(start);
        for (int i = start+1; i <= s.length(); i++) {
            String word = s.substring(start,i);
            if (dict.contains(word)){
                List<String> list = backtrace(s, dict, i);
                for (String string : list) {
                    ret.add(word + (string.equals("") ? "" : " " +
string));
                }
            }
        }
        map.put(start,ret);
        return ret;
    }
}
```

2.5.9 高楼扔鸡蛋

> 你将获得 K 个鸡蛋，并可以使用一栋从 1 到 N 共有 N 层楼的建筑。
> 每个蛋的功能都是一样的，如果一个蛋碎了，你就不能再把它掉下去。
> 你知道存在楼层 F ，满足 0 <= F <= N 任何从高于 F 的楼层落下的鸡蛋都会碎，从 F 楼层或比它低的楼层落下的鸡蛋都不会破。
> 每次移动，你可以取一个鸡蛋（如果你有完整的鸡蛋）并把它从任一楼层 X 扔下（满足 1 <= X <= N）。
> 你的目标是确切地知道 F 的值是多少。
> 无论 F 的初始值如何，你确定 F 的值的最小移动次数是多少?

```java
public class _887鸡蛋掉落 {
    private int[][] memo;
    public int superEggDrop(int K, int N) {
        memo = new int[K+1][N+1];
        return dp(K, N);
    }
    private int dp(int k, int n) {
        if(k == 1)
            return n;
        if(n == 0)
            return 0;
        if(memo[k][n] != 0)
            return memo[k][n];
        int res = Integer.MAX_VALUE;
        int l = 1, r = n;
        while(l <= r){
            int mid = (l+r)/2;
            int broken = dp(k-1, mid-1);
            int unbroken = dp(k, n-mid);
            if(broken > unbroken){
                r = mid - 1;
                res = Math.min(res, broken+1);
            }else{
                l = mid + 1;
                res = Math.min(res, unbroken+1);
            }
        }
        return memo[k][n] = res;
    }
}
```


2.5.10 打家劫舍问题

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

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

```
class _198打家劫舍 {
    public int rob(int[] nums) {
        if(nums.length == 0)
            return 0;
        if(nums.length == 1)
            return nums[0];
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0],nums[1]);
        for(int i = 2; i < nums.length; i++)
            dp[i] = Math.max(dp[i-1],dp[i-2]+nums[i]);
        return dp[nums.length-1];
    }
    //优化
    public int rob(int[] nums) {
        int pre = 0;
        int ppre = 0;
        int cur = 0;
        for(int i = 0; i < nums.length; i++) {
            cur = Math.max(pre, ppre + nums[i]);
            ppre = pre;
            pre = cur;
        }
        return cur;
    }
}
```

变式题

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

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

```
class _213打家劫舍II {
    public int rob(int[] nums) {
        if(nums.length == 1)
            return nums[0];
```

```
        return Math.max(rob(nums,0,nums.length-
1),rob(nums,1,nums.length));
    }
    public int rob(int[] nums, int start, int end) {
        int pre = 0;
        int ppre = 0;
        int cur = 0;
        for (int i = start; i < end; i ++) {
            cur = Math.max(pre, ppre + nums[i]);
            ppre = pre;
            pre = cur;
        }
        return cur;
    }
}
```

变式题

> 在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。
>
> 计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

记忆化搜索

```
class _337打家劫舍III {
    Map<TreeNode, Integer> memo = new HashMap<>();
    public int rob(TreeNode root) {
        if(root == null)
            return 0;
        if(memo.containsKey(root))
            return memo.get(root);
        int a = root.val + (root.left == null ? 0 :
rob(root.left.left) + rob(root.left.right))
        + (root.right == null ? 0 : rob(root.right.left) +
rob(root.right.right));
        int b = rob(root.left) + rob(root.right);
        int res = Math.max(a, b);
        memo.put(root, res);
        return res;
    }
}
```

改变递归语义 减少重复计算

```
class _337打家劫舍III {
```

```
    public int rob(TreeNode root) {
        int[] res = dp(root);
        return Math.max(res[0], res[1]);
    }
    private int[] dp(TreeNode root) {
        if(root == null)
            return new int[]{0, 0};
        int[] left = dp(root.left);
        int[] right = dp(root.right);
        int rob = root.val + left[0] + right[0];
        int notRob = Math.max(left[0], left[1]) + Math.max(right[0],
right[1]);
        return new int[]{notRob, rob};
    }
}
```

2.5.11 戳气球

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

现在要求你戳破所有的气球。每当你戳破一个气球 i 时，你可以获得 nums[left] *
nums[i] * nums[right] 个硬币。 这里的 left 和 right 代表和 i 相邻的两个气球
的序号。注意当你戳破了气球 i 后，气球 left 和气球 right 就变成了相邻的气球。

求所能获得硬币的最大数量。

说明:

你可以假设 nums[-1] = nums[n] = 1，但注意它们不是真实存在的所以并不能被戳破。
0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100
```

```
class _312戳气球 {
    private int[][] memo;
    public int maxCoins(int[] nums) {
        int n = nums.length;
        int[] newNum = new int[n+2];
        for(int i=0;i<n;i++)
            newNum[i+1] = nums[i];
        newNum[0] = 1;
        newNum[n+1] = 1;
        memo = new int[n+2][n+2];
        return helper(newNum, 0, n+1);
    }
    private int helper(int[] nums, int left, int right){
        if(memo[left][right] != 0)
            return memo[left][right];
        int res = 0;
        for(int i=left+1;i<right;i++)
```

```
            res = Math.max(res,
nums[left]*nums[right]*nums[i]+helper(nums,left,i)+helper(nums,i,right
));
        return memo[left][right] = res;
    }
}
```

2.6滑动窗口

> 给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

```
class _3无重复字符的最长子串 {
    public int lengthOfLongestSubstring(String s) {
        int[] freq = new int[128];
        int l = 0, r = 0;
        int res = 0;
        while(r < s.length()){
            if(freq[s.charAt(r)] == 0)
                freq[s.charAt(r++)]++;
            else
                freq[s.charAt(l++)]--;
            res = Math.max(res, r-l);
        }
        return res;
    }
}
```

变式题

> 给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字母的最小子串。
> 说明：
> 如果 S 中不存这样的子串，则返回空字符串 ""。
> 如果 S 中存在这样的子串，我们保证它是唯一的答案。

```
public class _76最小覆盖子串 {
    public String minWindow(String s, String t) {
        if (s == null || s.equals("") || t == null || t.equals("") ||
s.length() < t.length())
            return "";
        int[] needs = new int[128];
        int[] window = new int[128];
        for (int i = 0; i < t.length(); i++)
            needs[t.charAt(i)]++;
        int left = 0;
        int right = 0;
        String res = "";
        int count = 0;
```

```
        while (right < s.length()) {
            char ch = s.charAt(right);
            if (needs[ch] > window[ch])
                count++;
            window[ch]++;
            while (count == t.length()){
                ch = s.charAt(left);
                window[ch]--;
                if (needs[ch] > window[ch])
                    count--;
                if ("".equals(res) || res.length() > right-left+1)
                    res = s.substring(left, right + 1);
                left++;
            }
            right++;
        }
        return res;
    }
}
```

变式题

> 给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。
> 字符串只包含小写英文字母，并且字符串 s 和 p 的长度都不超过 20100。
> 说明：
> 字母异位词指字母相同，但排列不同的字符串。
> 不考虑答案输出的顺序。

```
public class _438找到字符串中所有字母异位词 {
    public List<Integer> findAnagrams(String s, String p) {
        int[] need = new int[128];
        int[] window = new int[128];
        List<Integer> ret = new ArrayList<>();
        for(int i=0;i<p.length();i++)
            need[p.charAt(i)]++;
        int left = 0;
        int right = 0;
        int count = 0;
        while(right < s.length()){
            char ch = s.charAt(right);
            if(need[ch] > window[ch])
                count++;
            window[ch]++;
            if(right - left + 1 == p.length()){
                if(count == p.length())
                    ret.add(left);
                ch = s.charAt(left);
```

```
                window[ch] --;
                if(need[ch] > 0 && window[ch] < need[ch]){
                    count--;
                }
                left++;
            }
            right++;
        }
        return ret;
    }
```

变式题

> 给定一个含有 n 个正整数的数组和一个正整数 s ，找出该数组中满足其和 ≥ s 的长度最小的连续子数组，并返回其长度。如果不存在符合条件的连续子数组，返回 0。

```
class _209长度最小的子数组 {
    public int minSubArrayLen(int s, int[] nums) {
        int l = 0, r = -1;
        int sum = 0;
        int res = nums.length + 1;
        while(l < nums.length){
            if(r+1 < nums.length && sum < s)
                sum += nums[++r];
            else
                sum -= nums[l++];
            if(sum >= s)
                res = Math.min(res, r-l+1);
        }
        return res == nums.length + 1 ? 0 : res;
    }
}
```

