# 数组

## 12.11 26，33，80

数组：

数组原地删除肯定快慢指针

要求时间复杂度在O(logN)以内---二分查找

注意边界---考虑到空和null的情况

空数组(.length==0)和null不同

## 12.12 81，1

注意数组临界条件 0，1，2 使用二分法时特别注意只有两个元素的情况

太nb了 若o==m，o++，continue 相当于去掉一个重复的干扰项 反正相同项无用

Break只能跳出一重循环

Map用put加元素 set，list用add

集合框架注意泛型的使用 Map<Integer, Integer> map = new HashMap<>();

看见评论里有人说hash表不一定有暴力破解来的快，so，我专门试了一下。程序肯定是针对大数据量才能看出差别，我就分别试了数组长度为一万、十万、百万的数组，并且查找次数保证最大。在一万的情况下，暴力破解法差不多要15-30毫秒，而hash表则是0-15毫秒；在十万的情况下，暴力破解法是1500-1700毫秒，而hash表则是15-40毫秒；百万级别的，hash表用了500-1500毫秒不等，但暴力破解法我就执行了一次，152130毫秒。也去专门去查了hashMap.containsKey()的时间复杂度，使用指针指向数组引用，时间复杂度为O(1)，未命中时，才回去遍历红黑树，时间复杂度为O(n)，有兴趣的可以取看看 <https://blog.csdn.net/qingtian_1993/article/details/80763381> 这篇帖子

