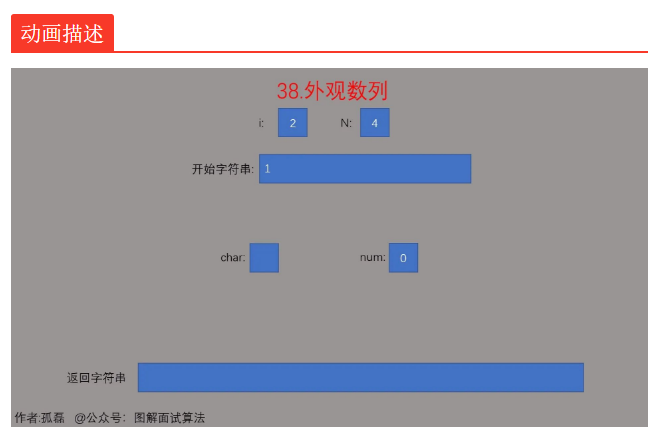
# 简单

## 1\_两数之和

暴力求解并不难，如果要达到O(n)的时间复杂度的话，那么就只能遍历一次，那么就要用到hash表了。这里建立的是map，分别存储nums[i]和索引i。

## 38\_外观数列





## 53\_最大子序和

1. maxSum = -2e32
2. Dp[i]表示以nums[i]结尾的最大连续子数组和
3. 记得使用空间优化

## 70\_爬楼梯

谁都知道是f[n] = f[n - 1] + f[n - 2].

主要是要进行空间优化。

a, b = 1, 2

for i in range(n - 2)

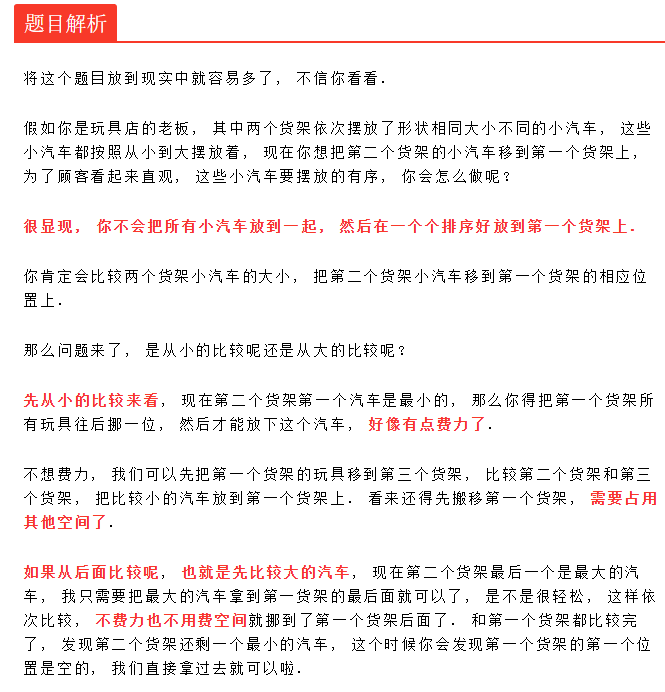
temp = b

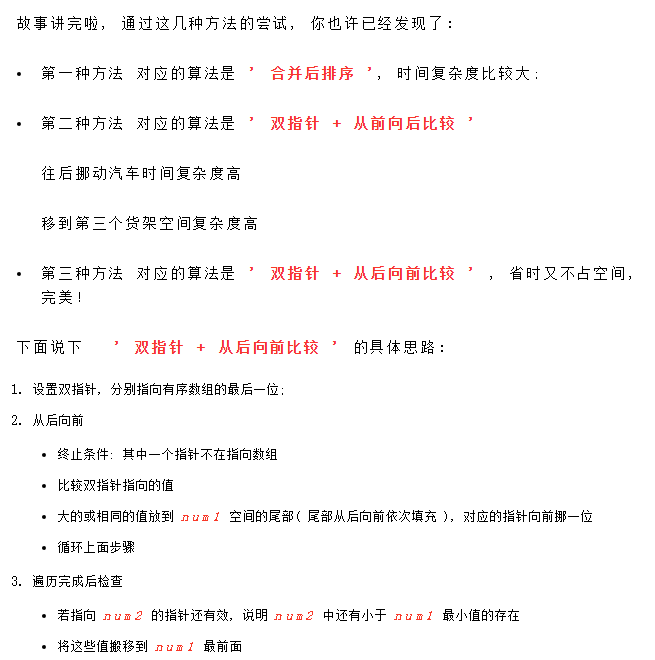
b = a + b

a = temp

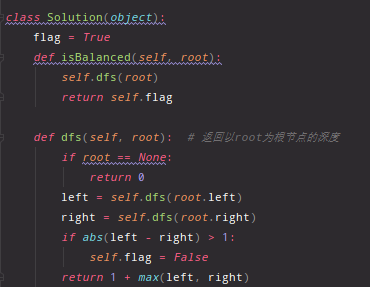
return b

## 88\_合并两个有序数组





110\_平衡二叉树



## 121\_买卖股票的最佳时机

维持一个利润最大值和买入最小值，从前到后扫描，不停更新最小值和利润最大值。

## 122\_买卖股票的最佳时机2

上题只能完成一次交易，而该题是能完成多次交易，故难一点。

既然可以完成多次交易，那么只要遇见就买入，遇见低价，说明股票开始跌了，那就卖出，买入低价新股。

初始化：假定第一个是最低价也是最高价，是一定要买入的，往后遍历，遇到高价，说明股票再上涨，接着往后遍历（high = price[i])，如果遇到了低价，那就转手卖掉(sum += high - low)。

记得最后就是股市关闭了，记得卖掉股票（return sum + high - low).

## 123\_买卖股票的最佳时机3

1、思路一：（letcode上除了最后一个测试用例超时之外，其余都能通过）

只能买卖两次。

那就在121的基础之上双向计算。

再维护一个最大值和最大利润。

遍历中间点。

注意二者可以叠加，因为当场卖掉，又可以当场买入。

所以中间节点可以是0---N-1

1. 思路二：状态机DP

两次买卖，可以分为五种状态：

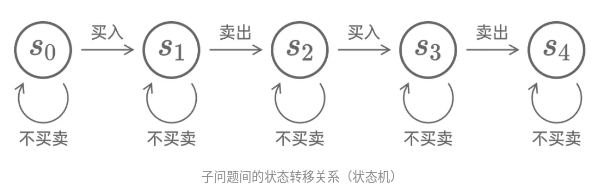
S0：可交易两次，不持股，可买入

S1 : 可交易一次，持股，可卖出

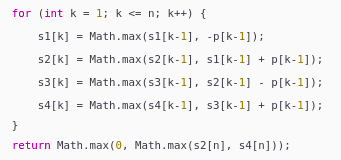
S2：可交易一次，不持股，可买入

S3：可交易0次，持股，可卖出

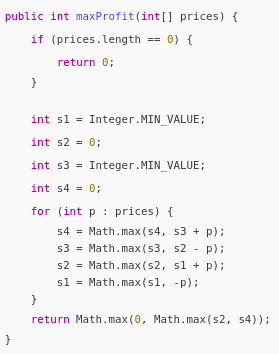
S4：可交易0次，不持股，可买入



五种状态，可选择保持状态，也可进入下一个状态，用max函数来进行选择保持还是进入下一个状态。S0始终为0，不参与进来。



空间进行优化后代码为：



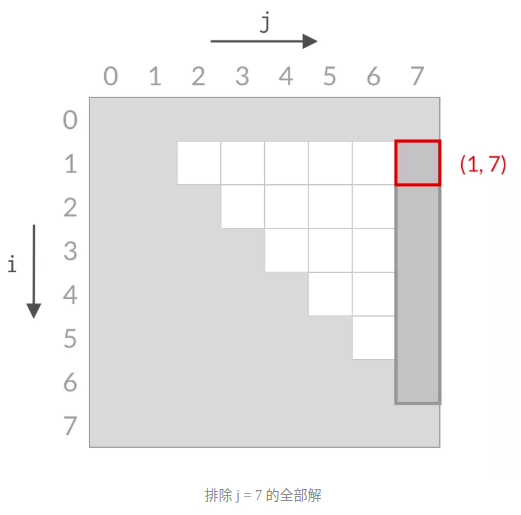
因为s1和s3持有股票，不能买入，故强制卖出，所以将其设为整数最小值。

## 141\_环形链表

定义快慢两个指针，快指针的速度是慢指针的两倍，如果快指针追上了慢指针，表明，有环。

## 167\_两数之和2

当然可以采用两数之和的hash表的方法去做，但是因为是有序的，如果利用双指针去缩减搜索空间，这样才是最适合这道题的解法。



## 198\_打家劫舍

Dp[i]表明打劫前i家商店的最高金额，但是第i家商店的金额是nums[i-1]

