# 常用变量名 通用的问题算法代码相关的总结

**记录一些程序中常用到的变量名**

num nums value array resultSet sum l r mid record result complement target index freq count

head cur pre next dummyHead succ

aux(auxiliary辅助的） capacity size

root visited point

memo token(符号）

i,j,k m,n l,r 感觉前面的通常表示索引下标 x y表示数

left right top bottom

parts part 比如string.split后的数组名可以是parts

valid need total

方法名

check()

**一些通用的问题算法代码相关的总结**

传入的参数不对throw new IllegalArgumentException("Illegal argument numbers");

输入的数据不对导致该问题在该数据下没有解throw new IllegalStateException("The input has no solution");

lc上的题解通常感觉不throw异常 直接不考虑或者写return new in[]{-1,-1}这种

数组相关的问题 当题解的方法传参是数组时 一定要记得先讨论 if(nums==null||nums.length==0) 时的情况的返回结果 有的时候即使不写 也能AC 但其实是错的 上面两种情况下可能return 0 不写的话程序直接运行到最后最小都会return 1

正式做题的时候遇到输入是数组的题时 还是上来先把nums==null nums.length=0or1时的结果写出来return 不然程序的主体逻辑就要考虑这些边界情况 很容易有些细节地方会考虑不周就会出错

上来不记得写 也要代码写完记得检查添加

算法题目第一遍写完代码 一定要检查一下有没有哪里会造成索引越界 比如如果一上来就用数组的下标取值 dp[n-1]=1 那就必须先判断n-1是否大于等于0 即n是否大于等于1 可能就要写个if(n==0) return res 让这种情况直接返回

#### 代码相关总结

当问题的返回结果是一个int数组时 之前常用的方法是因为不知道结果的size大小所以先设立一个arraylist存放结果 然后最后通过arraylist->Integer[]->使用stream流转化为int[] 或者foreach迭代遍历arraylist 将每轮的变量赋给int[i]最后返回

现在可以不用设立arraylist了 直接设int[] 初始化时不必知道size大小 只需设立一个肯定比结果size大的size就行 一般是原来传入的数组的大小或者是多个数组大小的和这种 最后return的时候

用Arrays.copyOf 或Arrays.copyOfRange截断即可

还是用先设立一个arraylist 最后直接arrayList.toArray(new int[arrayList.size()])就行 用arrayList直接add就行 用数组添加的时候还要记录索引

再更新toArray()方法里传入参数 但是只能传Integer[] 不能传int[]

mapToInt()把Stream<Integer>调用Integer::valueOf来转成IntStream  
// 而IntStream中默认toArray()是转成int[]。  
 int[] array1 = result.stream().mapToInt(Integer::valueOf).toArray();

/\*传递了一个空的字符串数组 new String[0] 作为参数给 toArray() 方法。这是因为如果传递的数组的长度  
小于 ArrayList 的大小，toArray() 方法会创建一个新的数组并返回；如果传递的数组的长度大于或等于  
ArrayList 的大小，toArray() 方法会使用传递的数组来存储元素并返回。\*/  
String[] array = als.toArray(new String[0]);

toArray()里面是int[n][]可以

list.toArray(new int[list.size()][])

* 对比同一种思路同一种算法在不同问题里具体到代码上的实现细节不同

比如dfs遍历得到一颗树的所有路径或所有根到叶子节点的路径

问题 求一个数字字符串能代表的所有字母组合 问题 二叉树上根节点到叶子节点的每条路径表示一个数 求所有数之和

两个问题都可以将中间结果设为类成员变量 但delete的位置不同

不要一遇到通过dfs遍历获取所有路径相关的问题 就立马觉得要设一个结果list保存res 有些问题直接通过递归函数的返回值以及在return里面完成一些算数运算就可以解决 如lc129sumroottoleaf

当dfs res是一个二维列表List<List<>>类型时 每次到递归出口要res.add()时 add的list一定要是新new的一个list res.add(new ArrayList(list)）

算法里要用dfs 写的时候大部分直接用递归就行 不需要自己显式地运用到栈 bfs时需要显式地在算法里运用队列

* 正式答题的时候如果有些提供的api忘了怎么使用 不要因为忘记了就觉得没有其他法子了 通常算法题都有很多种解答 忘记api的使用方法 那就换一条路或者自己写一些实现方法达到需要的效果就行 比如想到的题解要用正则表达式 想不起来正则表达式怎么写 就可以自己写一个辅助方法如public boolean isValid(char c){

return true;

return false;