## 12.13 null

千万不可以想先睡一觉，起来以后再做题，起来后脑子是木的，根本不好使，困就出去买咖啡！

## 12.14 15，16，18

排序加双指针

顺序数组考虑二分法

两数之和用map 三数之和先排序后双指针 四数之和在三数和的基础上再来一次遍历即可 n数和同理

18：

图片包含 屏幕截图, 文字

描述已自动生成n3

记住时间复杂度n是指所有元素总共 前面a遍历所有，b遍历所有，接下来cd是双指针，加起来是遍历所有

双指针的那次遍历记得是n

图片包含 屏幕截图

描述已自动生成 n

总共其实是对二维数组所有元素遍历一次，所以是n

## 12.15 27

27两种方法都很好

## 12.16 吃完饭一定买咖啡，要不困了不会想做题的

## 12.17 11，31，34，35，48，54

31. 对于一些复杂看不出规律的 多写几个实例，进行找规律

powcai的题解

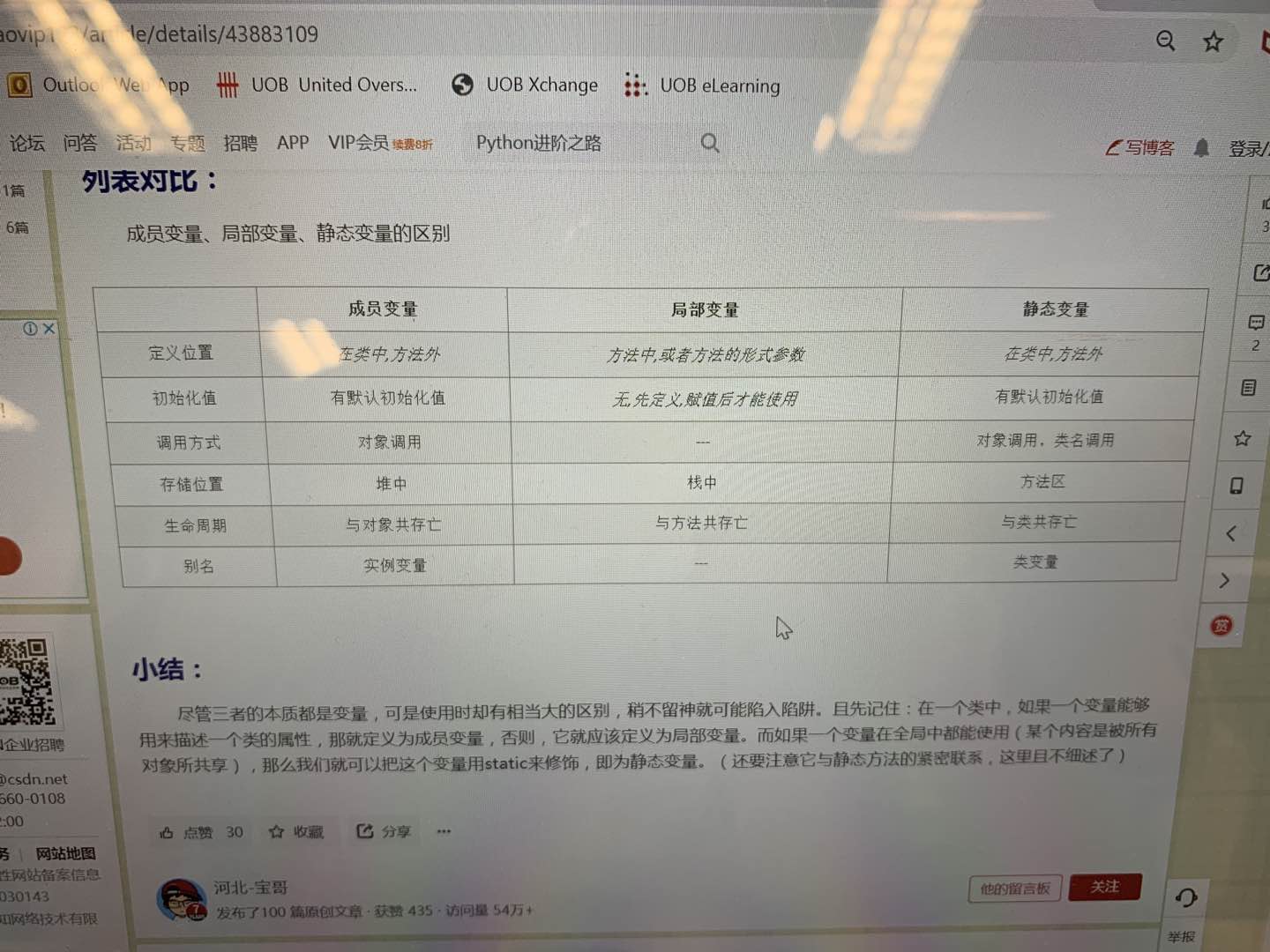
34.二分法要复杂度logn 这时 while(n++,n--)就不行 变n了

Math.max(a,b) 取最大的

int i=0；这个i是局部变量需要初始化后才能用

while（）｛i++｝

循环出来后i会变



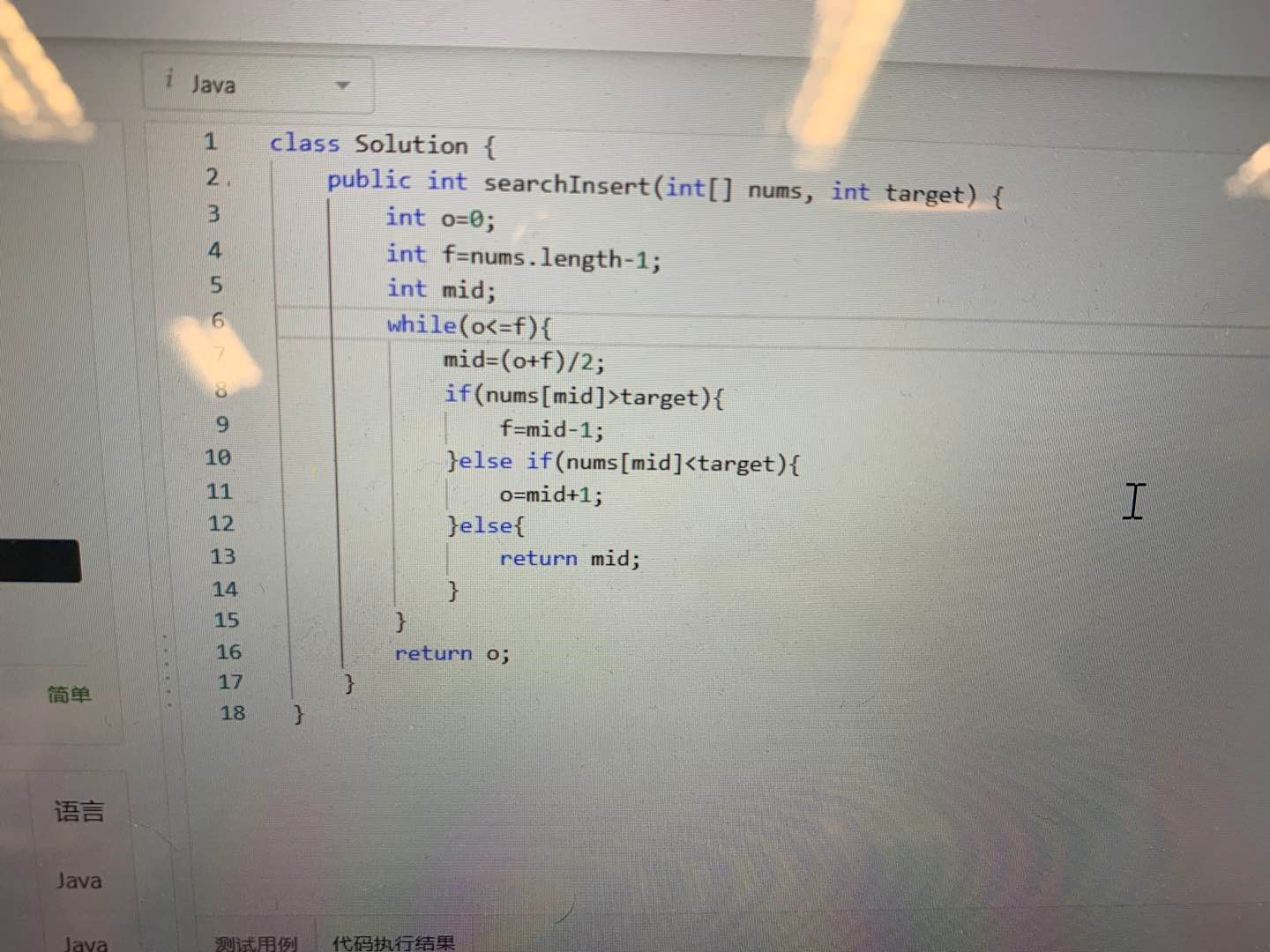
数组中间元素 （o+f）/2+o 但不用太精确

二分法注意事项️：

1.推荐写法 int mid = ( left+right )>>>1 ;