2.7分治

> 给定一个含有数字和运算符的字符串，为表达式添加括号，改变其运算优先级以求出不同的结果。你需要给出所有可能的组合的结果。有效的运算符号包含 +, - 以及 * 。

```
class _241为运算表达式设计优先级 {
    private List<Character> operator = new ArrayList<>();
    private List<Integer> num = new ArrayList<>();
    private List<Integer>[][] memo;
    public List<Integer> diffWaysToCompute(String input) {
        split(input);
        int n = num.size();
        memo = new ArrayList[n][n];
        return conquer(0, n-1);
```

```
    }
    private List<Integer> conquer(int start, int end){
        if(memo[start][end] != null)
            return memo[start][end];
        if (start == end) {
            memo[start][end] = new ArrayList<>
(Collections.singleton(num.get(start)));
            return memo[start][end];
        }
        List<Integer> ret = new ArrayList<>();
        for(int i = start; i < end; i++){
            List<Integer> left = conquer(start, i);
            List<Integer> right = conquer(i+1, end);
            for(int l : left){
                for(int r : right){
                    ret.add(calculate(l, r, operator.get(i)));
                }
            }
        }
        return memo[start][end] = ret;
    }
    private int calculate(int l, int r, char operator){
        switch(operator){
            case '+':
                return l + r;
            case '-':
                return l - r;
            case '*':
                return l * r;
            default:
                return l + r;
        }
    }
    private void split(String input){
        int res = 0;
        for(int i = 0; i < input.length(); i++){
            if(!Character.isDigit(input.charAt(i))){
                num.add(res);
                res = 0;
                operator.add(input.charAt(i));
            }else{
                res = 10 * res + input.charAt(i) - '0';
            }
        }
        num.add(res);
    }
}
```

给定两个大小为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。

请你找出这两个正序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。

你可以假设 nums1 和 nums2 不会同时为空。

```
class _4寻找两个正序数组的中位数 {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int n1 = nums1.length;
        int n2 = nums2.length;
        if (n1>n2)
            return findMedianSortedArrays(nums2, nums1);
        int k = (n1 + n2 + 1)/2;
        int left = 0;
        int right = n1;
        while(left < right){
            int m1 = left +(right - left)/2;
            int m2 = k - m1;
            if (nums1[m1] < nums2[m2-1])
                left = m1 + 1;
            else
                right = m1;
        }
        int m1 = left;
        int m2 = k - left;
        int c1 = Math.max(m1 <= 0 ? Integer.MIN_VALUE : nums1[m1-1],
                          m2 <= 0 ? Integer.MIN_VALUE : nums2[m2-1]);
        if ((n1 + n2) % 2 == 1)
            return c1;
        int c2 = Math.min( m1 >= n1 ? Integer.MAX_VALUE :nums1[m1],
                          m2 >= n2 ? Integer.MAX_VALUE : nums2[m2]);
        return (c1 + c2) * 0.5;
    }
}
```

2.8二分查找

2.8.1 查找

假设按照升序排序的数组在预先未知的某个点上进行了旋转。
(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。
搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。
你可以假设数组中不存在重复的元素。
你的算法时间复杂度必须是 O(log n) 级别。

```
public class _33搜索旋转数组 {
    public int search(int[] nums, int target) {
```

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

变式题

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。
你可以假设数组中无重复元素。

```
public class _35搜索插入位置 {
    public int searchInsert(int[] nums, int target) {
        int l = 0, r = nums.length-1;
        while(l <= r){
            int mid = (l+r)/2;
            if(nums[mid] < target)
                l = mid + 1;
            else
                r = mid - 1;
        }
        return l;
    }
}
```

变式题

```
这是一个 交互式问题
给你一个 山脉数组 mountainArr，请你返回能够使得 mountainArr.get(index) 等于 target 最小 的下标 index 值。
如果不存在这样的下标 index，就请返回 -1。
何为山脉数组？如果数组 A 是一个山脉数组的话，那它满足如下条件：
首先，A.length >= 3
其次，在 0 < i < A.length - 1 条件下，存在 i 使得：
A[0] < A[1] < ... A[i-1] < A[i]
A[i] > A[i+1] > ... > A[A.length - 1]
你将 不能直接访问该山脉数组，必须通过 MountainArray 接口来获取数据：
MountainArray.get(k) - 会返回数组中索引为k 的元素（下标从 0 开始）
MountainArray.length() - 会返回该数组的长度
```

```java
class _1095山脉数组中查找目标值 {
    public int findInMountainArray(int target, MountainArray mountainArr) {
        int l = 0;
        int r = mountainArr.length()-1;
        int peek = findPeek(mountainArr);
        int res = seekFromUpstair(mountainArr, l, peek, target);
        if(res != -1)
            return res;
        return seekFromDownstair(mountainArr, peek+1, r, target);
    }
    private int seekFromUpstair(MountainArray mountainArr, int l, int r, int target){
        while(l < r){
            int mid = (l+r)/2;
            if(mountainArr.get(mid) < target)
                l = mid + 1;
            else
                r = mid;
        }
        return mountainArr.get(l) == target ? l : -1;
    }
    private int seekFromDownstair(MountainArray mountainArr, int l, int r, int target){
        while(l < r){
            int mid = (l+r)/2;
            if(mountainArr.get(mid) > target)
                l = mid + 1;
            else
                r = mid;
        }
        return mountainArr.get(l) == target ? l : -1;
    }
    private int findPeek(MountainArray mountainArr){
        int l = 0, r = mountainArr.length()-1;
```

```
        while(l < r){
            int mid = (l+r)/2;
            if(mountainArr.get(mid) < mountainArr.get(mid+1))
                l = mid+1;
            else
                r = mid;
        }
        return l;
    }
}
```

2.8.2 左边界与右边界

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

你的算法时间复杂度必须是 O(log n) 级别。

如果数组中不存在目标值，返回 [-1, -1]。

```
public class _34在排序数组中查找元素的第一个和最后一个位置 {
    public int[] searchRange(int[] nums, int target) {
        if (nums.length == 0)
            return new int[]{-1,-1};
        int l = findLeftBound(nums, target);
        int r = findRightBound(nums, target);
        return new int[]{l, r};
    }
    private int findLeftBound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target)
                left = mid + 1;
            else
                right = mid - 1;

        }
        if (left >= nums.length || nums[left] != target)
            return -1;
        return left;
    }
    private int findRightBound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > target)
                right = mid - 1;

```

```
            else
                left = mid + 1;
        }
        if (right < 0 || nums[right] != target)
            return -1;
        return right;
    }
}
```

2.8.3 二分性质查找

珂珂喜欢吃香蕉。这里有 N 堆香蕉，第 i 堆中有 piles[i] 根香蕉。警卫已经离开了，将在 H 小时后回来。
珂珂可以决定她吃香蕉的速度 K （单位：根/小时）。每个小时，她将会选择一堆香蕉，从中吃掉 K 根。如果这堆香蕉少于 K 根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。
珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。
返回她可以在 H 小时内吃掉所有香蕉的最小速度 K（K 为整数）。

```
public class _875爱吃香蕉的珂珂 {
    public int minEatingSpeed(int[] piles, int H) {
        int left = 1, right = findMax(piles);
        while(left < right) {
            int mid = (left + right)/2;
            if(canFinish(piles, mid, H))
                right = mid;
            else
                left = mid + 1;
        }
        return left;
    }
    private int findMax(int[] piles){
        int res = piles[0];
        for(int i = 1; i < piles.length; i++)
            res = Math.max(res, piles[i]);
        return res;
    }
    private boolean canFinish(int[] piles, int speed, int h){
        int time = 0;
        for(int p : piles)
            time += p/speed + (p%speed == 0 ? 0 : 1);
        return time <= h;
    }
}
```

变式题

```
实现 int sqrt(int x) 函数。
计算并返回 x 的平方根，其中 x 是非负整数。
由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。
```

```java
class _69X的平方根 {
    public int mySqrt(int x) {
        if(x == 0)
            return x;
        long left = 1;
        long right = x/2;
        while(left < right){
            long mid = (left+right+1)/2;
            if(mid*mid > x)
                right = mid-1;
            else
                left = mid;
        }
        return (int)left;
    }
}
```

2.8.4 利用二分查找优化

之前的例题：

- 二分查找优化高楼扔鸡蛋问题
- 二分查找优化最长上升子序列问题

补充例题：

```
给定字符串 s 和 t ，判断 s 是否为 t 的子序列。
你可以认为 s 和 t 中仅包含英文小写字母。字符串 t 可能会很长（长度 ~= 500,000），而 s 是个短字符串（长度 <=100）。
字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。
```

原题很简单，补充后续挑战解法

```
如果有大量输入的 S，称作S1, S2, ... , Sk 其中 k >= 10亿，你需要依次检查它们是否为 T 的子序列。
在这种情况下，你会怎样改变代码?
```

```java
public class _392判断子序列 {
    public boolean isSubsequence(String s, String t) {
        int m = s.length();
        int n = t.length();
        Map<Character,ArrayList<Integer>> map = new HashMap<>();
        for(int i = 0; i < n; i++){
```

```
            char c = t.charAt(i);
            if(!map.containsKey(c))
                map.put(c, new ArrayList<>());
            map.get(c).add(i);
        }
        int j = 0;
        for(int i = 0; i < m; i++){
            char c = s.charAt(i);
            if(!map.containsKey(c))
                return false;
            int pos = leftBound(map.get(c), j);
            if(pos == map.get(c).size())
                return false;
            j = map.get(c).get(pos) + 1;
        }
        return true;
    }
    private int leftBound(ArrayList<Integer> nums, int target) {
        int l = 0, r = nums.size()-1;
        while(l <= r){
            int mid = (l+r)/2;
            if(nums.get(mid) < target)
                l = mid + 1;
            else
                r = mid - 1;
        }
        return l;
    }
}
```