## 215\_数组中第k个最大元素

这个问题的话其实很简单，但是如果非得用到数据结构的话，那毫无疑问就是堆，每一种语言几乎都有实现堆的函数，python里面的话就是heapq。熟练运用它也是不错的选择。

熟练运用heapq的一下几个函数：

Nums = [1, 2, 3, 4]

Heapq.heapfy(Nums) // 对nums构建最小堆，如果要构建最大堆，只有将Nums里面的数取反。

Heapq.heapPush(nums， num) // 往最小堆里面加入一个num并自动建堆

Heapq.heqpPop(nums) // 弹出nums中的堆顶元素，即最小元素

Heapq.nlargest(k, nums) // 弹出列表nums中的最大的k个元素，注意nums不需要建堆

Heapq.nsmallest(k, nums) // 与上面相反。

## 290\_单词规律

肯定是利用hash map， 将pattern和str一一对应，当不对应时返回False，但这里最值得注意的一点就是，举例来说pattern = ‘abba‘， str = ‘dog dog dog dog’，这样的话，a， b对应都就是一个了，这是不合理的，但是确是能通过的， map{a: dog, b:dog}，这是一一对应的，所以map.values中的值也不能相同。

## 463\_岛屿的周长

这道题肯定可以用遍历，遇到1周长先加4，再判断其左上有没有1，有的话自动减2，但这毫无疑问当湖足够大，岛屿足够小时，就会浪费很多的时间复杂度。故我决定采用queue，先找到岛屿的第一个边缘，然后将整个岛屿加入queue，从queue中提取一个格子时周长加4，当找到一个新格子时，周长减2即可。

*# 特别注意，不要将queue中原本存在的再加一遍。*

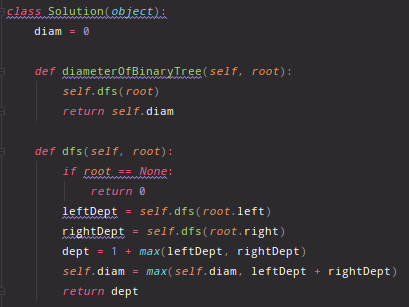
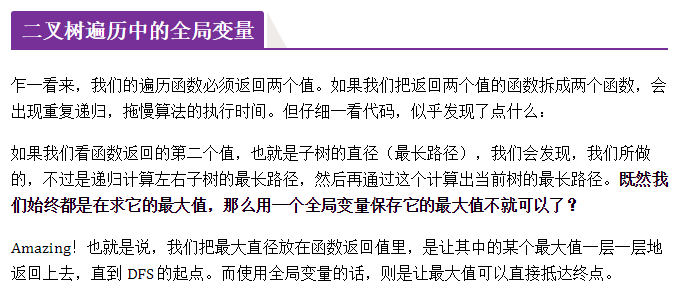
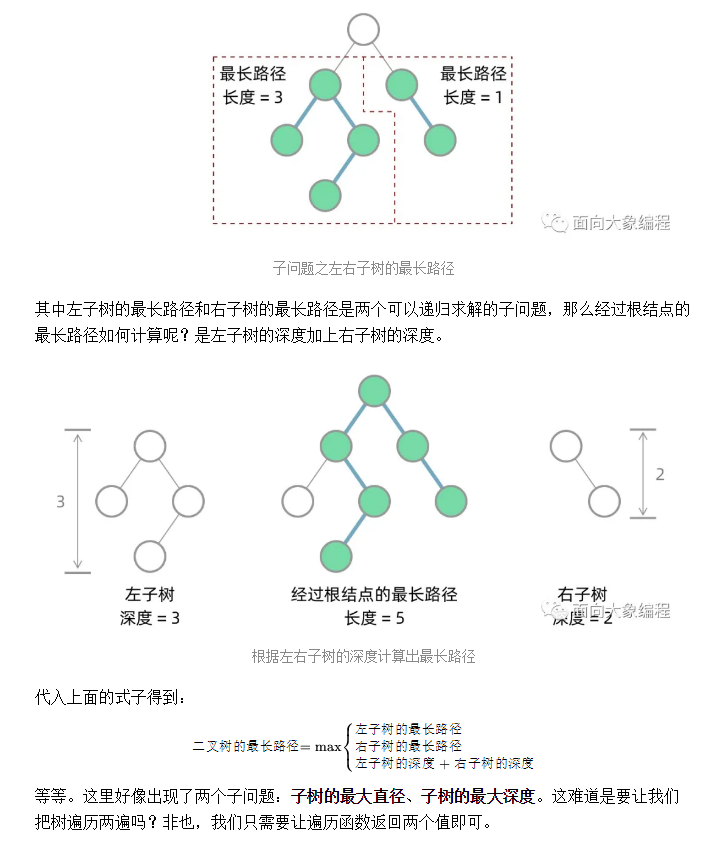
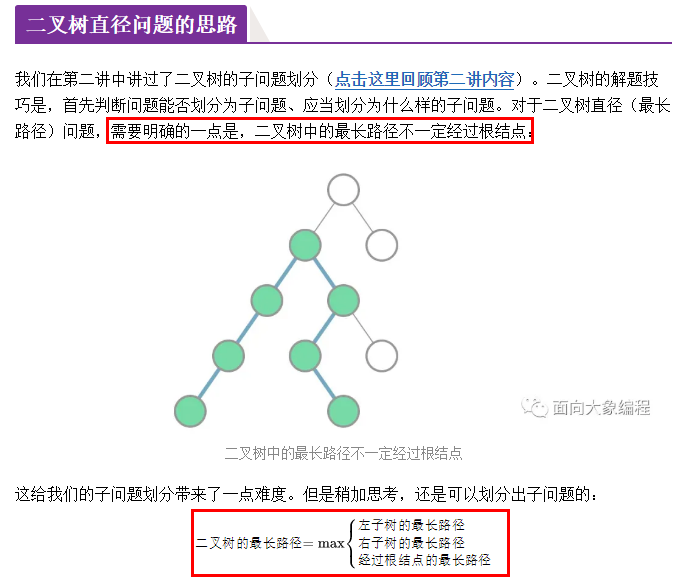
## 507\_完美数

一个数N与除了它自身之外的的所有正因子之和相等称为完美数。

那就必须先求出其所有因子，我们可知，只需要求到sqrt（N）之前的因子pres，就能求出所有因子，因为sqrt（N）之后的因子post可以通过N // pre[i]求出。

但这里要注意两点：1是所有的数的因子，所以sum的初始值为1，还有就是sqrt（N）这个数，只能被加一次，所以要单独处理。

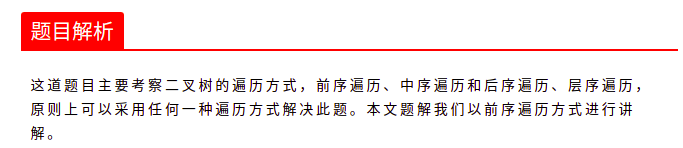
## 543\_二叉树的直径

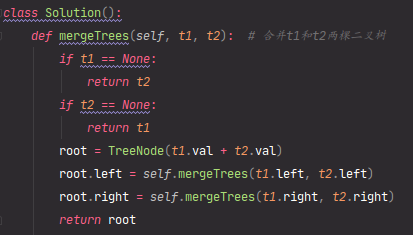


## 563\_二叉树的坡度



## 617\_合并二叉树





## 876\_链表的中间节点

定义快慢指针，快指针的速度是慢指针的两倍。

注意防止fast.next空指针异常，首先要判断fast不为nil

# 中等

## 11\_盛最多水的容器

很明显这道题就是要用到双指针的思想，也就是求最大面积，那么就是尽可能让底足够长，要么让高足够高，定义一个全局变量，实现O(n)的时间复杂度。

而且一定是先移动较短的那个柱子，假如移动的是较长那端的柱子，那么水的高度不会增加，而且底却在一直减小，显然不合理。

## 15\_三数之和

首先拿到这道题，如果用暴力解法的话，肯定是O(n3)， 所以呢， 我们需要降低时间复杂度，可以先将数组排好序，时间复杂度为O(n\*logn), 然后固定一个值，求另两数之和，这样就可以利用双指针缩小搜索空间了，固定n次，双指针用n次，故时间复杂度为

O（nlongn） + O（n2） = O（n2).