2.return 左边界

3.o一定是小于**等于f**



题目排序数组其实就是暗示二分查找

48.注意双重循环中有的变量需要恢复初始状态

图片包含 屏幕截图

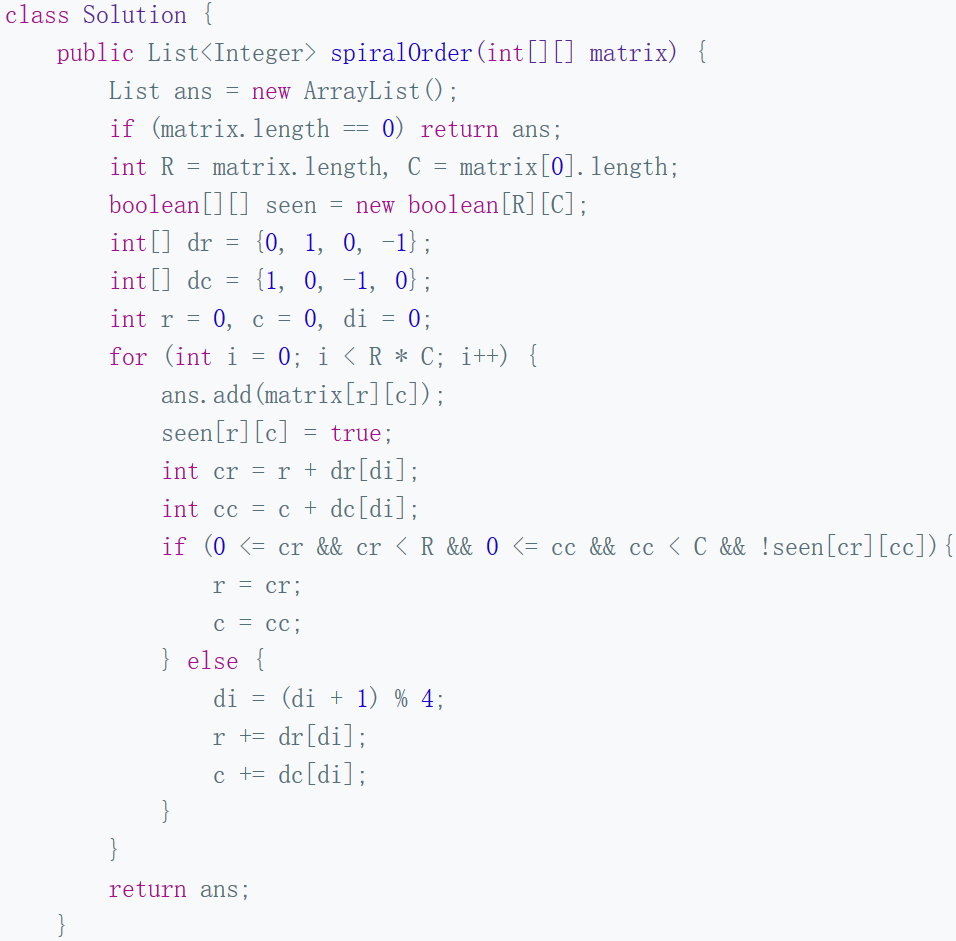
描述已自动生成 f=n-1;

创建二维数组 { { }，{ }，{ }，{ } }

a.length是行数，a[0].length是列数

54.

Arraylist的实际长度 list.size() list有list.contains() map是map.containskey()

创建方向数组，每次遇到重复或出边界改变方向

先初始位置放入，判断下一位置是否安全，安全则赋值，不安全换方向

注意方向数组顺时针你怎么设置int[] dr = {0, 1, 0, -1}; int[] dc = {1, 0, -1, 0};