2.9 搜索

2.9.1 floodfill

给定一个 m x n 的非负整数矩阵来表示一片大陆上各个单元格的高度。“太平洋”处于大陆的左边界和上边界，而“大西洋”处于大陆的右边界和下边界。

规定水流只能按照上、下、左、右四个方向流动，且只能从高到低或者在同等高度上流动。

请找出那些水流既可以流动到“太平洋”，又能流动到“大西洋”的陆地单元的坐标。

```
class _417太平洋大西洋水流问题 {
    public List<List<Integer>> pacificAtlantic(int[][] matrix) {
        List<List<Integer>> list = new ArrayList<>();
        m = matrix.length;
        if(m == 0)
            return list;
        n = matrix[0].length;
```

```
        boolean[][] p = new boolean[m][n];
        boolean[][] a = new boolean[m][n];
        for(int i = 0; i < m; i++){
            dfs(matrix, p, i, 0);
            dfs(matrix, a, i, n-1);
        }
        for(int i = 0; i < n; i++){
            dfs(matrix, p, 0, i);
            dfs(matrix, a, m-1, i);
        }
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(p[i][j] && a[i][j])
                    list.add(Arrays.asList(i,j));
        return list;
    }
    private int m;
    private int n;
    private int[][] dir = {{0,1},{0,-1},{1,0},{-1,0}};
    private void dfs(int[][] matrix, boolean[][] visited, int x, int
y){
        visited[x][y] = true;
        for(int[] d : dir){
            int nx = x + d[0];
            int ny = y + d[1];
            if(0 <= nx && nx < m && 0 <= ny && ny < n && !visited[nx]
[ny] && matrix[x][y] <= matrix[nx][ny])
                dfs(matrix, visited, nx, ny);
        }
    }
}
```

2.9.2 无权图最短路径

> 你现在手里有一份大小为 N x N 的「地图」（网格） grid，上面的每个「区域」（单元格）都用 0 和 1 标记好了。其中 0 代表海洋，1 代表陆地，请你找出一个海洋区域，这个海洋区域到离它最近的陆地区域的距离是最大的。
>
> 我们这里说的距离是「曼哈顿距离」（ Manhattan Distance）：(x0, y0) 和 (x1, y1) 这两个区域之间的距离是 |x0 - x1| + |y0 - y1| 。
>
> 如果我们的地图上只有陆地或者海洋，请返回 -1。

```
class _1162地图分析 {
    public int maxDistance(int[][] grid) {
        int n = grid.length;
        Queue<Integer> queue = new LinkedList<>();
        for(int i=0;i<n;i++){
```

```
            for(int j=0;j<n;j++){
                if(grid[i][j]==1)
                    queue.offer(i*n+j);
            }
        }
        if(queue.isEmpty() || queue.size() == n*n)
            return -1;
        int[][] dir = {{0,1},{0,-1},{1,0},{-1,0}};
        int distance = -1;
        while(!queue.isEmpty()){
            int size = queue.size();
            distance ++;
            for(int j=0;j<size;j++){
                int cur = queue.poll();
                int x = cur / n;
                int y = cur % n;
                for(int i=0;i<4;i++){
                    int newX = x + dir[i][0];
                    int newY = y + dir[i][1];
                    if(0 <= newX && newX < n && 0 <= newY && newY < n
&& grid[newX][newY] == 0){
                        grid[newX][newY] = 1;
                        queue.offer(newX*n+newY);
                    }
                }
            }
        }
        return distance;
    }
}
```

变式题

> 给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

```
class _279完全平方数 {
    public int numSquares(int n) {
        Queue<Integer> queue = new LinkedList<>();
        boolean[] visited = new boolean[n+1];
        queue.offer(n);
        visited[n] = true;
        int step = 0;
        while(!queue.isEmpty()){
            int size = queue.size();
            for(int i = 0; i < size; i++){
                int cur = queue.poll();
                for(int j = 1; j * j <= cur; j++){
```

```
                    int num = cur-j*j;
                    if(num == 0)
                        return step+1;
                    if(!visited[num]){
                        queue.offer(num);
                        visited[num] = true;
                    }
                }
            }
            step ++;
        }
        return step;
    }
}
```

2.9.3 状态搜索

在一个 2 x 3 的板上（board）有 5 块砖瓦，用数字 1~5 来表示，以及一块空缺用 0 来表示.
一次移动定义为选择 0 与一个相邻的数字（上下左右）进行交换.
最终当板 board 的结果是 [[1,2,3],[4,5,0]] 谜板被解开。
给出一个谜板的初始状态，返回最少可以通过多少次移动解开谜板，如果不能解开谜板，则返回 -1 。

```
class _773滑动谜题 {
    public int slidingPuzzle(int[][] board) {
        Queue<String> queue = new LinkedList<>();
        String s = board2String(board);
        if("123450".equals(s))
            return 0;
        queue.offer(s);
        Map<String,Integer> visited = new HashMap<>();
        visited.put(s, 0);
        while(!queue.isEmpty()){
            String cur = queue.poll();
            int zero = cur.indexOf("0");
            int x = zero/3;
            int y = zero%3;
            int[][] b = string2board(cur);
            for(int[] d : dir){
                int nx = x + d[0];
                int ny = y + d[1];
                if(0 <= nx && nx < 2 && 0 <= ny && ny < 3){
                    swap(b, x, y, nx, ny);
                    String next = board2String(b);
                    if("123450".equals(next))
                        return visited.get(cur) + 1;
                    if(!visited.containsKey(next)){
```

```
                    queue.offer(next);
                    visited.put(next, visited.get(cur) + 1);
                }
                swap(b, x, y, nx, ny);
            }
        }
    }
    return -1;
}
private void swap(int[][] board, int x, int y, int nx, int ny){
    int temp = board[x][y];
    board[x][y] = board[nx][ny];
    board[nx][ny] = temp;
}
private int[][] dir = {{0,1},{0,-1},{1,0},{-1,0}};
private String board2String(int[][] board){
    StringBuilder s = new StringBuilder();
    for(int i = 0; i < board.length; i++){
        for(int j = 0; j < board[0].length; j++){
            s.append(board[i][j]);
        }
    }
    return s.toString();
}
private int[][] string2board(String s){
    int[][] board = new int[2][3];
    for(int i = 0; i < s.length(); i++)
        board[i/3][i%3] = s.charAt(i)-'0';
    return board;
}
}
```

变式智力题

> 一个水桶5升，一个水桶3升，怎样才能量出4升的水？设计一个类来求解。

```
public class WaterPuzzle {
    private int[] pre;
    private int end = -1;
    public WaterPuzzle(){
        Queue<Integer> queue = new LinkedList<>();
        boolean[] visited = new boolean[100];
        pre = new int[100];
        queue.add(0);
        visited[0] = true;
        while (!queue.isEmpty()){
            int cur = queue.remove();
            int a = cur / 10;
```

```
                int b = cur % 10;
                ArrayList<Integer> nexts = new ArrayList<>();
                nexts.add(5*10+b);
                nexts.add(a*10+3);
                nexts.add(b);
                nexts.add(a*10);
                int x = Math.min(a, 3-b);
                nexts.add((a-x)*10+b+x);
                int y = Math.min(5-a, b);
                nexts.add((a+y)*10+b-y);
                for (int next : nexts) {
                    if(!visited[next]){
                        queue.add(next);
                        visited[next] = true;
                        pre[next] = cur;
                        if(next / 10 == 4 || next % 10 == 4) {
                            end = next;
                            return;
                        }
                    }
                }
            }
        }
        public Iterable<Integer> result(){
            ArrayList<Integer> res = new ArrayList<>();
            if(end == -1)
                return res;
            int cur = end;
            while (cur != 0){
                res.add(cur);
                cur = pre[cur];
            }
            res.add(0);
            Collections.reverse(res);
            return res;
        }
    }
```

2.9.4 双向搜索优化

给定两个单词（beginWord 和 endWord）和一个字典，找到从 beginWord 到 endWord 的最短转换序列的长度。转换需遵循如下规则：
每次转换只能改变一个字母。
转换过程中的中间单词必须是字典中的单词。
说明:
如果不存在这样的转换序列，返回 0。
所有单词具有相同的长度。
所有单词只由小写字母组成。
字典中不存在重复的单词。
你可以假设 beginWord 和 endWord 是非空的，且二者不相同。

```
class _127单词接龙 {
    public int ladderLength(String beginWord, String endWord,
List<String> wordList) {
        if (wordList == null || wordList.size() == 0)
            return 0;
        HashSet<String> dic = new HashSet<>(wordList);
        if (!dic.contains(endWord))
            return 0;
        HashSet<String> start = new HashSet<>();
        HashSet<String> end = new HashSet<>();
        start.add(beginWord);
        end.add(endWord);
        int step = 1;
        while(!start.isEmpty()){
            step++;
            HashSet<String> tmpSet=new HashSet<>();
            dic.removeAll(start);
            for(String s : start){
                char[] arr = s.toCharArray();
                for(int i = 0; i < arr.length; i++){
                    char tmp = arr[i];
                    for(char c = 'a'; c <= 'z'; c++){
                        if(tmp == c)
                            continue;
                        arr[i] = c;
                        String strTmp = new String(arr);
                        if(dic.contains(strTmp)){
                            if(end.contains(strTmp)){
                                return step;
                            }else{
                                tmpSet.add(strTmp);
                            }
                        }
                    }
                    arr[i]=tmp;
                }
            }
```

```
            if(tmpSet.size() < end.size()){
                start = tmpSet;
            }else{
                start = end;
                end = tmpSet;
            }
        }
        return 0;
    }
}
```