}

* l+(r-l)/2

* 暴力枚举法 二分搜索法 对撞指针法 twoSum 其实对撞指针就是相比于暴力解法遍历所有可能性 它会进行剪枝 排除一些不可能的情况不需要遍历 Ar11maxArea求能成最多水的容器

* // 发现我经常if写完了 下面的else经常忘记写 导致多走了一次分支

* 字符转为整数 int y=num2.charAt(i)-'0';

* string的substring（）方法的参数是[beginIndex,endIndex) 左开右闭

* int mid=l+(r-l)/2; 此时mid是可能等于l的(当只有两个数时） 但mid不会等于r （除非只有一个数时 但一般只有一个数就跳出while循环了）相当于是一种向下取整 所以在代码里有时候nums[mid]和nums[l]比较的时候会加= 此时大部分是l=mid了 而不是l和mid不相等 但nums[l]=nums[mid]

#### 问题解法

对于那些已经做过但不容易直接想到解法的算法题 再看的时候可能最先想到的还是本来的那些不高效甚至错误的解法 所以之后看做过的题重点要思考用记忆法关键词联想等方法一步一步推导联想让自己能记住这道题的解法 不然写过的下次还是不会写 有的不好推导的题目也可以干脆用记忆的方法 找到本题题目中的关键词 题解里的核心方法思路浓缩成较短的语句结论 然后在关键词和语句结论之间联想建立关联 反正就是让自己下次看到题目能想到解法就行 比如之前char short都是占用两个字节对这个概念很模糊不确定 我直接用记忆的方法打包在一起记忆变成牵手 牵手肯定是要两个人 所以是2个字节 现在根本忘不了

比如寻找一个字符串的最长回文子串 直觉上是能感受到这道题要用DP的 所以立马就要想到这道题的状态如何定义 在字符串里的常见的状态变量就是下标索引 应该是能想到状态要定义成dp[i][j]表示字符串里从i到j的位置处的字符串是否是回文串

其实有时候有些问题想不到解法时就可以想想暴力解法遍历所有可能性 再用一些hm hs这样额外的数据结构保存中间结果或者一开始就装好数据 为后面的过程做辅助 这样答案就出来了 比如给定一个未排序的整数数组 nums ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

数组 字符串相关的题 双指针 滑动窗口 DP

二叉树相关的题能用递归解的特别多

链表 虚拟头节点 快慢指针 链表反转 链表很多就是要操纵指针的

感觉不好解的题 想想是否可以用hashmap hashset辅助结构 hashmap常见的是存储多少次这种数字 但hashmap也可以value存储链表节点这种 hashmap还有一点是表示一种映射关系

dfs 用递归方式（常见的是用递归方式）或者栈 bfs必须要用到队列 当dfs res是一个二维列表List<List<>>类型时 每次到递归出口要res.add()时 add的list一定要是 新new的一个**list res.add(new ArrayList(list)）**

**求最大值最小值的问题 经常可以用优先队列辅助**

#### 刷题心得

要经常总结 做算法题的时候其实多做一些就会慢慢有一些自己的感悟 会形成自己的一些总结 比如关于某种数据结构经常怎么用 以及某些算法双指针 滑动窗口 dp dfs 经常用于处理哪些问题的总结 我的习惯就是写某道题的时候有了一些想法感悟的话 我就会立马自己整理下用文字记录下来 因为这种你自己有感悟的时刻是特别珍贵的 网上其他人写的总结的内容可能更好 但是那毕竟不是自己的 很容易看了也很快忘了 是自己想出来的东西自己也会印象更深刻 所以更要整理记下来 反正就是各种的我有感触的都会记录一下 比如有时候我看别人代码的某个变量名 方法名 他用的很准确 又是我没想到过的 我也会记一下方便我自己之后遇到也能用上

#### od相关

##### 不太熟的

并查集

字符转字符串 String.valueOf('c')或'c'+""