特别注意的时，固定一个数后，left指针从固定这个数的位置后一个位置起，能够有效减少搜索空间。还有就是，因为存在这很多重复的元素，答案可能需要去重，这里因为已经排序，去重的方法就是，当相等时，左右指针判断相邻的元素是否相等，直到搜索到不相等的元素位置，包括固定元素也是一样。

## 19\_删除链表中的倒数第N个节点

利用快慢指针，快指针先走N个节点，慢指针后走，那么快指针到末尾后，慢指针所指位置即是倒数第N个节点。

这里值得注意的一点就是，如果快指针走了N个节点为空了，就表明第一个节点就是要删除的节点。

## 34\_在排序数组中查找元素的第一个位置和最后一个位置

题目要求时间复杂度为O(logN), 很明显要用到二分查找的思想，

If target > nums[mid]:

Left = mid + 1；

Else：

Right = mid

这样会找到左边那个，条件反之，会找到右边那个。

因为只有当target大于mid， left才会动，当left指向了第一个要查找的元素，那么它永远不会移动到第二个查找元素上，因为之后的target会一直小于等于nums[mid].

也可以这么理解，只有相等的哪一方才能滑动到另外一方。

这里有一个小bug，因为整除元素3 // 2 = 1, 所以二分查找最好是滑右边，这样就会left == right， 滑左边的话，当left和mid相等是，将始终保持left + 1 == right。就会造成死循环。

所以这个时候循环跳出条件应该是left + 1 >= right.

但是无论是滑那边，最终那个元素始终都会是left指向的位置。

但是其实可以转为一样的，比如 3 // 2 = 1 + 1 =2, 那这样的话， 左滑右滑都是一个样子的啦。条件也都是left < right。但是正常来说还是右滑比较好，即右边相等。

## 36\_有效的数独

对每一行，每一列，每一个盒子都建立一张hash表，也就是27张hash表，存入三个数组row、col、box，row[i]表示第一行的元素的hash表，col[j]表示第一列的元素的hash表，box[(i // 3) \* 3 + (j // 3)]表示第几个盒子的hash表。

建立一个二层循环，依次存入hash表，hash表中存在即return False。

## 46\_全排列

关键思想时，回溯数组，k左边为已选集合，k右边为候选集合，当k到达末尾，候选集合为空时，实现一次全排列，记录所有情况即可。

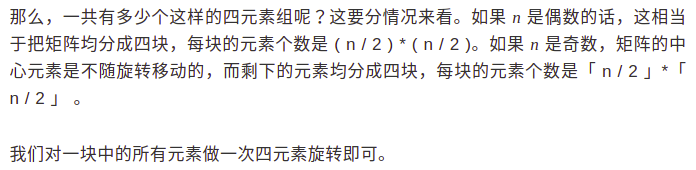
变种：1、从n个元素中选m个数全排列，那么就是已选集合达到m个时，记录一下即可。

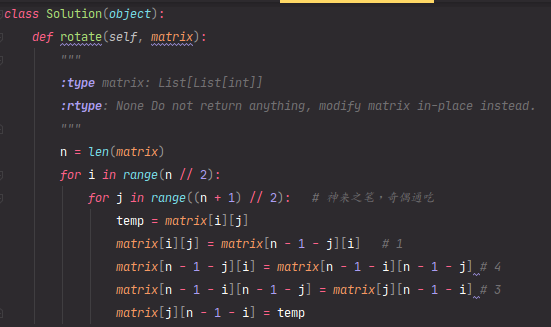
2、从n个元素中选m个数，即组合，注意两点，数组要排序，不交换。见77\_组合题。

## 48\_旋转图像









## 54\_螺旋矩阵

这种矩阵走路问题，三个必然用到的条件：

1. inbound函数
2. Dx = [0,1-1,0], dy = [1, 0, -1, 0]
3. Visit矩阵

方向变化：

1. 方向向右，到头，向下转
2. 方向向下，到头，向左转
3. 方向向左，到头，向上转
4. 方向向上，到头，向右转

注意：

res.append(matrix[nextX][nextY])  
 visit[nextX][nextY] = 1

这两者是同在的。

## 127\_单词接龙

将其转化为邻接表，然后利用图的BFS，因为是BFS，所以当遇到最终节点时，就会使最短路径。但是会超出时间限制，希望以后能找到解决方法

## 130\_被围绕的区域

无法直接找到被围绕的区域有哪些，那么就找没被围绕的区域有哪些，然后将没被围绕的区域的O改为T，然后再将所以非T区域改为X，然后将T区域改为O，就实现了等价操作。

这里采用queue求解，特别适合不规则回溯方法。

## 152\_最大乘积子数组

1. 数组中有负数，所以maxProduct = -2e32
2. 相比于最大连续和子数组来说，这里因为是乘积，所以要维持两个数，一个最大乘积，一个最小乘积，前者一定为正，后者一定为负，当遇到负数时，交换两者。这是精髓，一定要记住。

## 162\_寻找峰值

这道题目其实不难，就是要降低时间复杂度，如果你遇到一个大部分升序序列，知道最后第二个元素才是峰值，那么时间复杂度接近O（N），所以这种题目就是告诉你变相使用二分法。

IF Num[mid -1] < num[mid] > num[mid + 1]

Return mid

Else IF Num[mid -1] < num[mid] < num[mid + 1]

Start = mid

Else

End = mid

## 213\_打家劫舍2

在打家劫舍的基础上分为两种情况：

1、# 第一间房屋必偷， 那么就要从第三间房屋开始偷起， 偷到第N - 1间房屋为止

2、# 第一间不偷，偷到最后一间房屋为止

重用代码。

## 240\_搜索二维矩阵2

从左下角或者从右上角开始，以右上角A[x,y]为例:

If target > A[x, y]:

y--

Elif target < A[x, y]:

x++

Else:

Return [x,y]

## 337\_打家劫舍3

递归函数表示# 从root节点进入 偷root节点和不偷root节点返回的最大金额

注意：兄弟节点不算相邻

## 406\_根据身高重建队列

很明显这道题目并不是真的按照身高进行排序，而是根据身高和前面比它的身高高的人进行排序，所以你不能先将矮个子排序，因为题目中的第二个参数是，前面比它高的人的人数，如果你从最高的开始排序，那么你就能够完美知道哪些人比你高，你就可以站到指定的位置。

换言之，这是一道你只知道比你高的人有多少个的题目。

本质上就是插入排序。但这个插入排序利用了新的空间。

## 417\_太平洋和大西洋水流问题

跟130题被围绕的区域一样，不可能从中间入手，只能从周围入手。

找到从四周流往中心的路线，最终四条路线重叠的部分就是所求解。

也就是从低处往高处爬。

很明显，这个问题存在这大量的dfs叠加的问题，需要抽取出来。然后遍历即可。

结果超出时间限制，case通过率111/113

采用queue来做这种题目，visit的赋值在取出的那一下赋值即可，方便简单。

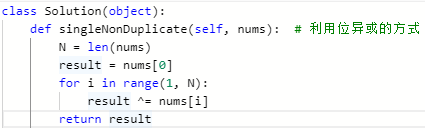
## 445\_两数相加2

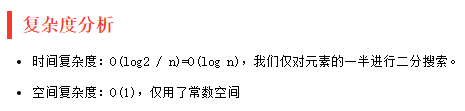
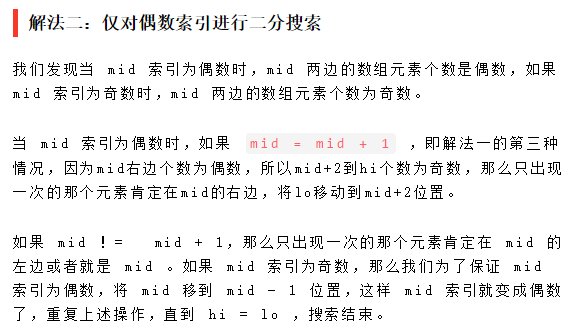
因为两者的位数可能不同，所以变得有点难搞，所以必须逆序，那么就可以借助栈来实现逆序。加入长度不一致之后的话，相加的那一部分的因子为0即可。所以循环跳出的条件应该是stack1 or stack2 or carry != 0， 而且应该采用从前往后重新构造链表的方式，而不是从后往前重新构造链表。