变式题

```
给定两个单词（beginWord 和 endWord）和一个字典 wordList，找出所有从 beginWord
到 endWord 的最短转换序列。转换需遵循如下规则：
每次转换只能改变一个字母。
转换过程中的中间单词必须是字典中的单词。
说明:
如果不存在这样的转换序列，返回一个空列表。
所有单词具有相同的长度。
所有单词只由小写字母组成。
字典中不存在重复的单词。
你可以假设 beginWord 和 endWord 是非空的，且二者不相同。
```

```
class _126单词接龙II {
    public List<List<String>> findLadders(String beginWord, String
endWord, List<String> wordList) {
        List<List<String>> res = new ArrayList<>();
        Set<String> words = new HashSet<>(wordList);
        if (!words.contains(endWord))
            return res;
        Map<String, List<String>> mapTree = new HashMap<>();
        Set<String> begin = new HashSet<>(), end = new HashSet<>();
        begin.add(beginWord);
        end.add(endWord);
        if (buildTree(words, begin, end, mapTree, true))
            dfs(res, mapTree, beginWord, endWord, new LinkedList<>());
        return res;
    }
    private boolean buildTree(Set<String> words, Set<String> begin,
Set<String> end, Map<String, List<String>> mapTree, boolean isFront){
        if (begin.size() == 0)
            return false;
        if (begin.size() > end.size())
            return buildTree(words, end, begin, mapTree, !isFront);
        words.removeAll(begin);
        boolean isMeet = false;
        Set<String> nextLevel = new HashSet<>();
```

```
        for (String word : begin) {
            char[] chars = word.toCharArray();
            for (int i = 0; i < chars.length; i++) {
                char temp = chars[i];
                for (char ch = 'a'; ch <= 'z'; ch++) {
                    chars[i] = ch;
                    String str = String.valueOf(chars);
                    if (words.contains(str)) {
                        nextLevel.add(str);
                        String key = isFront ? word : str;
                        String nextWord = isFront ? str : word;
                        if (end.contains(str))
                            isMeet = true;
                        if (!mapTree.containsKey(key))
                            mapTree.put(key, new ArrayList<>());
                        mapTree.get(key).add(nextWord);
                    }
                }
                chars[i] = temp;
            }
        }
        if (isMeet)
            return true;
        return buildTree(words, nextLevel, end, mapTree, isFront);
    }
    private void dfs (List<List<String>> res, Map<String,
List<String>> mapTree, String beginWord, String endWord,
LinkedList<String> list) {
        list.add(beginWord);
        if (beginWord.equals(endWord)) {
            res.add(new ArrayList<>(list));
            list.removeLast();
            return;
        }
        if (mapTree.containsKey(beginWord))
            for (String word : mapTree.get(beginWord))
                dfs(res, mapTree, word, endWord, list);
        list.removeLast();
    }
}
```


2.10 图论

2.10.1 最小生成树

想象一下你是个城市基建规划者，地图上有 N 座城市，它们按以 1 到 N 的次序编号。

给你一些可连接的选项 conections，其中每个选项 conections[i] = [city1, city2, cost] 表示将城市 city1 和城市 city2 连接所要的成本。（连接是双向的，也就是说城市 city1 和城市 city2 相连也同样意味着城市 city2 和城市 city1 相连）。

返回使得每对城市间都存在将它们连接在一起的连通路径（可能长度为 1 的）最小成本。该最小成本应该是所用全部连接代价的综合。如果根据已知条件无法完成该项任务，则请你返回 -1。

2.10.1.1 Kruskal算法

```
class _1135最低成本联通所有城市 {
    public int minimumCost(int N, int[][] connections) {
        Arrays.sort(connections, Comparator.comparingInt(a -> a[2]));
        UnionFind uf = new UnionFind(N);
        int cost = 0;
        for (int[] arr : connections){
            int v = arr[0] - 1;
            int w = arr[1] - 1;
            if (!uf.isConnected(v,w)) {
                cost += arr[2];
                uf.unionElements(v, w);
            }
        }
        return uf.getCount() == 1 ? cost : -1;
    }
}
class UnionFind{
    private int[] parent;
    private int[] rank;
    private int count;
    public UnionFind(int size){
        parent = new int[size];
        rank = new int[size];
        count = size;
        for (int i = 0; i < size; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }
    private int find(int p){
        if (p != parent[p])
            parent[p] = find(parent[p]);
        return parent[p];
    }
    public int getCount(){
        return count;
    }
```

```
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }
    public void unionElements(int p, int q) {
        int pRoot = find(p);
        int qRoot = find(q);
        if(pRoot == qRoot)
            return;
        count --;
        if(rank[pRoot] < rank[qRoot])
            parent[pRoot] = qRoot;
        else if(rank[pRoot] > rank[qRoot])
            parent[qRoot] = pRoot;
        else {
            parent[qRoot] = pRoot;
            rank[pRoot] += 1;
        }
    }
}
```

2.10.1.2 Prim算法

```
class Solution {
    public int minimumCost(int N, int[][] connections) {
        Queue<int[]> queue = new PriorityQueue<>
(Comparator.comparingInt(a->a[2]));
        boolean[] visited = new boolean[N];
        visited[0] = true;
        Set<int[]>[] graph = new Set[N];
        for(int i = 0; i < N; i++)
            graph[i] = new HashSet<>();
        for(int[] edge : connections){
            int v = edge[0] - 1;
            int w = edge[1] - 1;
            graph[w].add(new int[]{v,edge[2]});
            graph[v].add(new int[]{w,edge[2]});
        }
        for (int[] arr : graph[0])
            queue.offer(new int[]{0,arr[0],arr[1]});
        int cost = 0;
        while (!queue.isEmpty()){
            int[] cur = queue.poll();
            if (visited[cur[0]] && visited[cur[1]])
                continue;
            cost += cur[2];
            int newV = visited[cur[0]] ? cur[1] : cur[0];
            visited[newV] = true;
            for (int[] adj : graph[newV]){
                if (!visited[adj[0]]){
```

```
                    queue.offer(new int[]{newV, adj[0],adj[1]});
                }
            }
        }
        for(int i = 0; i < visited.length; i++)
            if(!visited[i])
                return -1;
        return cost;
    }
}
```

2.10.2 最短路径

> 有 N 个网络节点，标记为 1 到 N。
> 给定一个列表 times，表示信号经过有向边的传递时间。 times[i] = (u, v, w)，其中 u 是源节点，v 是目标节点， w 是一个信号从源节点传递到目标节点的时间。
> 现在，我们从某个节点 K 发出一个信号。需要多久才能使所有节点都收到信号？如果不能使所有节点收到信号，返回 -1。

2.10.2.1 Dijkstra算法

```
class _743网络延迟时间 {
    public int networkDelayTime(int[][] times, int N, int K) {
        HashMap<Integer,Integer>[] map = new HashMap[N+1];
        for(int i = 0; i <= N;i++)
            map[i] = new HashMap<>();
        for(int[] t : times)
            map[t[0]].put(t[1],t[2]);
        int res = 0;
        int[] distance = new int[N+1];
        Arrays.fill(distance, Integer.MAX_VALUE);
        boolean[] visited = new boolean[N+1];
        Queue<int[]> queue = new PriorityQueue<>((a,b)->a[1]-b[1]);
        queue.offer(new int[]{K,0});
        distance[K] = 0;
        while(!queue.isEmpty()){
            int cur = queue.poll()[0];
            if(visited[cur])
                continue;
            visited[cur] = true;
            for(Map.Entry<Integer,Integer> entry :
map[cur].entrySet()){
                int w = entry.getKey();
                int v = entry.getValue();
                if(distance[cur] + v < distance[w]){
                    distance[w] = distance[cur] + v;
                    queue.offer(new int[]{w, distance[w]});
                }
            }
```

```
        }
        for(int i = 1; i < distance.length;i++){
            if(distance[i] == Integer.MAX_VALUE)
                return -1;
            res = Math.max(res, distance[i]);
        }
        return res;
    }
}
```

2.10.2.2 BellmanFord算法

```
class _743网络延迟时间 {
    public int networkDelayTime(int[][] times, int N, int K) {
        int[] distance = new int[N+1];
        int[][] graph = new int[N+1][N+1];
        for(int[] g : graph)
            Arrays.fill(g, -1);
        for(int[] t : times)
            graph[t[0]][t[1]] = t[2];
        Arrays.fill(distance, Integer.MAX_VALUE);
        distance[K] = 0;
        for(int pass = 1; pass < N; pass ++){
            for(int i = 1; i <= N; i++){
                for(int j = 1; j <= N; j++){
                    if(graph[i][j] == -1)
                        continue;
                    if(distance[i] != Integer.MAX_VALUE && distance[j]
> distance[i] + graph[i][j]){
                        distance[j] = distance[i] + graph[i][j];
                    }
                }
            }
        }
        int res = 0;
        for(int i=1;i<distance.length;i++){
            if(distance[i] == Integer.MAX_VALUE)
                return -1;
            res = Math.max(res, distance[i]);
        }
        return res;
    }
}
```