boolean [][] seen=new boolean[m][n]; 注意创建二维数组的长度是不需要-1的

## 12.18 59

int[][] a=new int [n][n]; 自动初始化都为0

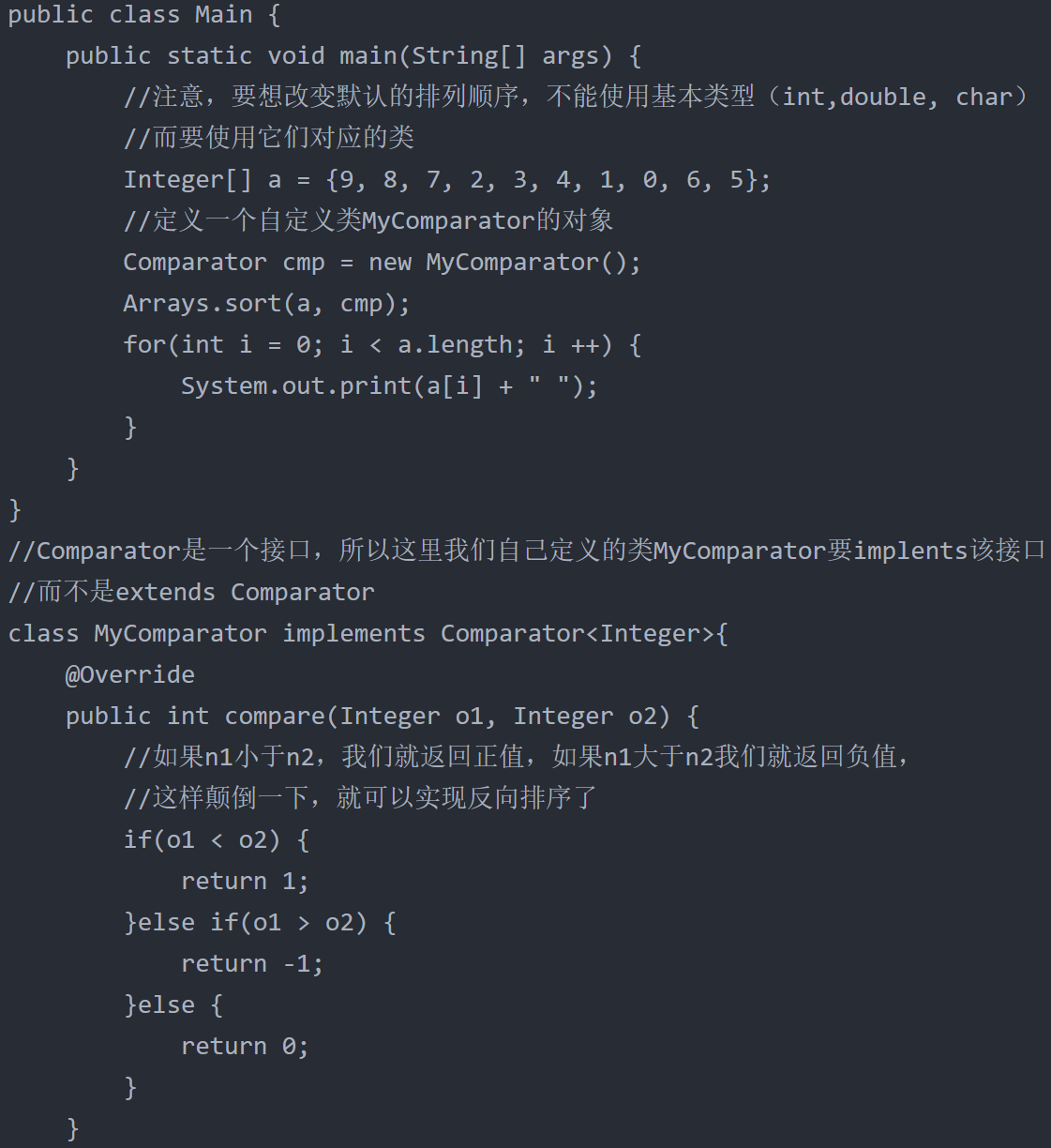
## 12.19 56，55，66，73

56.

Lambda 表达式是一个匿名函数（对于 Java 而言并不很准确，但这里我们不纠结这个问题）。简单来说，这是一种没有声明的方法，即没有访问修饰符，返回值声明和名称。

Java 中的 Lambda 表达式通常使用语法是 (argument) -> (body)，比如： (arg1, arg2...) -> { body }

Array.sort()还可以传入Comparator接口，重写其中的compare方法 自定义比较算法



两种方法

1. Arrays.sort(intervals, (a, b) -> a[0] - b[0]); // a[0] - b[0]大于0就交换顺序

2.

图片包含 屏幕截图

描述已自动生成

注：是array**s.**sort() 最后list转换成数组 list.toarray()

.toarray() 有两个重载

一个是list.toArray()，转换成object[ ]

另一个是list.toArray(T[] a); 转换成需要类型的数组

1) list.toArray(new String[list.size()])；

2) list.toArray(new String[0]);

3) list.toArray(new int[0][])

new int[0][]表示不指定行、行自动填充，如果为new int[4][], 即使结果为[[1,6],[8,10],[15,18]]， 也会强制输出[[1,6],[8,10],[15,18],null]，即不足行null补充

可以用0代表不指定具体，也可以直接指定具体值

66.创建一个有大小的数组，则此数组自动被填充0 int[] b=new int[5];

73.Map的key不可以相同！！！ 相同会被覆盖

## 12.20 74，75

75. 一共三种类型在数组中进行分类 三指针法

得取集合长度和指定元素List.size() list.get(index)