和区间相关的题 都常都涉及到排序 即将给定的一堆区间按照区间起始位置或终止位置升序排列

有些题目并不是要找规律的 找规律很难 而是先找到min max确定结果的范围 然后用二分搜索法

题目条件是给定一个整数n 求关于这个整数的XXX 往往可以看做是给定了一个从0到n的数组 尤其是和连续相关的题 和连续相关的题经常用到双指针滑动窗口

求连续区间的和 还可以用前缀和法 用前缀和通常很容易理解 直接用双重for循环计算每个区间是否符合要求即可

写完要看看题目里的每个边界条件是否都覆盖了

dfs bfs的题经常需要把输入的变量定义为全局的

遇到有数据范围是10的9次方这种的就要注意很可能是要考察整型溢出了 10的9次方是10亿 三个数相加就会30亿 超过21亿

数据范围在万级别的 如小朋友人数范围是 [0, 40000]。 通常是为了限制这道题不能用暴力枚举on2 会超时

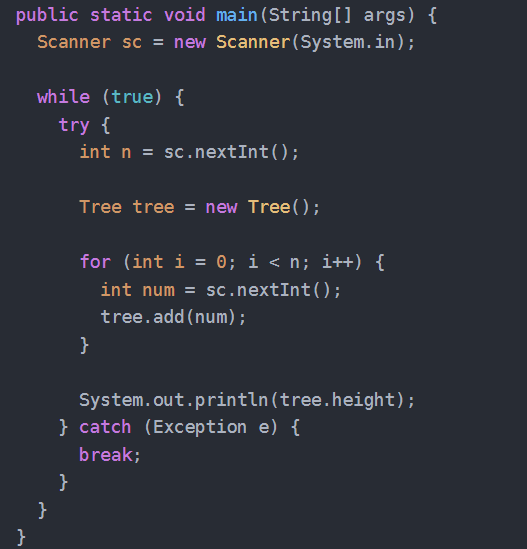
后面复习的时候重点关注下 那些自己刚开始想不出来用某种算法或某个数据结构辅助的题

单调栈解决寻找下一个最近的更大值或更小值 比如找朋友的题 通常寻找下一个最近的更大值是栈底到栈顶递减

hashmap要想按照key 排序输出value 必须先转为entryset

map不能直接用stream 必须先转为keyset entryset values这些

有多组输入案例 用while(true)循环 捕捉exception



并查集处理集合合并 查询问题 通常解决那种输入是二维矩阵 且只包含0 1的二维矩阵的问题 0就表示初始时ij之间不直接联通 1表示ij直接联通 并查集建立好了之后 并查集里面的数组表示每个节点的父节点 就能根据父节点是否相同判断两个节点是否能联通

二分法查找插入位置 -index-1

表示时间时分

static Pattern pattern= Pattern.compile("(([01][0-9])|([2][0-3]))[0-5][0-9]");

对于有二维矩阵的题 不仅可以用DBFS 也可以试着用DP求解 还可能用并查集求解

##### 题目记录

BS m个月完成n份工作每个月需要的最小人力 二分法+双指针check时 这里是在求所有工作都完成最少需要几个月 所以肯定是一个最大的和一个最小的搭配在一起 这样才平均 最后需要的时间才最少

DBFS 加密算法 static int[][] moveDirection={{-1,0},{0,-1},{0,1},{1,0}};必须是上左右下

跳马问题 BFS

// 最小步数和矩阵，stepMap[i][j]记录各个马走到棋盘(i,j)位置的最小步数之和  
static int[][] stepMap;  
// 记录所有马都可达的公共位置坐标 用二维坐标一维化的方式存储 之后reach还要与别人取交集  
static HashSet<Integer> reach;

m项工作 n个人 最少几天完成 二分法min就是人够多每项工作都有一个人负责时需要的时间

max就是人只有一个时需要的时间

派遣员工问题 代号为 x 的国家和代号为 y 的国家分别需要 cntx 名和 cnty 名员工

// 使用此范围，实际通过率55%  
// long min = cntx + cnty;  
// long max = Long.MAX\_VALUE;  
  
// 使用此范围，实际通过率可以100%  
long min = 1;  
long max = 1000000000L;

通过密码本找明文对应的密文 从每个起始位置开始检查是否能找到目标密文 此题里偏移量是有要求的 必须是上，左，右，下偏移量 因为如果有多条密文，返回字符序最小的密文。

跳马问题 马有等级k代表可走k步 问是否能将所有马跳到同一位置，如果存在，输出最少需要的总步数（每匹马的步数相加），不存在则输出-1。

对每匹马进行BFS走日（每只马行走的过程中每到达一个点 该点的步数就要加相应的步数，k步都走完之后将公共可达位置reach和当前马可达位置取交集，交集部分就是新的公共可达位置）

抢7游戏必须用BigInteger

篮球放入圆桶后可以左右取出 用队列模拟圆桶 并且每放入一次就试试能不能从左右取出

多任务排序问题 一个hm记录每个任务的入度是几 另一个hm HashMap<String, ArrayList<String>> next = new HashMap<>();记录每个任务的所有后继节点