2.10.2.3 Floyd算法

```
class _743网络延迟时间 {
    public int networkDelayTime(int[][] times, int N, int K) {
        int[][] graph = new int[N + 1][N + 1];
```

```
        for (int i = 1; i <= N; i++)
            for (int j = 1; j <= N; j++)
                graph[i][j] = i == j ? 0 : -1;
        for (int[] time : times)
            graph[time[0]][time[1]] = time[2];
        for (int k = 1; k <= N; k++) {
            for (int i = 1; i <= N; i++) {
                for (int j = 1; j <= N; j++) {
                    if (graph[i][k] != -1 && graph[k][j] != -1) {
                        if (graph[i][j] != -1)
                            graph[i][j] = Math.min(graph[i][j],
graph[i][k] + graph[k][j]);
                        else
                            graph[i][j] = graph[i][k] + graph[k][j];
                    }
                }
            }
        }
        int maxDistance = 0;
        for (int i = 1; i <= N; i++) {
            if (graph[K][i] == -1)
                return -1;
            maxDistance = Math.max(maxDistance, graph[K][i]);
        }
        return maxDistance;
    }
}
```

2.10.3 拓扑排序

现在你总共有 n 门课需要选，记为 0 到 n-1。
在选修某些课程之前需要一些先修课程。 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示他们: [0,1]
给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。
可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

```
class _210课程表II {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        List<Integer>[] graph = new List[numCourses];
        for(int i = 0; i < graph.length; i++)
            graph[i] = new ArrayList<>();
        int[] inDegree = new int[numCourses];
        for(int[] pre : prerequisites){
            graph[pre[1]].add(pre[0]);
            inDegree[pre[0]]++;
        }
        Queue<Integer> queue = new LinkedList<>();
```

```
        for(int i = 0; i < inDegree.length; i++)
            if(inDegree[i] == 0)
                queue.offer(i);
        int[] ret = new int[numCourses];
        int index = 0;
        while(!queue.isEmpty()){
            int cur = queue.poll();
            ret[index++] = cur;
            for(int adj : graph[cur]){
                inDegree[adj] --;
                if(inDegree[adj] == 0)
                    queue.offer(adj);
            }
        }
        return index == numCourses ? ret : new int[]{};
    }
}
```

2.10.4 关键路径

```
n个实验 m行前置实验
n个数字，从1开始，表示每个实验需要花费的时间
m行前置实验，表示前者为后者的前置实验
计算最晚完成实验的时间
```

```
public class Experiment {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        int m = scanner.nextInt();
        Set<Integer>[] graph = new HashSet[n];
        for (int i = 0; i < graph.length; i++) {
            graph[i] = new HashSet<>();
        }
        int[] cost = new int[n];
        int[] degree = new int[n];
        for (int i = 0; i < n; i++) {
            cost[i] = scanner.nextInt();
        }
        for (int i = 0; i < m; i++) {
            int a = scanner.nextInt()-1;
            int b = scanner.nextInt()-1;
            graph[a].add(b);
            degree[b]++;
        }
        List<Integer> topology = new ArrayList<>();
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < degree.length; i++) {
```

```
            if (degree[i] == 0)
                queue.offer(i);
        }
        int totalTime = 0;
        int[] earliest = new int[n];
        int[] latest = new int[n];
        while (!queue.isEmpty()){
            int u = queue.poll();
            topology.add(u);
            for (int v : graph[u]) {
                earliest[v] = Math.max(earliest[v],
earliest[u]+cost[u]);
                degree[v] --;
                if (degree[v] == 0){
                    queue.offer(v);
                    totalTime =
Math.max(totalTime,earliest[v]+cost[v]);
                }
            }
        }
        for (int i = topology.size()-1; i >= 0; i--) {
            int u = topology.get(i);
            if (graph[u].size() == 0){
                latest[u] = totalTime-cost[u];
            }else {
                latest[u] = Integer.MAX_VALUE;
            }
            for (int v : graph[u]) {
                latest[u] = Math.min(latest[u], latest[v]-cost[u]);
            }
        }
        System.out.println(totalTime);
    }
}
```

2.10.5 二分图

给定一个无向图graph，当这个图为二分图时返回true。
如果我们能将一个图的节点集合分割成两个独立的子集A和B，并使图中的每一条边的两个节点一个来自A集合，一个来自B集合，我们就将这个图称为二分图。
graph将会以邻接表方式给出，graph[i]表示图中与节点i相连的所有节点。每个节点都是一个在0到graph.length-1之间的整数。这图中没有自环和平行边： graph[i] 中不存在i，并且graph[i]中没有重复的值。

深度优先遍历解法

```
class _785判断二分图 {
    public boolean isBipartite(int[][] graph) {
```

```
        int n = graph.length;
        int[] color = new int[n];
        Arrays.fill(color, -1);
        for(int i = 0; i < n; i++)
            if(color[i] == -1 && !dfs(graph, 0, i, color))
                return false;
        return true;
    }
    private boolean dfs(int[][] graph, int c, int i, int[] color){
        color[i] = c;
        for(int w : graph[i]){
            if(color[w] == -1 && !dfs(graph, 1-c, w, color))
                return false;
            else if(color[w] == color[i])
                return false;
        }
        return true;
    }
}
```

广度优先遍历解法

```
class _785判断二分图 {
    public boolean isBipartite(int[][] graph) {
        int n = graph.length;
        int[] color = new int[n];
        Arrays.fill(color, -1);
        for (int i = 0; i < n; ++i) {
            if (color[i] == -1) {
                Stack<Integer> stack = new Stack<>();
                stack.push(i);
                color[i] = 0;
                while (!stack.empty()) {
                    int cur = stack.pop();
                    for (int adj: graph[cur]) {
                        if (color[adj] == -1) {
                            stack.push(adj);
                            color[adj] = color[cur] ^ 1;
                        } else if (color[adj] == color[cur]) {
                            return false;
                        }
                    }
                }
            }
        }
        return true;
    }
}
```

3.数学、模拟与技巧篇

3.1 数学

3.1.1 进制问题

> 对于一个十进制数A，将A转换为二进制数，然后按位逆序排列，再转换为十进制数B，我们称B为A的二进制逆序数。
> 例如对于十进制数173，它的二进制形式为10101101，逆序排列得到10110101，其十进制数为181，181即为173的二进制逆序数。

模拟解法

```
import java.util.Scanner;
import java.util.Stack;
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String s = scanner.next();
        Stack<Integer> stack = new Stack<>();
        while (s.length() > 0){
            stack.push((s.charAt(s.length()-1)-'0')%2);
            s = divide(s,2);
        }
        StringBuilder sb = new StringBuilder();
        while (!stack.isEmpty())
            sb.append(stack.pop());
        String rev = "0";
        for (int i = sb.length()-1; i >= 0; i--) {
            rev = multiply(rev,2);
            rev = add(rev,sb.charAt(i)-'0');
        }
        System.out.println(rev);
    }
    public static String multiply(String string, int x){
        int carry = 0;
        char[] chars = string.toCharArray();
        for (int i = chars.length-1; i >= 0; i--) {
            int cur = x * (chars[i] - '0') + carry;
            chars[i] = (char) (cur % 10 + '0');
            carry = cur / 10;
        }
        return (carry != 0 ? carry : "") + new String(chars);
    }
    public static String divide(String string, int x){
        char[] chars = string.toCharArray();
        int reminder = 0;
        for (int i = 0; i < string.length(); i++) {
```

```
            int cur = reminder*10 + chars[i] - '0';
            chars[i] = (char) (cur/x + '0');
            reminder = cur % x;
        }
        int pos = 0;
        while (pos < chars.length && chars[pos] == '0')
            pos++;
        return new String(chars,pos, chars.length-pos);
    }
    public static String add(String string, int x){
        int carry = x;
        char[] chars = string.toCharArray();
        for (int i = chars.length-1; i >= 0; i--) {
            int cur = chars[i] - '0' + carry;
            chars[i] = (char) (cur % 10 + '0');
            carry = cur / 10;
        }
        return (carry != 0 ? carry : "") + new String(chars);
    }
}
```

流氓解法

```
public class Main{
    public static void main(String[] args){
        Scanner scanner = new Scanner(System.in);
        while(scanner.hasNext()){
            String s = scanner.next();
            System.out.println(new BigInteger(new StringBuilder(new
BigInteger(s,10).toString(2)).reverse().toString(),2).toString(10));
        }
    }
}
```

3.1.2 最大公因数

```
public int gcd(int x, int y){
    return y == 0 ? x : gcd(y, x%y);
}
```