这种进位的思想值得学习。永远求的都是进位本身，而不是进位的下一位。

## 540\_有序数组中的单一元素

解法一：





二分了N次，直到只剩下一个元素，就是我们要找的那个数:

2 ^ N = n ----> N = logn

## 718\_最长重复子数组

# dp[i][j] 表示A以第i个字符结尾和B前j个字符结尾的最长重复子数组长度



空间优化：无法空间优化，因为递推关系式既设计上一行也涉及上一列

## 978\_最长湍流子数组

1. dp从1开始计数，因为刚开始和当两数相等时，从新开始寻找湍流子数组时，本身计为1，所以从第二个开始循环，相等之后，也相当与从相等之后的下一个数开始循环
2. 设置flag，flag为1表示前面为上升趋势，为0为下降趋势。

定义成一个子问题的话，以A[k]结尾的最长湍流子数组，要分4种情况：

1. 最后一段是上升，前一段是下降，tmp += 1， flag = 1
2. 最后一段是下降，前一段是上升，tmp += 1， flag = 0
3. 最后一段是水平，前一段无论是啥， tmp = 1
4. 否则， tmp = 2

既然都要讨论前一段是上升还是下降，那么干脆定义成两个子问题，一是以A[k]结尾的最后一段时上升的最长湍流子数组f[k]， 二是以A[k]结尾的最后一段是下降的最长湍流子数组g[k].

If A[k] > A[k - 1]:

f[k] = g[k] + 1

G[k] = 1

Elif A[k] < A[k-1]:

G[k] = f[k] + 1

F[k] = 1

Else:

F[k] = 1

G[k] = 1

## 1372\_二叉树中的最长交错路径

题目不难，难得是如何只进行一次深度遍历，解决方法是：

当该往左节点时，那么就不会往右节点了，那么这个时候右节点就可以重新计算最长交错路径，达到一次遍历，两次运用的目的。

## 1143\_最长公共子序列

参考718题

# 困难

## [面试题 08.12. 八皇后](https://leetcode-cn.com/problems/eight-queens-lcci/)

在八皇后的基础上，只要将p用相应的字符串表达出来即可。

## LCP 14\_切分数组

用gcd（a, b）> 1来判断太过耗时间，只需求出b的所有质数u[]，看a%u[i] == 0是否满足，满足就表明最大公约数大于1，这是题目设计好了的，就是要你用质数筛。

## 23\_合并k个排序链表

结合归并排序，转化为合并两个排序链表的算法。

## 42\_接雨水

双指针问题，left，right分别记录当前柱子的高度，根据木桶理论，与左右较矮柱子的高度差即为当前位置能够蓄水的体积，所以从较矮柱子那边开始计算雨水的体积，来回切换，直到左指针大于右指针。

如果用单指针，每次都要计算两侧的leftMax和rightMax，那么时间复杂度将是O（n2）。

## 72\_编辑距离

看一个例子就懂：

horse --> ros

插入s: horse --> ro dp[i][j-1]

删除e: hors --> ros dp[i-1][j]

替换e: hors --> ro dp[i-1][j-1]

注：1、当比较的字母相等时，不需要替换操作

2、当word1长度为0时，要插入word2长度，反之删除word1长度

## 124\_二叉树的最大路径和



应该划分为：（倒数第三句）

1. root.val + 左子树的最大路径和（因为右子树为负）
2. root.val + 右子树的最大路径和（因为左子树为负）
3. Root.val + 左右最大路径和（左右子树都非负）

## [1373\_二叉搜索子树的最大键值和](https://leetcode-cn.com/problems/maximum-sum-bst-in-binary-tree/)

我的思路：取一个全局变量，从上往下记录最大键值和self.sum，当不满足二叉树的性质时，self.sum清0，从不满足二叉树性质的那个节点所在层从新计算。

阶梯思路：上面的思路错误，我的想法是，从上往下，这样是大错特错的，因为加入底下不是二叉搜索树，那上面更不会是，所以应该采用的是自底向上，这里的思路很巧，返回三个值，以root为根节点的二叉搜索树的路径和，该二叉搜索树的最小值和最大值，这时应熟练处理，当root为None和当root为根节点的二叉树不是二叉搜索树时的情况，将题目条件运用的非常巧妙。



# 要钱的letcode和不存在的letcode

## 迷宫1\2\3\4

**490、一个二维数组表示一个迷宫，0表示为空，1表示为路障，给定任一个位置，人是否可以走出迷宫，迷宫出口默认在二维数组右下角：**

这种本身带有路障的矩形矩阵问题可以不设置visit数组。

Queue其实也是回溯，只是因为queue因为先进先出的特性自带回溯。

Queue的缺点就是无法求出具体路线。

1. **相比于1，要找出走出迷宫的最短距离。**

增设一个让循环进入continue的条件即可：

即：给经过的网格标数，当下一个网格的数不大于当前网格的数， 表明你没有资格走到那个网格。

如何设置数呢，其实很简单：maze[nextX][nextY] = maze[x][y] + 1

1. **给定三个值初始化的m行n列的二维网格，-1表示墙壁或障碍物，0表示门，INF表示空房间，使用值2的31次方 - 1 = 2147483647表示INF，假设到门的距离小于2147483647。在代表每个空房间的网络中填入与门最近的距离，如果不可能到达门口，则应填入INF。**

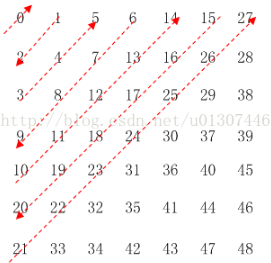
**4、给定一个m行n列的二维字符数组表示迷宫。它有四种房间，‘S’代表从哪开始（只有一个起点），‘E’代表迷宫的出口（当抵达出口时，将离开迷宫，该题目会有多个出口），‘\*’代表这个房间可以经过，‘#’代表一堵墙，不能经过墙。每次可以上下左右移动到达一个房间，花费一分钟时间，但是不能到达墙。本例将得出离开这个迷宫所需的最少时间，如果不能离开，则返回-1.**

## 不同岛屿的个数1\2

给定一个由0和1组成的非空二维网格代表一个岛屿，一个岛屿是指四个方向（包括恒向和纵向）都相连的一组1， 其中1表示陆地，0表示水域。假设网格的四个边缘都被水包围了，找出所有不同岛屿的个数，如果一个岛屿可以被转换（不考虑旋转和翻折）成另一个岛屿，则认为两个岛屿是相同的。

## 矩阵元素zigzag返回

给定一个m行、n列的矩阵，以ZigZag的顺序返回矩阵中所有元素，如图所示：



## 4、能否到达终点\

给定一个m行n列的矩阵表示地图，1代表空地，0代表障碍物，9代表终点，判断从（0,0）开始能否到达终点，若能到达终点返回True，否则返回False

## 5、建立邮局

给出一个二维网格，每一格可以代表墙（2）、房子（1）和空（0），在网格中找到一个空的位置建立邮局，使得所有房子到邮局的距离和是最小的。返回所有房子到邮局的最小距离和，如果没有地方建立邮局，则返回-1.