## 12.21 88

交换二维数组xy相当于沿左上至右下平分线旋转180度

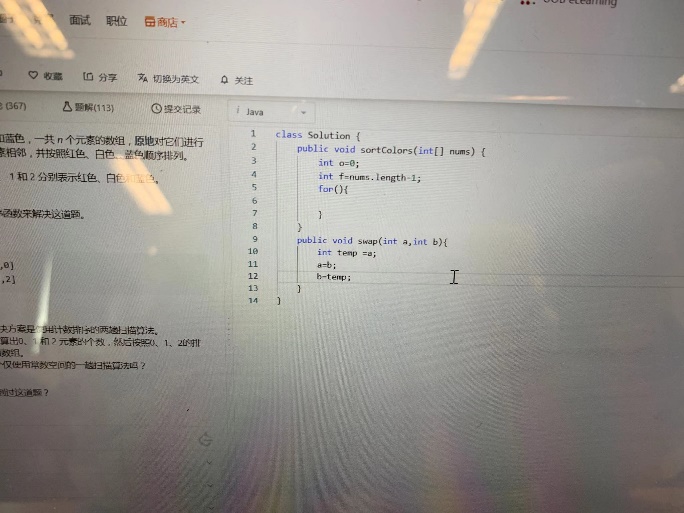
int[][] 十字线 行相等或列相等 而不是行或列等于行

对二维数组进行二分搜索，将index与[][]相互转换即可

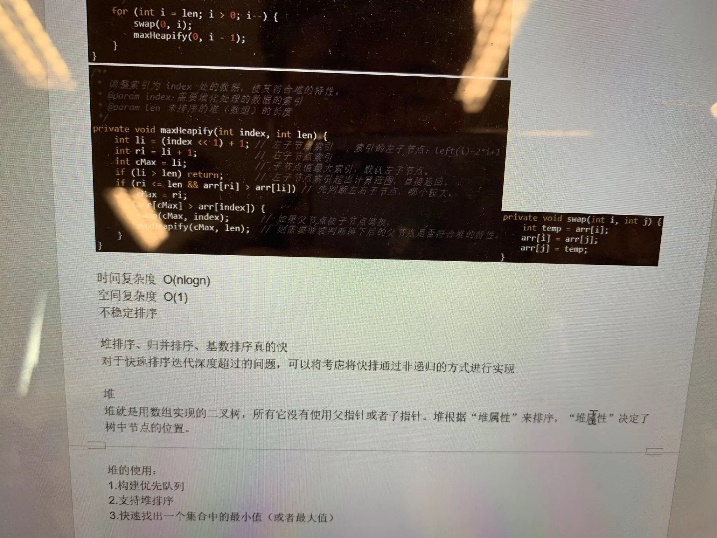
system.arraycopy(a,x,b,y,z)

## 12.22

注意这样是不会换的



需要用数组才行，数组不是引用

 数组：二分法；双指针；三指针

# 链表--143

此为自定义的单向链表

Definition for singly-linked list.

 \* public class ListNode {

 \*     int val;

 \*     ListNode next;

 \*     ListNode(int x) { val = x; }

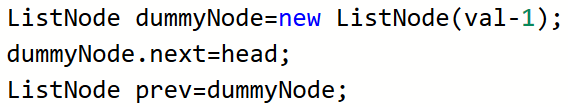
Java中的Linkedlist可当作双向链表使用

链表所有题都可以试着用缓存解决

## 12.27 203，141，142，160，19，206

203.注意链表是可以无头节点的，你若想可以虚拟出一个头节点

分清实际节点与节点名的区别

 dummynode是实际节点 prev是当前节点名，用于遍历

可以一个元素自成环 边界条件 if (head == null) { return null; }

160. 循环判断条件

图片包含 天空

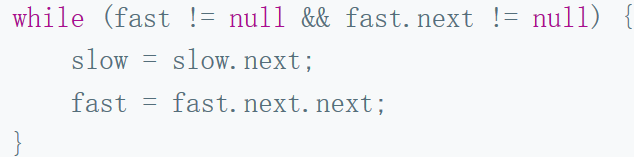
描述已自动生成你不能a.next!=null 这样a.next=null时 a就加不进去了

206.调转指针，反转链表

## 12.28 21，148

prev.next = l1 == null ? l2 : l1; l1是不是等于null呢 是的话next=l2 不是的话next=l1

148.对链表排序使用并归排序

当一指针每次跳两格时防止空指针套路 fast&fast.next！=null

图片包含 屏幕截图

描述已自动生成正常都是fast&slow=head

## 1.2 61，86，138

61.