3.1.3 素数

```
给定n，a求最大的k，使n！可以被a^k整除但不能被a^(k+1)整除。
```

```
import java.util.Arrays;
import java.util.Scanner;
public class Main {
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Arrays.fill(isPrim,true);
        init();
        while (scanner.hasNext()){
            int n = scanner.nextInt();
            int a = scanner.nextInt();
            int[] count1 = new int[1001];
            int[] count2 = new int[1001];
            for (int i = 0; i < primSize; i++) {
                int t = n;
                while (t != 0){
                    count1[i] += t/prim[i];
                    t /= prim[i];
                }
            }
            int ans = Integer.MAX_VALUE;
            for (int i = 0; i < primSize; i++){
                while (a % prim[i] == 0){
                    count2[i]++;
                    a /= prim[i];
                }
                if (count2[i] == 0)
                    continue;
                ans = Math.min(ans,count1[i]/count2[i]);
            }
            System.out.println(ans);
        }
    }
    private static boolean[] isPrim = new boolean[1001];
    private static int[] prim = new int[1001];
    private static int primSize = 0;
    private static void init() {
        for (int i = 2; i < 1001; i++) {
            if (isPrim[i]){
                prim[primSize++]=i;
                for (int j = i*i; j < 1001; j += i) {
                    isPrim[j] = false;
                }
            }
        }
    }
}
```


3.1.4 快速幂

实现 pow(x, n) ，即计算 x 的 n 次幂函数。

```
class _50Pow {
    public double myPow(double x, int n) {
        if(x == 0)
            return 0.0;
        long d = n;
        if(d < 0){
            x = 1/x;
            d = -d;
        }
        double res = 1.0;
        while(d > 0){
            if((d & 1) == 1)
                res *= x;
            x *= x;
            d >>= 1;
        }
        return res;
    }
}
```

3.1.5 矩阵快速幂

给定一个n*n矩阵，求矩阵k次幂

```
import java.util.Scanner;
public class 矩阵幂 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            int n = scanner.nextInt();
            int k = scanner.nextInt();
            int[][] arr = new int[n][n];
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    arr[i][j] = scanner.nextInt();
                }
            }
            int[][] ret = fastExponentiation(arr, k);
            print(ret);
        }
    }
    //matrix m*n x n*k
    private static int[][] multiply(int[][] matrix, int[][] x){
        int m = matrix.length;
        int n = matrix[0].length;
        int k = x[0].length;
        int[][] ret = new int[m][k];
        for (int i = 0; i < m; i++) {
```

```
                for (int j = 0; j < k; j++) {
                    for (int l = 0; l < n; l++) {
                        ret[i][j] += matrix[i][l] * x[l][j];
                    }
                }
            }
            return ret;
        }
        private static void print(int[][] matrix){
            for (int[] ints : matrix) {
                for (int j = 0; j < matrix[0].length; j++) {
                    System.out.print(ints[j] + (j == matrix[0].length - 1
? "" : " "));
                }
                System.out.println();
            }
        }

        private static int[][] fastExponentiation(int[][] matrix, int k){
            int[][] ret = new int[matrix.length][matrix[0].length];
            for (int i = 0; i < matrix.length; i++)
                for (int j = 0; j < matrix[0].length; j++)
                    ret[i][j] = i == j ? 1 : 0;
            while (k != 0){
                if ((k & 1) == 1)
                    ret = multiply(ret,matrix);
                k >>= 1;
                matrix = multiply(matrix,matrix);
            }
            return ret;
        }
    }
```

3.1.6 模拟高精度运算

用BigInteger和BigDecimal即可

> 给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。

```
public class _43字符串相乘 {
    public String multiply(String num1, String num2) {
        if("0".equals(num1) || "0".equals(num2))
            return "0";
        int m = num1.length();
        int n = num2.length();
        int[] result = new int[m+n];
        for(int i=num1.length()-1;i>=0;i--){
            int a = num1.charAt(i)-'0';
```

```
            for(int j=num2.length()-1;j>=0;j--){
                int b = num2.charAt(j)-'0';
                int sum = a*b + result[i+j+1];
                result[i+j+1] = sum % 10;
                result[i+j] += sum /10;
            }
        }
        StringBuilder sb = new StringBuilder();
        for(int i=0;i<result.length;i++){
            if(i==0 && result[i] == 0)
                continue;
            sb.append(result[i]);
        }
        return sb.toString();
    }
}
```

给定a和n，计算a+aa+aaa+a...a(n个a)的和。

```
import java.util.Scanner;
import java.util.Stack;
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()){
            int a = scanner.nextInt();
            int n = scanner.nextInt();
            Stack<Integer> stack = new Stack<>();
            int carry = 0;
            for (int i = 0; i < n; i++) {
                int sum = a * (n-i) + carry;
                stack.push(sum % 10);
                carry = sum / 10;
            }
            StringBuilder sb = new StringBuilder();
            while (!stack.isEmpty())
                sb.append(stack.pop());
            System.out.println(sb);
        }
    }
}
```


3.1.7 下一个排列

实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。
如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。
必须原地修改，只允许使用额外常数空间。

```java
class _31下一个排列 {
    public void nextPermutation(int[] nums) {
        int i = nums.length - 2;
        while(i >= 0 && nums[i+1] <= nums[i])
            i--;
        if(i >= 0){
            int j = nums.length - 1;
            while(nums[j] <= nums[i])
                j--;
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
        i ++;
        int k = nums.length - 1;
        while(i < k){
            int temp = nums[i];
            nums[i] = nums[k];
            nums[k] = temp;
            i++;
            k--;
        }
    }
}
```

给出集合 [1,2,3,…,n]，其所有元素共有 n! 种排列。
给定 n 和 k，返回第 k 个排列。
说明：
给定 n 的范围是 [1, 9]。
给定 k 的范围是[1, n!]。

```java
class _60第K个排列 {
    public String getPermutation(int n, int k) {
        int[] factorialNum = new int[n];
        k = k - 1;
        factorialNum[n - 1] = 0;
        for (int i = 1; i < n; i++) {
            factorialNum[n - i - 1] = k % (i + 1);
            k /= i + 1;
        }
        List<Integer> nums = new ArrayList<>();
        for (int i = 0; i < n; i++)
            nums.add(i + 1);
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < n; i++) {
            sb.append(nums.get(factorialNum[i]));
            nums.remove(factorialNum[i]);
        }
```

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

3.2 数组

3.2.1 前缀和

```
给定一个整数数组  nums，求出数组从索引 i 到 j  (i ≤ j) 范围内元素的总和，包含 i,  j 两点。
```

```
class _303区间和检索数组不可变 {
    private int[] sum;
    public NumArray(int[] nums) {
        sum = new int[nums.length+1];
        for(int i=1;i<sum.length;i++)
            sum[i] = sum[i-1] + nums[i-1];
    }
    public int sumRange(int i, int j) {
        return sum[j+1] - sum[i];
    }
}
```

```
返回 A 的最短的非空连续子数组的长度，该子数组的和至少为 K 。
如果没有和至少为 K 的非空子数组，返回 -1 。
```

```
class _862和至少为K的最短子数组{
    public int shortestSubarray(int[] A, int K) {
        int n = A.length;
        int[] sum = new int[n+1];
        for(int i=0;i<n;i++)
            sum[i+1] = sum[i]+A[i];
        Deque<Integer> queue = new LinkedList<>();
        int res = n+1;
        for(int i=0;i<=n;i++){
            while(!queue.isEmpty() && sum[i]<= sum[queue.getLast()])
                queue.removeLast();
            while(!queue.isEmpty() && sum[i]-sum[queue.getFirst()] >=
K)
                res = Math.min(res,i-queue.removeFirst());
            queue.addLast(i);
        }
        return res == n+1 ? -1 : res;
    }
}
```

3.2.2 出现数字次数

数组nums包含从0到n的所有整数，但其中缺了一个。请编写代码找出那个缺失的整数。你有办法在O(n)时间内完成吗?

```java
class Solution {
    public int missingNumber(int[] nums) {
        int n = nums.length;
        int res = 0;
        res += n;
        for(int i = 0; i < n; i++)
            res += i - nums[i];
        return res;
    }
}
```

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

```java
class Solution {
    public int findRepeatNumber(int[] nums) {
        for(int i=0;i<nums.length;i++){
            int index = Math.abs(nums[i]);
            if(nums[index] < 0)
                return index;
            nums[index] *= -1;
        }
        return 0;
    }
}
```

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

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

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

```java
class Solution {
```

```
    public int[] singleNumbers(int[] nums) {
        int sum=0;
        for (int i = 0; i <nums.length ; i++)
            sum^=nums[i];
        int first = 1;
        while((sum&first)==0)
            first=first<<1;
        int result[]=new int[2];
        for(int i=0;i<nums.length;i++){
            if((nums[i]&first)==0)
                result[0]^=nums[i];
            else
                result[1]^=nums[i];
        }
        return result;
    }
}
```

3.3 位运算

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

```
public class _191位1的个数 {
    public int hammingWeight(int n) {
        int res = 0;
        while(n != 0){
            n = n & (n-1);
            res ++;
        }
        return res;
    }
}
```

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

```
class Solution {
    public boolean isPowerOfTwo(int n) {
        return n > 0 && (n & (n-1)) == 0;
    }
}
```