# 数据结构

## 图：

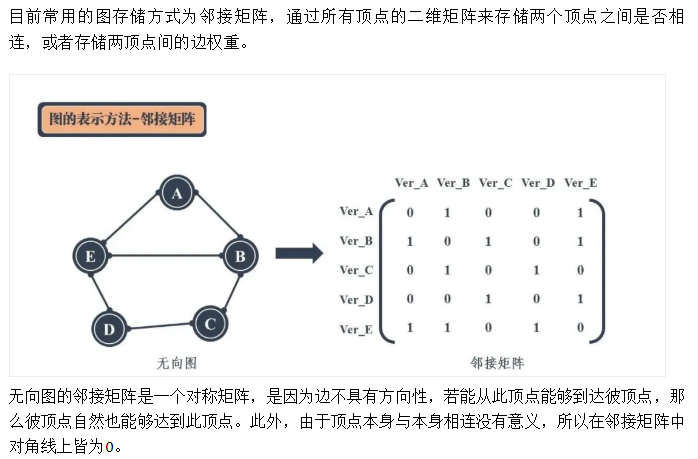
图分两种：

有向图（节点有入度和出度）、无环图（可以间接认为只有出度）

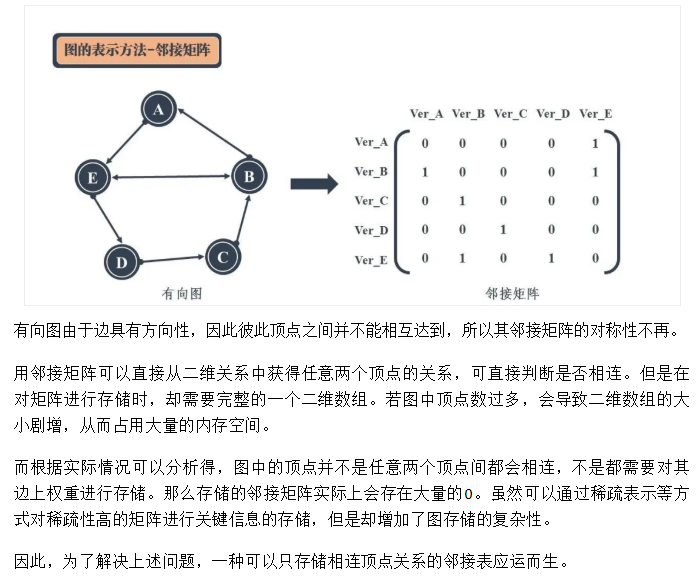
图有两种存储方式：（我的代码实现的是简单的邻接表）

A、邻接矩阵：

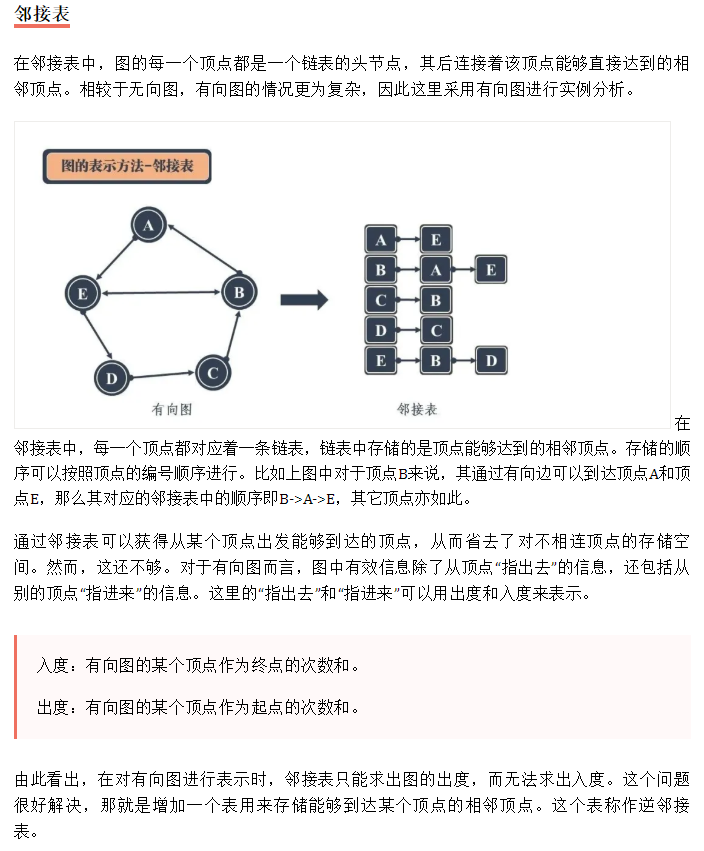
无向图：

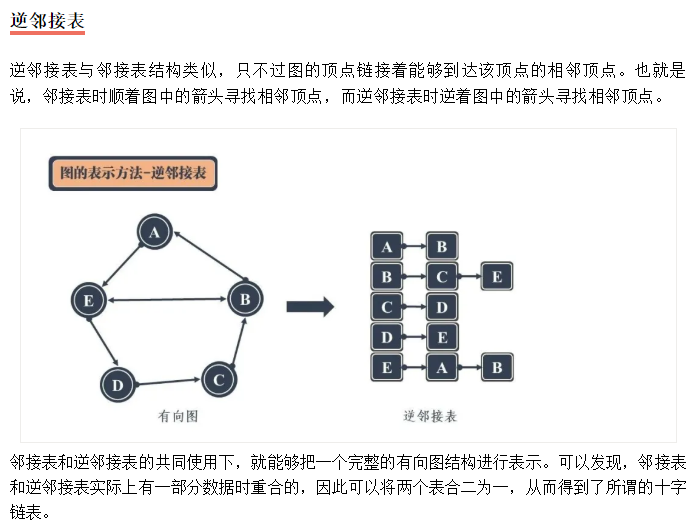


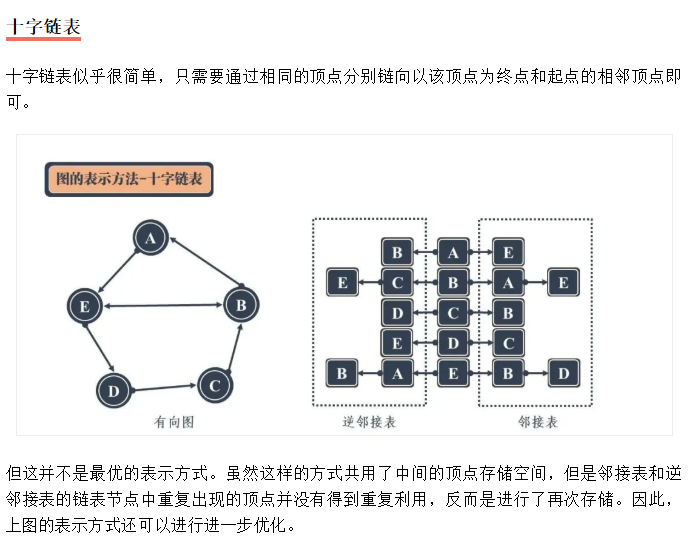
有向图：



1. 邻接表







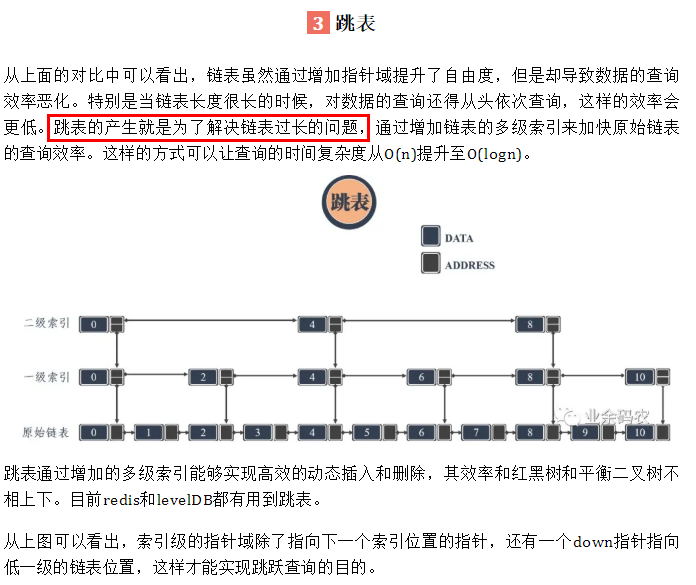


## 数组和链表



数组因为连续的存储空间所有可以实现随机访问，但这样的话增删元素会移动被修改元素之后的元素。

## 跳表



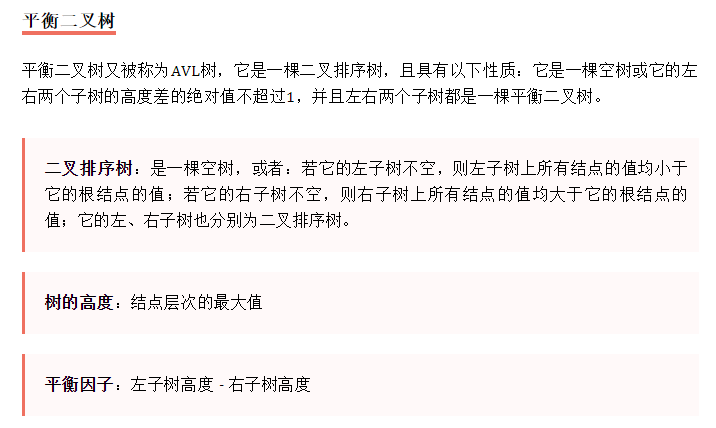
## 红黑树vs AVL

### 二者的定义

都是二叉排序树：

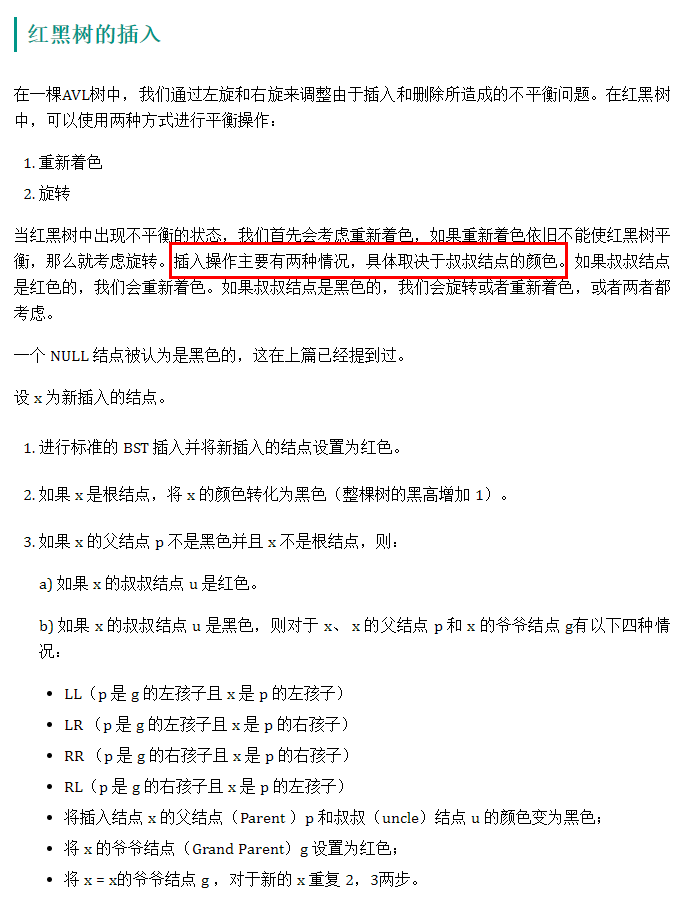
红黑树五个特性要记牢：

1. 根节点为黑
2. 节点要么为黑、要么为红
3. 不存在两个连续的红节点（红节点的子节点为黑）