贪心歌手赚钱 用一个优先队列小跟堆记录歌手在路上所有站赚的钱 只要遇到比堆顶元素大的 就把堆顶弹出 换成加入当前的选择 最终结束后优先队列里的所有元素就是结果

简易内存池问题用一个linkedlist存储分配的空间 模拟内存池

LinkedList<int[]> usedMemory=new LinkedList<>();  
// 内存池总大小为100字节 所以100之后肯定是用不到了 之所以要一开始就添加这块把[100,101]当做已使用的内存 是为了给分配的内存  
// 设立一个边界 添加的时候比较到最后只要不和[100,101]相交 就一定符合要求  
usedMemory.add(new int[]{100,101});

求字符串中所有整数的最小和要用BigInteger

输入 bb12-34aa  
\* 输出 -31  
\* 说明 1+2+（-34） = -31

数据单元的变化替换

static Pattern p = Pattern.compile("(<.\*?>)"); // 01<D>23<B>45<C>6,B,C,D 白天的时候怎么也想不明白为什么加? 就能

输入法联想的题 就是用一个treeset存储原文 treeset用stream流过滤出s.startwith(pre)

5G题用并查集构建最小生成树

##### 数学知识

判断两个数是否互质 辗转相除法 互质指最大公约数等于1的两个自然数。

求两个数的最大公约数 辗转相除

判断一个数n是否是素数 非2的偶数都不是素数 之后就是除3 除5 一直除到sqrt(n)即可

#### 二叉树相关

二叉搜索树的中序遍历序列是有序的

二叉树的节点时 不一定每次入队出队都是一个节点 也可能入队出队两个节点 而且入队的顺序不一定需要就是每一排从前往后 完全可以根据具体的场景选择入队哪个元素

比如迭代判断二叉树是否是对称二叉树 这里用队列就是每次出队两个节点n1 n2 然后将n1.left和n2.right入队 再是n1.right和n2.left入队

二叉树相关的题大多都可以用递归来解决 尤其是需要dfs遍历的 多思考用递归解决

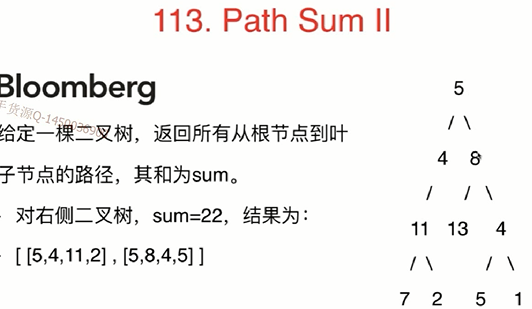
用dfs遍历所有路径寻找答案的代码 通常如果dfs方法的入参是stringbuilder或list 在dfs方法里递归调用dfs方法结束之后都要remove sb list里的元素 如果想要避免这种写法的话 可以dfs方法的入参改为string 在dfs方法里新建stringbuilder处理增加数据 之后递归调用的时候传递sb.toString() 对于dfs方法的入参是list的 可以在dfs方法里通过list<> sublist=new Arraylist(list)新建一个sublist 递归调用的时候传递的就是这个sublist 这样就能防止递归的时候分支污染

还可以像treepaths方法那样 直接将递归方法设为有返回值类型的 这样就不用传递中间结果









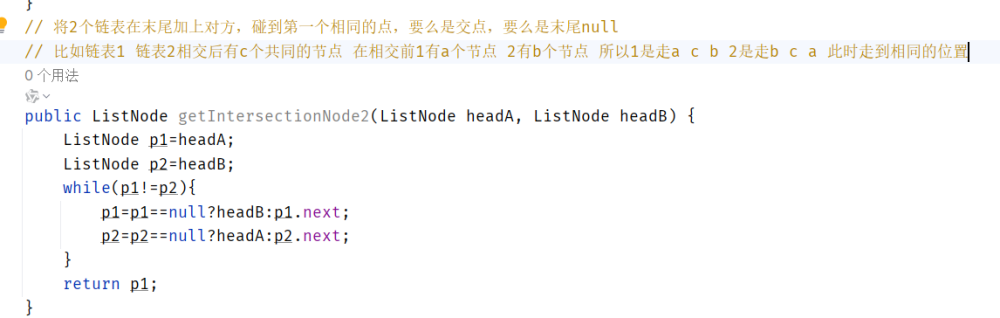
#### 链表

链表解题经典三板斧**，哑巴节点（删除链表中值为val的所有节点），栈，快慢指针（删除链表中倒数第n个节点 判断链表是否有环）。**

链表的题如果要改变原链表里节点的顺序生成新链表 细节较多 麻烦就麻烦在先要一段一段截断原本的链表 后续一点点生成的子链表又要不断地连起来 要记录的节点就很多

链表反转也可以用递归的写法

1个链表判断是否有环 可以用快慢指针或hashset 2个链表判断是否有相交可以用hashset或者指针遍历到达末尾就指向对方



找环形链表开始入环的第一个节点 此题fast slow同时指向head 快慢指针相遇时快慢指针分别走了2n和n个环的周长



#### 栈队列优先队列

java中Stack就是类 可以直接new Stack对象 Queue是接口 LinkedList虽然可以用作栈 但不能用stack引用变量指向LinkedList对象 因为LinkedList没有继承Stack类 只实现了list deque接口