变式题

二进制手表顶部有 4 个 LED 代表小时（0-11），底部的 6 个 LED 代表分钟（0-59）。
每个 LED 代表一个 0 或 1，最低位在右侧。
给定一个非负整数 n 代表当前 LED 亮着的数量，返回所有可能的时间。

```java
class _401二进制手表 {
    public List<String> readBinaryWatch(int num) {
        List<String> res = new ArrayList<>();
        for(int i = 0; i < 12; i++)
            for(int j = 0; j < 60; j++)
                if(Integer.bitCount(i) + Integer.bitCount(j) == num)
                    res.add(i + ":" + (j < 10 ? "0" + j : j));
        return res;
    }
}
```

格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。
给定一个代表编码总位数的非负整数 n，打印其格雷编码序列。格雷编码序列必须以 0 开头。

```java
public class _89格雷编码 {
    public List<Integer> grayCode(int n) {
        List<Integer> ret = new ArrayList<>();
        for(int i = 0; i < 1<<n; ++i)
            ret.add(i ^ i>>1);
        return ret;
    }
}
```

4.设计篇

4.1 LRU

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。
写入数据 put(key, value) - 如果密钥已经存在，则变更其数据值；如果密钥不存在，则插入该组「密钥/数据值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

基于哈希表与双向链表

```java
class _146LRU缓存机制 {
    private class ListNode{
        private int key;
        private int value;
        private ListNode pre;
        private ListNode post;
        public ListNode(int key, int value) {
            this.key = key;
```

```
            this.value = value;
        }
    }
    private int capacity;
    private Map<Integer,ListNode> map;
    private ListNode head;
    private ListNode tail;
    public LRUCache(int capacity) {
        this.capacity = capacity;
        map = new HashMap<>();
        head = new ListNode(-1,-1);
        tail = new ListNode(-1,-1);
        head.post = tail;
        tail.pre = head;

    }
    public int get(int key) {
        if(map.containsKey(key)){
            ListNode node = map.get(key);
            move2head(key);
            return node.value;
        }
        return -1;
    }
    public void put(int key, int value) {
        if(map.containsKey(key)){
            map.get(key).value = value;
            move2head(key);
            return;
        }
        if(map.size() == capacity){
            ListNode node = removeTail();
            map.remove(node.key);
        }
        ListNode newNode = new ListNode(key, value);
        map.put(key,newNode);
        add2head(newNode);
    }
    private void add2head(ListNode node) {
        ListNode oldNode = head.post;
        oldNode.pre = node;
        node.post = oldNode;
        node.pre = head;
        head.post = node;
    }
    private ListNode removeTail() {
        ListNode oldTail = tail.pre;
        ListNode newTail = oldTail.pre;
        newTail.post = tail;
```

```
        tail.pre = newTail;
        oldTail.pre = null;
        oldTail.post = null;
        return oldTail;
    }
    private void move2head(int key) {
        ListNode node = map.get(key);
        node.pre.post = node.post;
        node.post.pre = node.pre;
        add2head(node);
    }
}
```

基于LinkedHashMap

```
class LRUCache extends LinkedHashMap<Integer,Integer> {
    private int capacity;
    public LRUCache(int capacity) {
        super(capacity, 0.75F, true);
        this.capacity = capacity;
    }
    public int get(int key) {
        return getOrDefault(key,-1);
    }
    public void put(int key, int value) {
        super.put(key, value);
    }
    @Override
    protected boolean removeEldestEntry(Map.Entry<Integer, Integer>
eldest) {
        return this.size() > capacity;
    }
}
```


4.2 LFU

请你为 最不经常使用（LFU）缓存算法设计并实现数据结构。它应该支持以下操作：get 和 put。

get(key) - 如果键存在于缓存中，则获取键的值（总是正数），否则返回 -1。
put(key, value) - 如果键已存在，则变更其值；如果键不存在，请插入键值对。当缓存达到其容量时，则应该在插入新项之前，使最不经常使用的项无效。在此问题中，当存在平局（即两个或更多个键具有相同使用频率）时，应该去除最久未使用的键。
「项的使用次数」就是自插入该项以来对其调用 get 和 put 函数的次数之和。使用次数会在对应项被移除后置为 0 。

```
class _460LFU缓存 {
    Map<Integer, Node> cache;
```

```
    Map<Integer, LinkedHashSet<Node>> freqMap;
    int size;
    int capacity;
    int min;
    public LFUCache(int capacity) {
        cache = new HashMap<> (capacity);
        freqMap = new HashMap<>();
        this.capacity = capacity;
    }
    public int get(int key) {
        Node node = cache.get(key);
        if (node == null) {
            return -1;
        }
        freqInc(node);
        return node.value;
    }
    public void put(int key, int value) {
        if (capacity == 0) {
            return;
        }
        Node node = cache.get(key);
        if (node != null) {
            node.value = value;
            freqInc(node);
        } else {
            if (size == capacity) {
                Node deadNode = removeNode();
                cache.remove(deadNode.key);
                size--;
            }
            Node newNode = new Node(key, value);
            cache.put(key, newNode);
            addNode(newNode);
            size++;
        }
    }
    void freqInc(Node node) {
        int freq = node.freq;
        LinkedHashSet<Node> set = freqMap.get(freq);
        set.remove(node);
        if (freq == min && set.size() == 0) {
            min = freq + 1;
        }
        node.freq++;
        LinkedHashSet<Node> newSet = freqMap.get(freq + 1);
        if (newSet == null) {
            newSet = new LinkedHashSet<>();
            freqMap.put(freq + 1, newSet);
```

```
        }
        newSet.add(node);
    }
    void addNode(Node node) {
        LinkedHashSet<Node> set = freqMap.get(1);
        if (set == null) {
            set = new LinkedHashSet<>();
            freqMap.put(1, set);
        }
        set.add(node);
        min = 1;
    }
    Node removeNode() {
        LinkedHashSet<Node> set = freqMap.get(min);
        Node deadNode = set.iterator().next();
        set.remove(deadNode);
        return deadNode;
    }
}
class Node {
    int key;
    int value;
    int freq = 1;
    public Node() {}
    public Node(int key, int value) {
        this.key = key;
        this.value = value;
    }
}
```


4.3 设计推特

设计一个简化版的推特(Twitter)，可以让用户实现发送推文，关注/取消关注其他用户，能够看见关注人（包括自己）的最近十条推文。你的设计需要支持以下的几个功能：

postTweet(userId, tweetId): 创建一条新的推文
getNewsFeed(userId): 检索最近的十条推文。每个推文都必须是由此用户关注的人或者是用户自己发出的。推文必须按照时间顺序由最近的开始排序。
follow(followerId, followeeId): 关注一个用户
unfollow(followerId, followeeId): 取消关注一个用户

```
class Tweet{
    int tweetId;
    int timestamp;
    Tweet next;
    Tweet(int tweetId, int timestamp){
        this.tweetId = tweetId;
        this.timestamp = timestamp;
```

```
    }
}
class Twitter {
    private int timestamp = 0;
    public Twitter() {
        follow = new HashMap<>();
        publish = new HashMap<>();
        queue = new PriorityQueue<>((a, b) -> b.timestamp -
a.timestamp);
    }
    private Map<Integer, Set<Integer>> follow;
    private Map<Integer,Tweet> publish;
    private Queue<Tweet> queue;
    public void postTweet(int userId, int tweetId) {
        timestamp++;
        if(!publish.containsKey(userId))
            publish.put(userId, new Tweet(tweetId, timestamp));
        else{
            Tweet old = publish.get(userId);
            Tweet nw = new Tweet(tweetId,timestamp);
            nw.next = old;
            publish.put(userId, nw);
        }
    }
    public List<Integer> getNewsFeed(int userId) {
        List<Integer> ret = new ArrayList<>();
        queue.clear();
        if (publish.containsKey(userId)) {
            queue.offer(publish.get(userId));
        }
        Set<Integer> set = follow.get(userId);
        if(set != null && set.size() > 0){
            for(int u : set){
                Tweet tweet = publish.get(u);
                if(tweet != null)
                    queue.offer(tweet);
            }
        }
        int count = 0;
        while (!queue.isEmpty() && count < 10) {
            Tweet head = queue.poll();
            ret.add(head.tweetId);
            if (head.next != null) {
                queue.offer(head.next);
            }
            count++;
        }
        return ret;
    }
```

```
    public void follow(int followerId, int followeeId) {
        if(followerId == followeeId)
            return;
        if(!follow.containsKey(followerId)){
            follow.put(followerId, new HashSet<>());
        }
        follow.get(followerId).add(followeeId);
    }
    public void unfollow(int followerId, int followeeId) {
        if(followerId == followeeId)
            return;
        if(!follow.containsKey(followerId))
            return;
        follow.get(followerId).remove(followeeId);
    }
}
```

4.4 设计公平洗牌算法

```
class FisherYates{
    public void shuffle(int[] arr){
        int n = arr.length;
        for(int i = 0; i < n; i++){
            int x = (int)(Math.random()*(n-i)) + i;
            swap(arr, i, x);
        }
        /*
        for(int i = n-1; i >= 0; i--){
            int x = (int)(Math.random()*(i+1));
            swap(arr, i, x);
        }
        */
    }
    private void swap(int[] arr, int x, int y){
        int tmp = arr[x];
        arr[x] = arr[y];
        arr[y] = tmp;
    }
    public void check(){
        int N = 1000000;
        int[] arr = {1,0,0,0,1};
        int[] count = new int[arr.length];
        for(int i = 0; i < N; i++){
            shuffle(arr);
            for(int j = 0; j < arr.length; j++){
                if(arr[j] == 1){
                    count[j] ++;
                    break;
                }
```

```
            }
        }
        for(int freq : count)
            System.out.print(freq/N + " ");
    }
}
```