4. 从任一个节点到叶子节点出发的黑高相等
5. 叶子结点为黑





### 红黑树的插入



总结：

1. 插入的节点的都是红色节点，但插入的节点作为根节点，那么将其重新着色为黑色
2. 若父节点是黑色，则不用重新着色和旋转
3. 若父节点是红色：
4. 叔叔节点是红色，那么只需将叔叔和父节点重新着色为黑色，将爷爷节点置为红色，那么视爷爷节点为新插入的节点，重复2、3步。
5. 叔叔节点是黑色，结合爷爷节点和父节点的位置关系分为：

LL（右旋爷爷节点，交换新的爷爷节点和右孩子的颜色）

LR（先左旋父节点，转化为LL）

RR（左旋爷爷节点，交换新的爷爷节点和左孩子的颜色）

RL（右旋父节点，转化为LL）

综上：其实叔叔节点是黑色更好一点，因为不用重复2、3步。

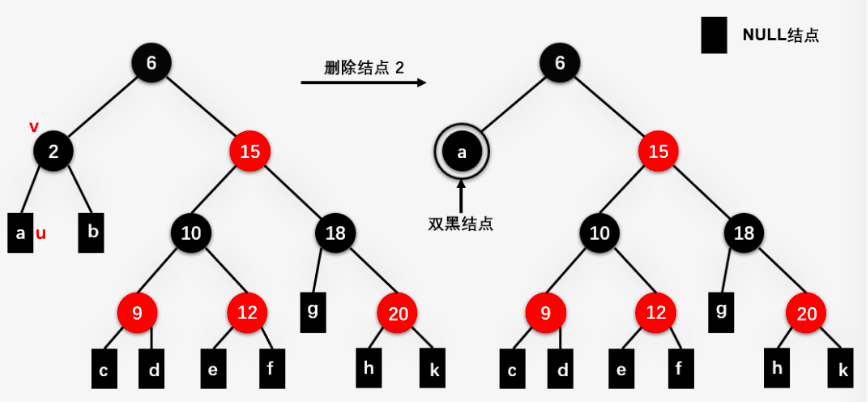
### 红黑树的删除操作

通过检查兄弟节点的颜色来决定恰当的平衡操作。

插入是为了避免红黑树中出现两个连续的红色节点，而删除操作是为了避免删除黑色节点导致黑高降低。

当删除的节点v是黑色节点，且其被黑色子节点替换时，其子节点就被标记为双黑：

删除操作的最主要任务就是将双黑节点转化为我们的普通黑色节点。

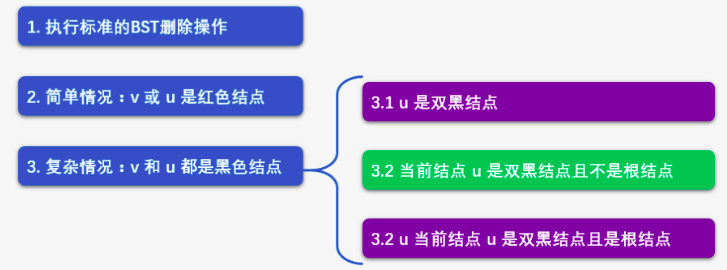
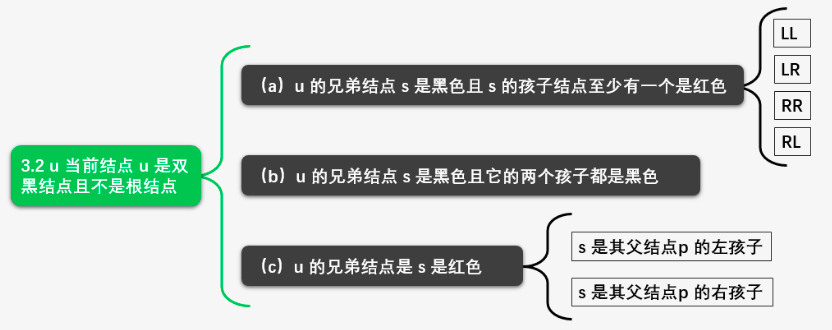