方法一：连接成环，找到头尾后解环

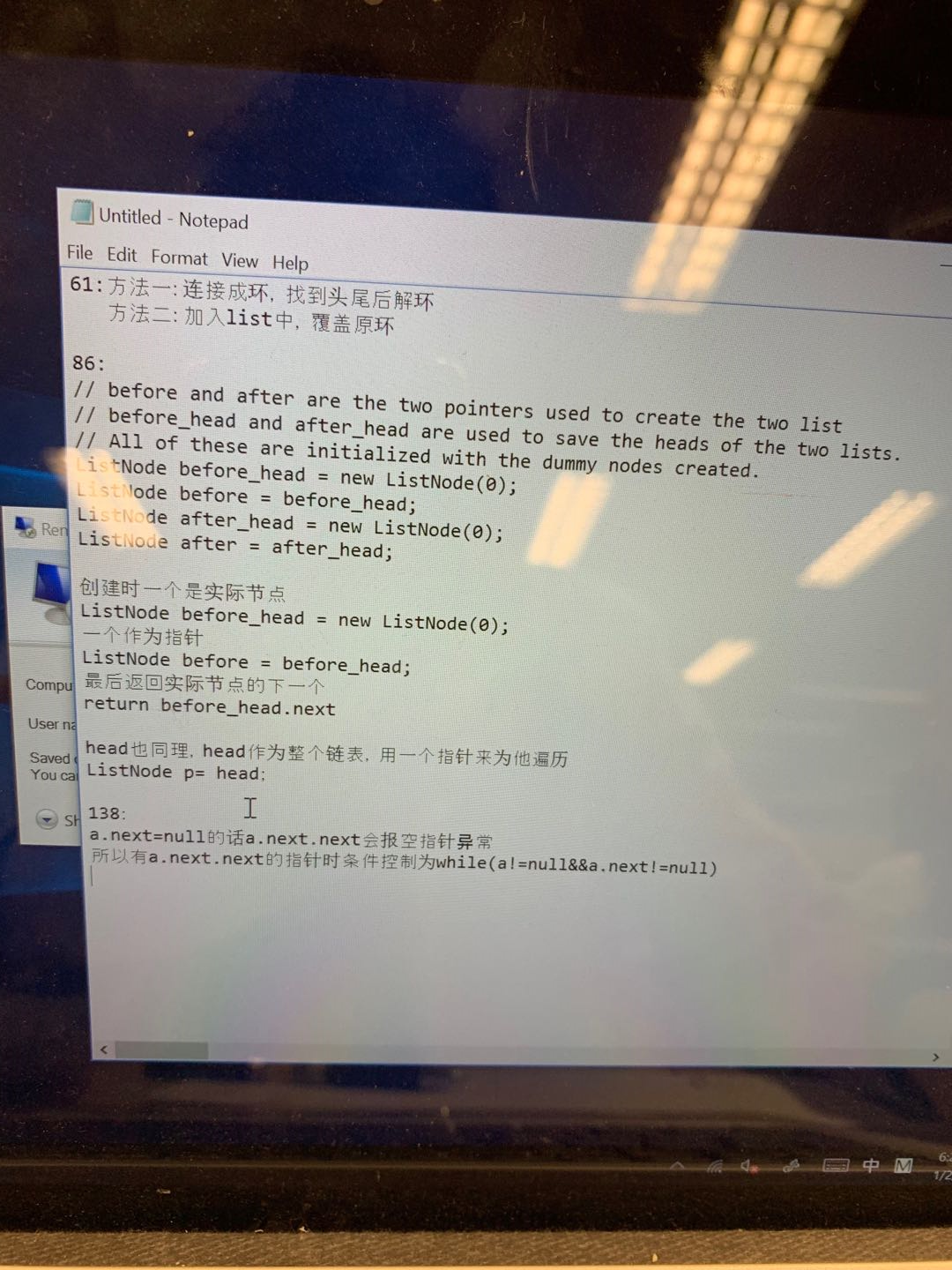
方法二：加入list中，覆盖原环

138.

a.next=null的话，a.next.next会报空指针异常

所以有a.next.next的指针时条件控制为whlie( a!=null && a.next!=null )

86.