5.模版篇

5.1 并查集模版

```
public class UnionFind{
    private int[] parent;
    private int[] rank;
    private int count;
    public UnionFind(int size){
        parent = new int[size];
        rank = new int[size];
        for (int i = 0; i < size; i++) {
            parent[i] = i;
            rank[i] = 1;
        }
    }
    private int find(int p){
        if (p != parent[p])
            parent[p] = find(parent[p]);
        return parent[p];
    }
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }
    public int getCount() {
        return count;
    }
    public void unionElements(int p, int q) {
        int pRoot = find(p);
        int qRoot = find(q);
        if(pRoot == qRoot)
            return;
        count--;
        if(rank[pRoot] < rank[qRoot])
            parent[pRoot] = qRoot;
        else if(rank[pRoot] > rank[qRoot])
            parent[qRoot] = pRoot;
        else {
            parent[qRoot] = pRoot;
            rank[pRoot] += 1;
        }
```

```
    }
}
```

5.2 链表解题模版

```
public class LinkedList {
    //迭代方法
    private ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode cur = head;
        while (cur != null) {
            ListNode post = cur.next;
            cur.next = prev;
            prev = cur;
            cur = post;
        }
        return prev;
    }
    //递归方法
    public ListNode reverseListRecursion(ListNode head) {
        if(head == null || head.next == null)
            return head;
        ListNode next = head.next;
        head.next = null;
        ListNode ret = reverseListRecursion(next);
        next.next = head;
        return ret;
    }
    //查找链表中点偏左
    private ListNode endOfFirstHalf(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
        while (fast.next != null && fast.next.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }
    //查找链表中点偏右
    private ListNode endOfFirstHalfRight(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }
    ListNode successor = null;
```

```
    //反转链表前N个节点
    private ListNode reverseN(ListNode head, int n) {
        if (head == null)
            return null;
        if (n == 1) {
            successor = head.next;
            return head;
        }
        ListNode last = reverseN(head.next,n-1);
        head.next.next = head;
        head.next = successor;
        return last;
    }
}
```

5.3 01背包模版

```
public class Backpack {
    public int backPackII(int m, int[] A, int[] V) {
        int n = A.length;
        if(n == 0)
            return 0;
        int[][] dp = new int[n][m+1];
        for(int i = 0; i <= m; i++)
            dp[0][i] = i >= A[0] ? V[0] : 0;
        for(int i = 1; i < A.length; i++){
            for(int j = 0; j <= m; j++){
                dp[i][j] = dp[i-1][j];
                if(j >= A[i]){
                    dp[i][j] = Math.max(dp[i][j],dp[i-1][j-
A[i]]+V[i]);
                }
            }
        }
        return dp[n-1][m];
    }
    //空间优化
    public int backPack(int m, int[] A, int[] V) {
        int n = A.length;
        if(n == 0)
            return 0;
        int[] dp = new int[m+1];
        for(int i = 0; i < n; i++)
            for(int j = m; j >= A[i]; j--)
                dp[j] = Math.max(dp[j],dp[j-A[i]]+V[i]);
        return dp[m];
    }
}
```

5.4 环检测模版

5.4.1 无向图

```
public class CircleDetection {
    private Graph g;
    private boolean[] visited;
    private boolean hasCycle = false;
    public CircleDetection(Graph G){
        this.g = G;
        visited = new boolean[G.getV()];
        for(int v = 0; v < g.getV(); v++)
            if(!visited[v])
                if(dfs(v, v)){
                    hasCycle = true;
                    break;
                }
    }
    private boolean dfs(int v, int parent){
        visited[v] = true;
        for(int w: g.adj(v))
            if(!visited[w]) {
                if(dfs(w, v))
                    return true;
            }
            else if(w != parent)
                return true;
        return false;
    }
    public boolean hasCircle(){
        return hasCycle;
    }
}
```

5.4.2 有向图

```
public class DirectedCycleDetection {
    private Graph g;
    private boolean[] visited;
    private boolean[] onPath;
    private boolean hasCycle = false;
    public DirectedCycleDetection(Graph G){
        this.g = G;
        visited = new boolean[G.getV()];
        onPath = new boolean[G.getV()];
        for(int v = 0; v < g.getV(); v++)
```

```
            if(!visited[v])
                if(dfs(v, v)){
                    hasCycle = true;
                    break;
                }
    }
    private boolean dfs(int v, int parent){
        visited[v] = true;
        onPath[v] = true;
        for(int w: g.adj(v))
            if(!visited[w]) {
                if(dfs(w, v))
                    return true;
            }
            else if(onPath[w])
                return true;
        onPath[v] = false;
        return false;
    }
    public boolean hasCircle(){
        return hasCycle;
    }
}
```

5.5 联通分量模版

```
public class CC {
    private Graph g;
    private int[] visited;
    private int count = 0;
    public CC(Graph G){
        if (G.isDirected())
            throw new IllegalArgumentException("can't be applied to
directed graph");
        this.g = G;
        visited = new int[G.getV()];
        Arrays.fill(visited, - 1);
        for(int v = 0; v < g.getV(); v++)
            if(visited[v] == -1) {
                dfs(v, count);
                count++;
            }
    }
    private void dfs(int v, int count){
        visited[v] = count;
        for(int w: g.adj(v))
            if(visited[w] == -1)
                dfs(w, count);
    }
```

```
    public boolean isConnected(int v, int w){
        return visited[v] == visited[w];
    }
    public ArrayList<Integer>[] components(){
        ArrayList<Integer>[] res = new ArrayList[count];
        for(int i=0;i<count;i++)
            res[i] = new ArrayList<Integer>();
        for(int v=0;v<g.getV();v++)
            res[visited[v]].add(v);
        return res;
    }
    public int getCount(){
        return count;
    }
}
```

5.6 排序模版

```
public class Sort {
    private void swap(int[] arr, int x, int y){
        int temp = arr[x];
        arr[x] = arr[y];
        arr[y] = temp;
    }
    public void insertSort(int[] arr) {
        int n = arr.length;
        for (int i = 1; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j > 0 && arr[j-1] > temp ; j--)
                arr[j] = arr[j-1];
            arr[j] = temp;
        }
    }
    public void insertSort2(int[] arr) {
        int n = arr.length;
        for (int i = 1; i < n; i++) {
            for (int j = i; j > 0 && arr[j] < arr[j-1]; j--) {
                swap(arr, arr[j], arr[j-1]);
            }
        }
    }
    public void mergeSort(int[] arr) {
        temp = new int[arr.length];
        mergeSort(arr, 0, arr.length-1);
    }
    private void mergeSort(int[] arr, int l, int r) {
        if (l >= r)
            return;
```

```
        int mid = (l + r) / 2;
        mergeSort(arr, l, mid);
        mergeSort(arr, mid + 1, r);
        if (arr[mid] > arr[mid + 1])
            merge(arr, l, mid, r);
    }
    public void mergeSortBU(int[] arr) {
        for (int size = 1; size <= arr.length; size++) {
            for (int i = 0; i + size < arr.length; i += size) {
                merge(arr,i,i+size-1,Math.min(arr.length-
1,i+size+size-1));
            }
        }
    }
    public void quickSort(int[] arr) {
        quickSort(arr, 0, arr.length-1);
    }
    public void quickSortThreeWay(int[] arr, int l, int r) {
        if (l >= r)
            return;
        Random random = new Random();
        swap(arr,l,random.nextInt(r-l)+l);
        int guard = arr[l];
        int lt = l;
        int gt = r + 1;
        int i = l + 1;
        while (i < gt) {
            if (arr[i] < guard) {
                swap(arr,++lt, i);
            }else if (arr[i] > guard) {
                swap(arr,i,--gt);
            }else {
                i++;
            }
        }
        swap(arr,l,lt);

    }
    private void quickSort(int[] arr, int l, int r) {
        if (l >= r)
            return;
        int p = partition(arr, l, r);
        quickSort(arr,l,p-1);
        quickSort(arr,p+1,r);
    }

    private int partition(int[] arr, int l, int r) {
        Random random = new Random();
        swap(arr,l,random.nextInt(r-l)+l);
```

```
        int guard = arr[l];
        int pos = l;
        for (int i = l + 1; i <= r; i++) {
            if (arr[i] < guard){
                swap(arr, ++pos,i);
            }
        }
        swap(arr,pos,l);
        return pos;
    }

    private int[] temp;
    private void merge(int[] arr, int l, int mid, int r) {
        if (r + 1 - l >= 0)
            System.arraycopy(arr, l, temp, l, r + 1 - l);
        int i = l;
        int j = mid+1;
        for (int k = l; k <= r; k++) {
            if (i > mid){
                arr[k] = temp[j];
                j++;
            }else if(j > r){
                arr[k] = temp[i];
                i++;
            }else if(temp[i] < temp[j]){
                arr[k] = temp[i];
                i++;
            }else{
                arr[k] = temp[j];
                j++;
            }
        }
    }
}
```