图中，为了方便，双黑节点直接变为了叶子节点，试想一下不是叶子节点的情况，你就能想明白了。

既定删除节点为v，u是用来替换v的孩子节点：

删除操作总纲：

1. 执行标准的BST的删除操作。
2. 简单情况，u或者v是红色
3. 复杂情况：u和v都是黑色节点

1. 标准的BST删除操作：

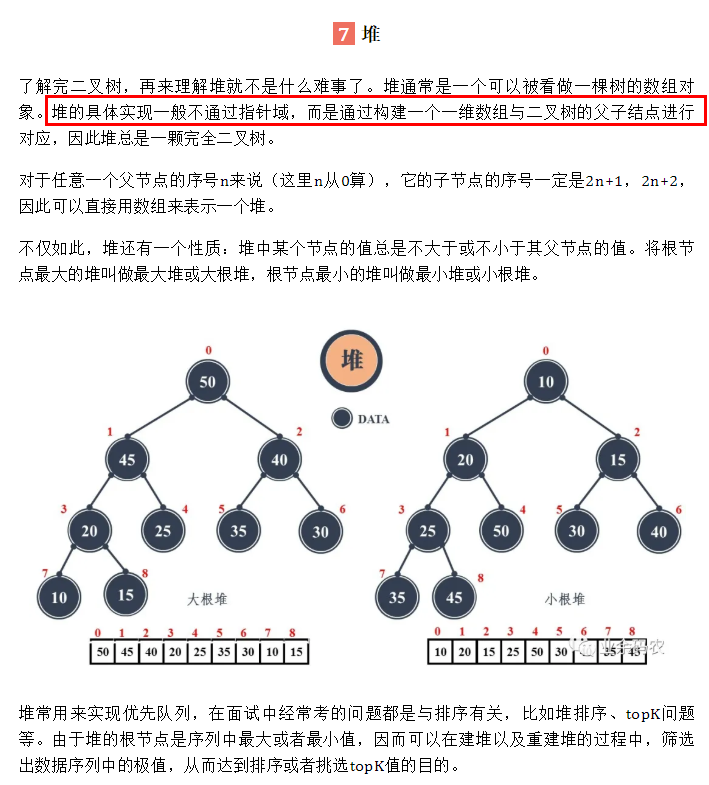
值得一提的是，最终都会以删除一个叶子节点或者只有一个孩子的节点而结束。

1. 简单情况：u和v有一个是红色节点

将u替换v，并置黑

1. 复杂情况：u和v都是黑色节点

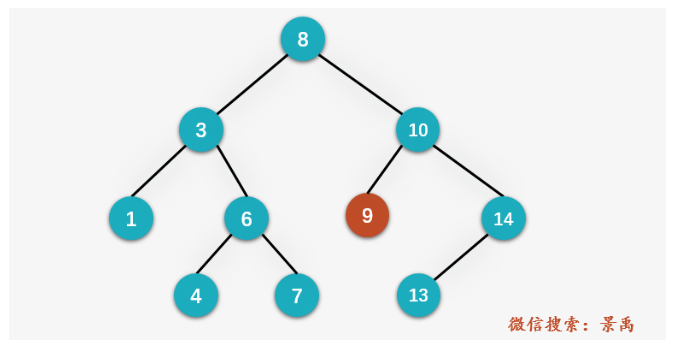
## 5、堆

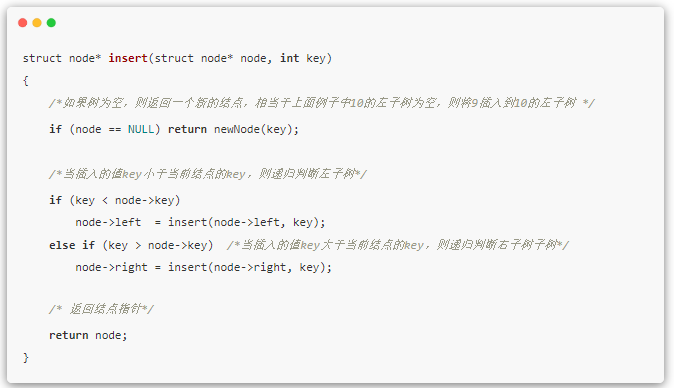


## 树

### 二叉排序树的插入

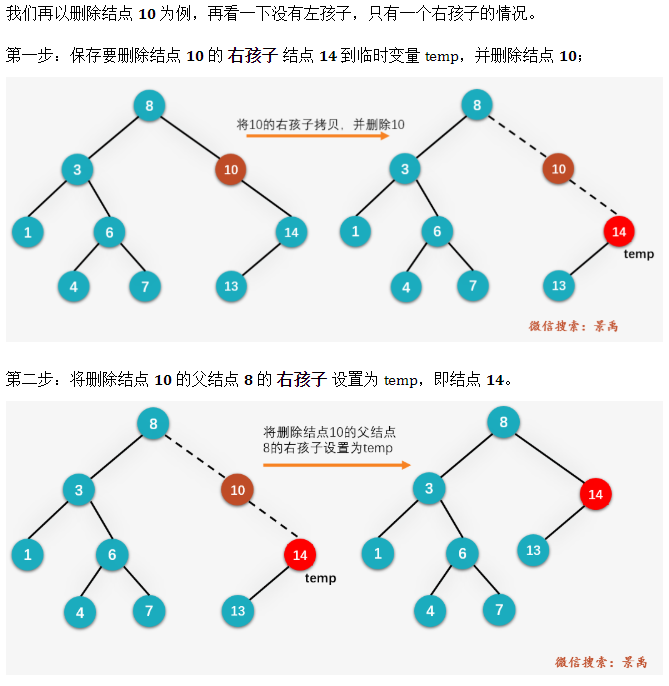
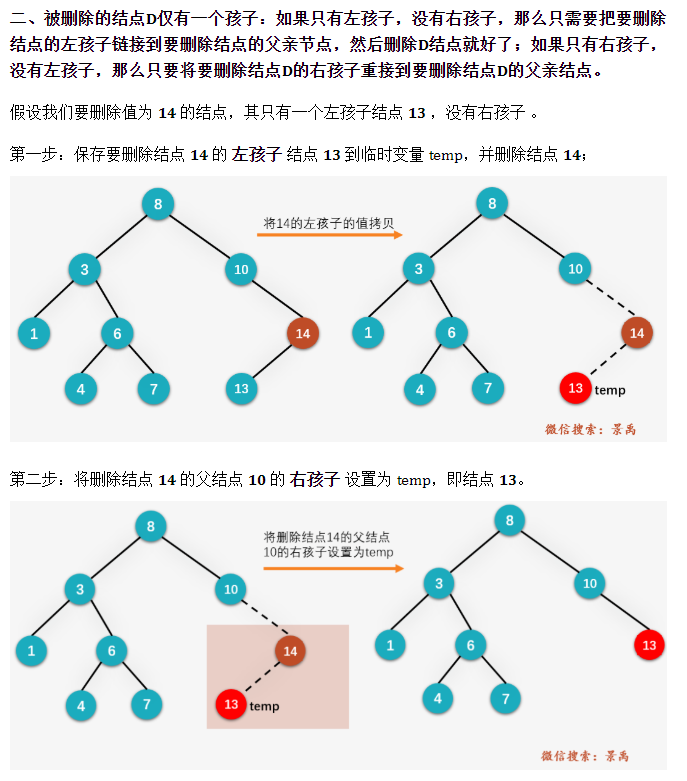
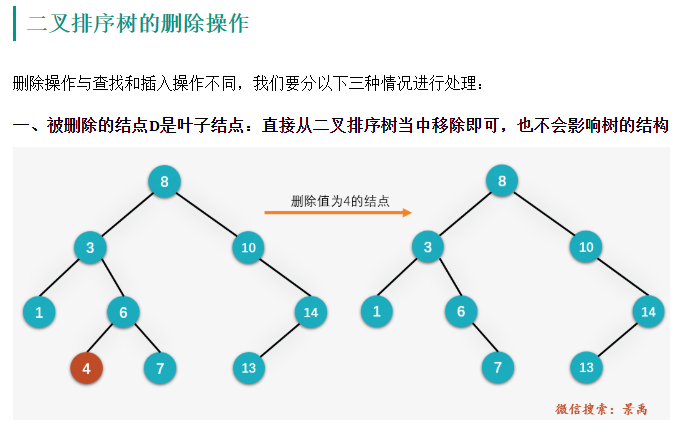
二叉树的插入其实和查找没什么区别，因为给定一个节点，进行插入时，你细品一下，一定插入的位置会是叶子节点的。





二叉树的插入其实就是构建一棵二叉树的过程，返回的应该是新的二叉树的节点。

### 二叉排序树的删除



1. **当左右孩子都存在时，根据中序遍历，那么可以将该节点的左子树的最大值或者右子树的最小值的节点替换该节点，然后删除左子树的最大值或右子树的最小值，而这两者其实都是叶子节点，所以完美解决。**