24.

注意交换指针的顺序

图片包含 屏幕截图

描述已自动生成

143.

超好的题---重新做

图片包含 屏幕截图

描述已自动生成图片包含 屏幕截图

描述已自动生成

链表：使用hashset做缓存，快慢双指针

# 队列

图片包含 屏幕截图

描述已自动生成

queue.offer() 添加一个元素并返回true 如果队列已满，则返回false //add()

queue.poll() 移除并返问队列头部的元素 如果队列为空，则返回null //remove()

queue.peek() 返回队列头部的元素 如果队列为空，则返回null //element()

queue.size() 返回队列长度

queue.put() 添加一个元素 如果队列满，则阻塞

queue.take() 移除并返回队列头部的元素 如果队列为空，则阻塞

# 栈

图片包含 屏幕截图

描述已自动生成

进栈 stack.push(Object); //返回的是入栈的内容

出栈 stack.pop(); //输出并删除栈顶元素

stack.peek(); //输出不删除栈顶元素

是否为空 stack.isEmpty();

栈长度 stack.size()

输出栈 System.out.println(stack); （[ , , , …])

当你想首先处理最后一个元素时，栈将是最合适的数据结构。

栈顶是可以插入和删除端 栈底是最开始元素

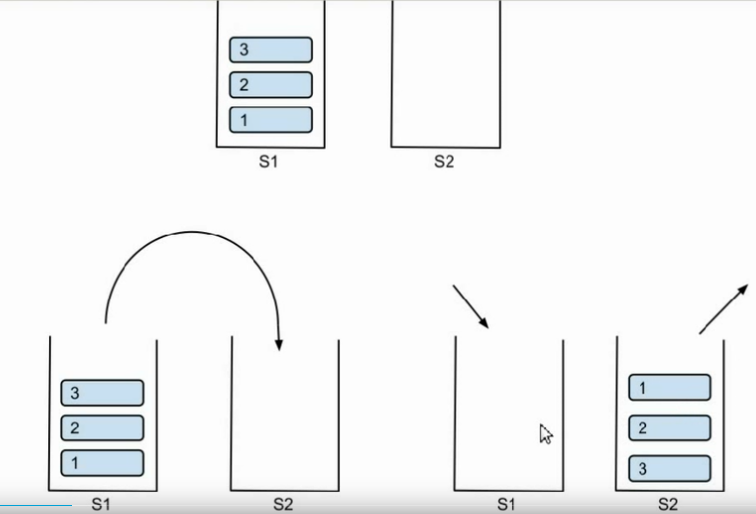
## 1.6 20

无map 栈最典型的应用就是匹配括号

图片包含 屏幕截图, 文字

描述已自动生成

两个栈实现队列



# 图

## BFS ---队列queue

图片包含 屏幕截图

描述已自动生成

有时，确保我们永远不会访问一个结点两次很重要。否则，我们可能陷入无限循环。如果是这样，我们可以在上面的代码中添加一个哈希集来解决这个问题。这是修改后的伪代码：

有两种情况你不需要使用哈希集：

1.你完全确定没有循环，例如，在树遍历中；

2.你确实希望多次将结点添加到队列中。

图片包含 屏幕截图

描述已自动生成

## DFS ---栈stack

一般dfs都用递归

递归

图片包含 屏幕截图

描述已自动生成

非递归

图片包含 屏幕截图

描述已自动生成

## 1.8 200

200. dfs和bfs都可 已实现dfs



# 树