所以可以用queue或dequeb变量引用指向LinkedList对象 queue和deque都是接口 linkedlist实现了deque接口

使用优先队列的例子 ：一个包含有序链表的数组 将所有链表合并成一个升序链表 优先队列中存储的是所有链表的头结点ListNode

PriorityQueue<ListNode> queue1 = new PriorityQueue<ListNode>((e1,e2)->(e1.val-e2.val));

#### 双指针 滑动窗口

求三数之和 先排序 然后使用一层循环里面用双指针 具体就是单循环枚举第一个数 里面用双指针枚举第二个数和第三个数

求四数之和 先排序 然后使用二层循环里面用双指针 具体就是外循环枚举第一个数 内循环枚举第二个数 里面用双指针枚举第三个数和第四个数

求两数之和的题是返回数在数组中的下标（是唯一的解） 上面的三数四数的题是找到所有三元组 四元组 所以可以排序 两数之和 不能排序排序就会打乱了

两数之和的题用hashmap 创建一个哈希表，从前往后遍历 对遍历到的每一个 x，我们首先查询哈希表中是否存在 target - x，存在就直接return答案 不存在就将 x 插入到哈希表中，即可保证不会让 x 和自己匹配。

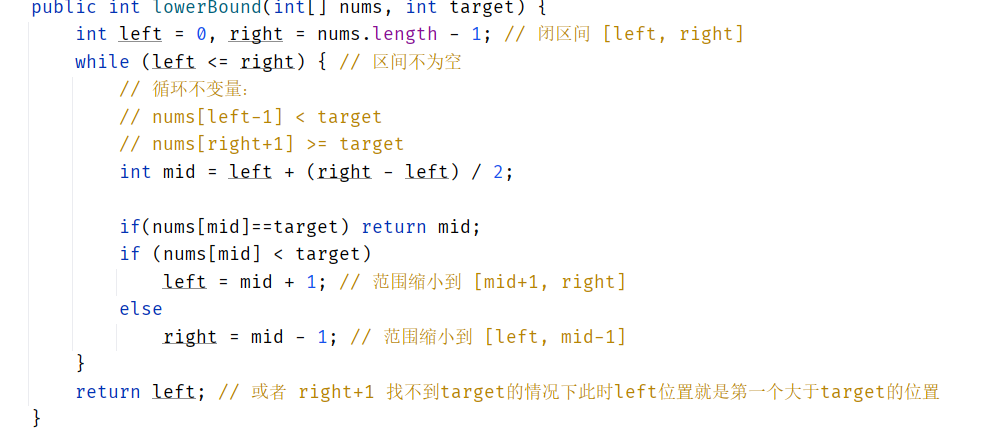
两数之和为target还有一个题 数组是有序的 所以可以用对撞指针来解

#### 二分查找

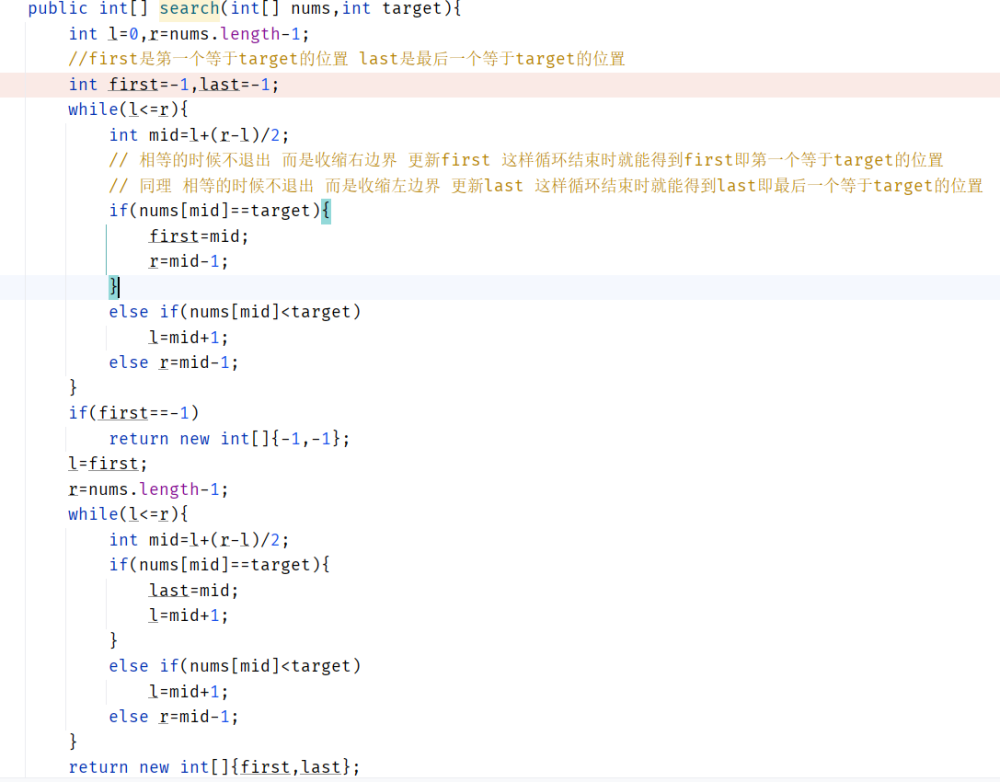
二分查找l=0 r=nums.length-1在[l..r]中寻找满足条件的情况 有可能是在循环里出现某个情况时就得到答案了 也有可能是跳出循环时的情况是答案

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。

\* 请必须使用时间复杂度为 O(log n) 的算法。 nums 为无重复元素的升序排列数组

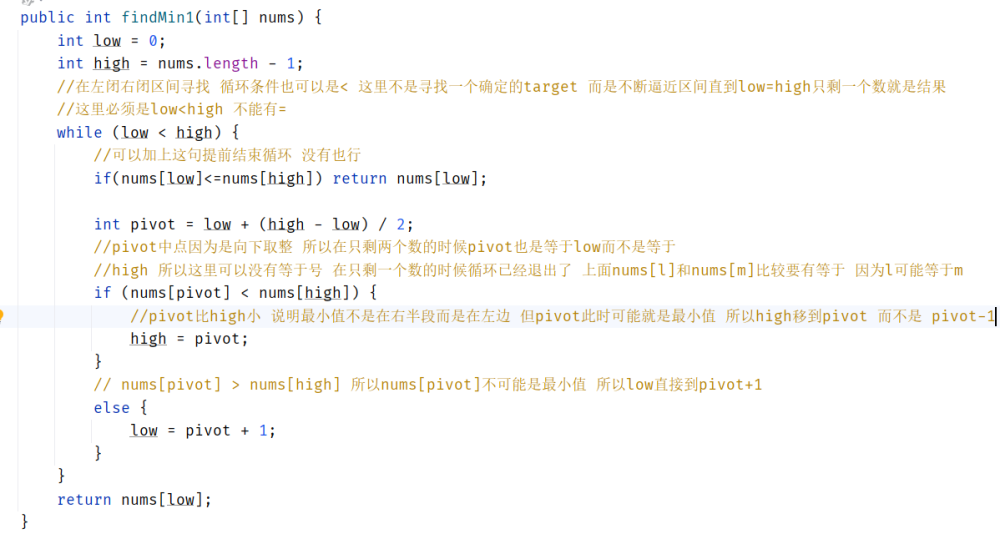


给你一个按照非递减顺序排列的整数数组 nums，和一个目标值 target。请你找出给定目标值在数组中的开始位置和结束位置。



搜索旋转排序后的数组的最小值

此时没有了固定的target 要用二分查找法 所以此时nums[mid]不能和target比较 那想到的就是nums[mid]和nums[left]或nums[right]比较 本题和nums[right]比较会简单点



#### DP相关

dp状态定义的结果是如果最终答案是要获取dp[n]/dp[n-1] 那外层循环就是i从0或1开始直到n 如果最终答案是要获取dp[0] 那外层循环就是i从n或n-1开始直到0

记忆化搜索的实现代码通常就是用递归得到答案 不需要写for循环或for循环少一层 dp的实现就是用for循环逐渐递推得到结果 不需要写递归

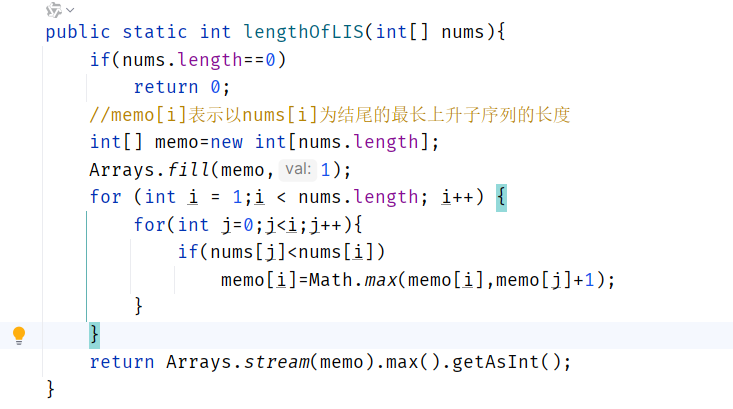
第一个动态规划问题：爬楼梯一次可以爬一个台阶或两阶 问总共有多少种不同的方法 状态转移方程就是要斐波那契数列的

二维动态规划问题：0-1背包问题 背包里能装下的物品价值最大 f(i,c)=max(f(i-1,c),f(i-1,c-wi)+vi)

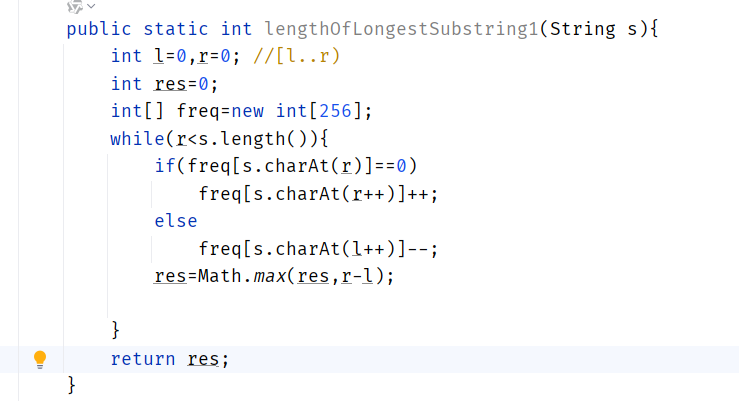
DP通常用来解决那种寻找最优方案 最大 最小 多少种 通常是没法一下子就能看出答案怎么找出来 但可以通过dp一步步递推得到的

dp通常解决的问题是那种要得到一个数字答案的问题 因为dp[][]数组里通常存的就是数字 也可以解决求字符串的最长回文子串这种非数字答案的问题 此时dp[i][j]表示s[i..j]是否是回文串 是bool值 最终问题的答案由在遍历dp[][]过程中不断记录更新的maxLen begin 根据最后这两个结果可以得到最终的答案

LIS数字序列的最长上升子序列长度 LCS两个字符串的最长公共子序列长度用dp求解 而求字符串的最长不重复子串长度是用滑动窗口







**子串是需要连续的 子序列是不需要连续的** 因为子串是连续的 所以求字符串的最长不重复子串长度只需要维护一个窗口使得里面的字符不重复即可 如果是求最长不重复子序列那还简单些 直接就是字符串里不重复的字符加一起就是了

LIS求数字序列的最长上升子序列长度 用dp解 lis(i)表示以第i个数字结尾的最长上升子序列长度 反而比求最长上升子串难 上升子串要连续那就是遍历的时候遇到的数字比前面的大1就加1 要不然就要重置为1

dp数组不仅可以是基础类型int[] dp这种 还可以是自定义的类变量类型的数组 比如wood[] dp wood里包含一个数字和一个列表 这样dp[i]每个状态里不仅可以存一个值还可以利用列表存储其他需要的信息

#### DBFS

全排列问题 dfs里要用for循环 for循环从0开始遍历 需要用到visited[]



dfs的参数加个index也可以 像上面的解法不加也可以



组合问题 dfs参数必须要带startIndex dfs里的for循环就是从startIndx开始循环的 不需要visited



组合问题 有选或不选 和枚举选哪个 这两种解法



#### 递归相关

**递归的一些心得**

有些题目用递归解决 代码就会很简洁简单 只要根据问题**合理的定义递归函数的作用** 然后正确地书写出逻辑 难点就在于有的问题想不到可以用递归来解决 不是那种明面上一看就是能想到递归的 通常我们都把Solution类里的那个要返回答案的方法看作是普通方法 按照惯性我们想到或者说只能想到他的作用就是给定要求的输入数据 得到想要的输出 没想过转换下思路这个方法也能当做递归方法 即**输入输出都能推广到更一般的情况** 如本来问题是给定一个原链表输出深拷贝后的新链表 输入输出肯定都是链表的头结点 但其实这里就可以推广到更一般的情况 即输入是原链表的某个中间节点n 输出就是新链表中和n对应的那个节点 例题lk138深拷贝带random指针的链表(但是这题递归不好想 还是用hm辅助存储 遍历两次的解法）

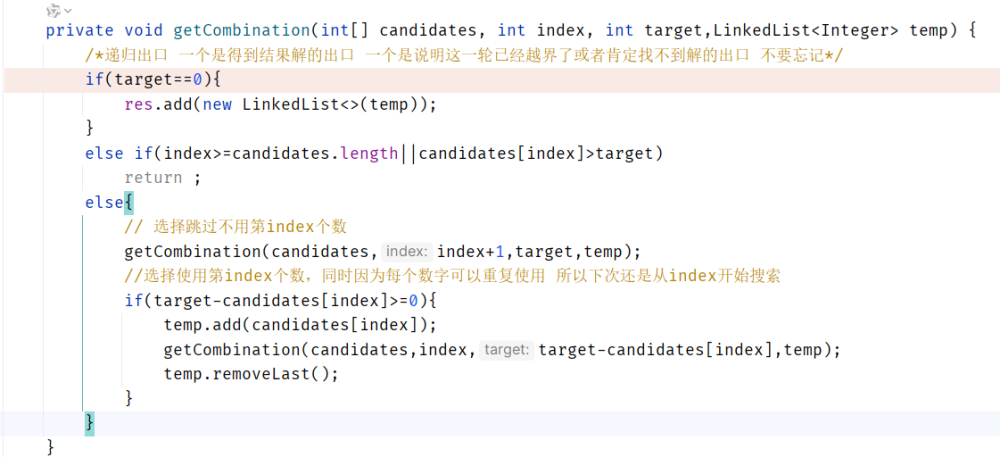
有些问题知道可以用递归来解决 不需要纠结怎么理清逻辑 直接在递归函数里写出来下一步要进行的子问题的递归（可能会有多个子问题） 然后再看子问题的递归都解决后 此时怎么把结果合并或者计算出来 这种情况在二叉树相关的问题中尤其多 因为二叉树本身就是一个递归结构 无论你要得到什么或者完成什么 几乎都可以通过先使左右子树达到目标 再完成一些操作就能使根节点达到目标 递归的好用就在于 你不需要思考子问题是怎么解决的 就因为这个递归函数就是这么定义的 直接上来就从子问题已经解决了之后该怎么做开始思考 通常到这里要思考的东西就少很多 很容易理清逻辑写出解答了

如 LBT114flatten 将二叉树展开为链表 用递归直接就可以思考root的左右子树已经展开为两个链表了 下一步要怎么得到最终需要3的链表

其实上面这两点核心都是一样的 对递归不敏感 以后看别人的代码一个递归函数里面调用子问题的递归 要能立马意识到这行代码结束后这个子问题就已经得到解决了

**递归从子问题已经解决完了之后开始思考 不用关心子问题是怎么解决的 直接上来就思考在子问题已经得到解决的情况下怎么根据已得到的子问题的解再进行哪些操作就能得到完整问题的解**

**上面这些只针对那种很容易想到子问题的 有些情况递归函数需要的参数除了题目中已给出的很明显的输入 还需要自己构造想出来其余需要的参数 这种情况下子问题不太好想 所以可能还是需要自顶向下思考**



#### 递归回溯

一定要**检查递归终止条件里的return**有没有写 以及流程结束末尾的return有没有写 通常当递归函数的返回类型是void时 很容易就忘了写 这里第二个return不用写 但当函数的返回类型不是void时就一定要记得写

像二叉树这种自身就是树结构就可以通过.left .right到达其他节点 在dfs(root)里写dfs(root.left) dfs(root.right） 就能遍历到整棵树的每个节点 本题这种自身不是树结构的树形问题可以**通过递归函数里面有循环 循环里面再调用递归函数 来达到遍历所有的目的** 这样进入到下一层的递归里 还是会经过循环 所以能遍历完所有可能

二维平面上的用递归回溯寻找解法 寻找是否存在 寻找有多少可能性的问题 通常会需要定义int d[][] = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}}; //偏移量数组 在二维平面上移动经常使用 上右下左boolean[][] visited 记录坐标位置是否会访问过

如果觉得很复杂 通常可以试着思考只从某一个点出发如何寻找答案即可 很多问题都是下面这种双重循环里调用一个递归方法寻找解决方案