**注：1、当然你需要用到二分查找的方式去查找最小值和最大值。**

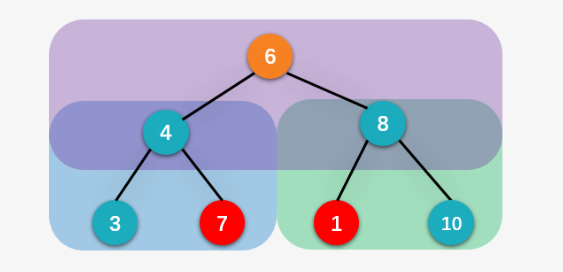
1. **删除其实也是重新构建二叉排序树的过程**

### 判断是否是二叉排序树

记住，二叉排序树的定义是，树及子树的所有左边节点小于根节点，所有右节点大于根节点。

所以下列判断方法是错的：

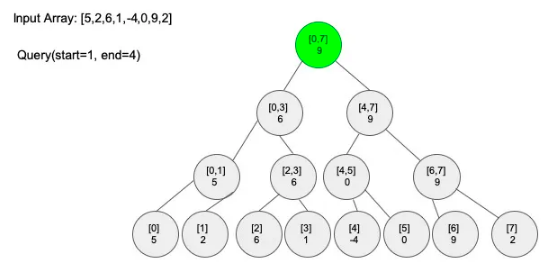




这样一棵树其实不是二叉排序树，但是返回True

为了只遍历一遍树，那么就要建立辅助函数，传入的参数不仅仅是一个根节点，而应该包含最大和最小节点。

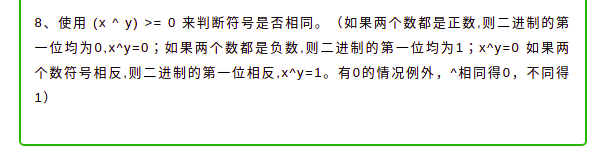
## 线段树



存的是区间和该树的最大值

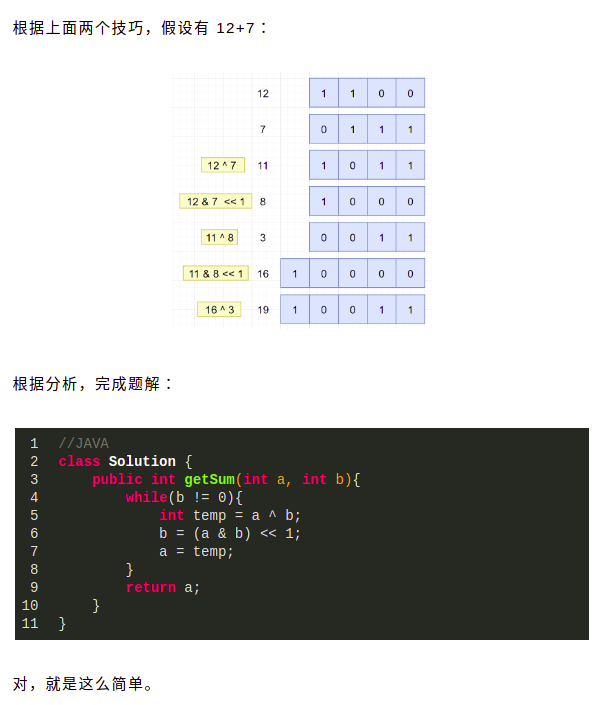
## 位





具体实现看代码： 数据结构--位

### 两数之和

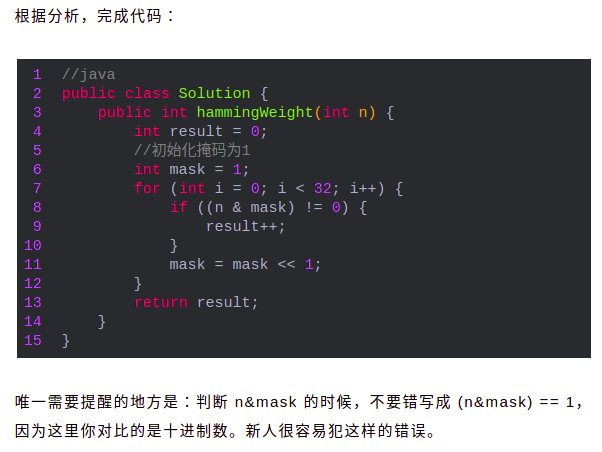


### 是否是2的幂



### 汉明距离



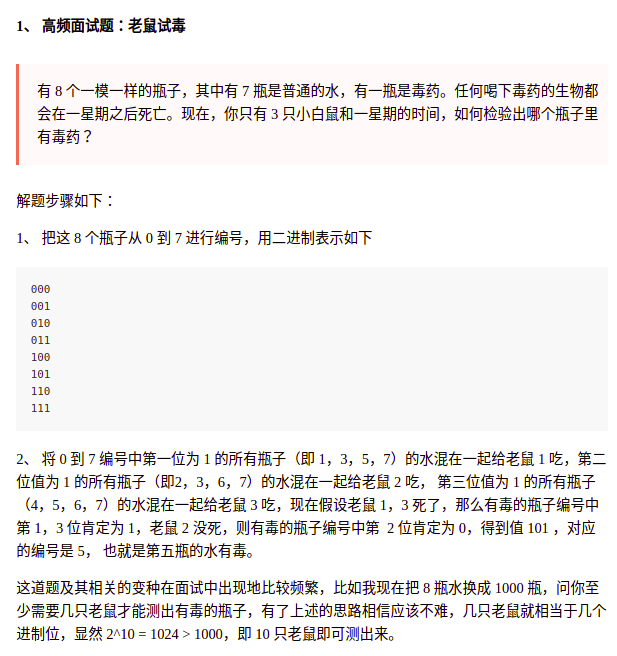


而且 *# 最好将1向左移动，而不是将数向右移动，因为算术右移的话，左侧补的是符号位*

### 只出现一次的数字（其余的数字出现也是奇数次， 假如为3）

外层循环是位数，如64位，内层循环为所有的数字，这样就能计算每一位有多少个1，然后将1的个数取3的余数，那么剩下的就是那个奇数位的数字的1，然后化为十进制。

### 老鼠试毒



### 八皇后

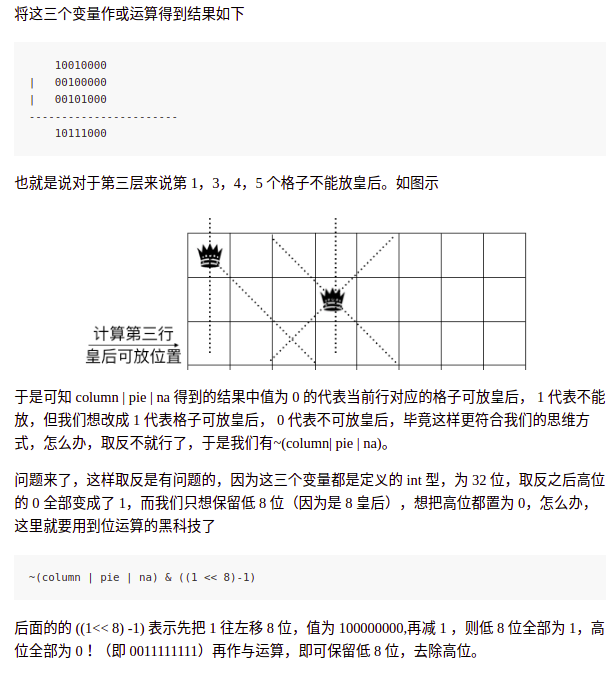
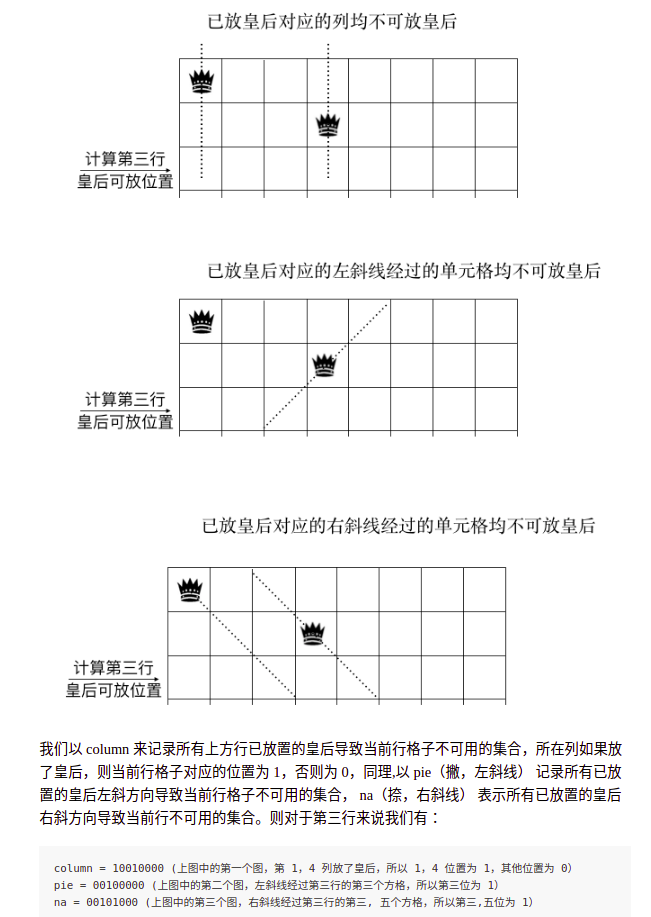
八皇后的问题就是要考察你对回溯法的利用，基础思路为：

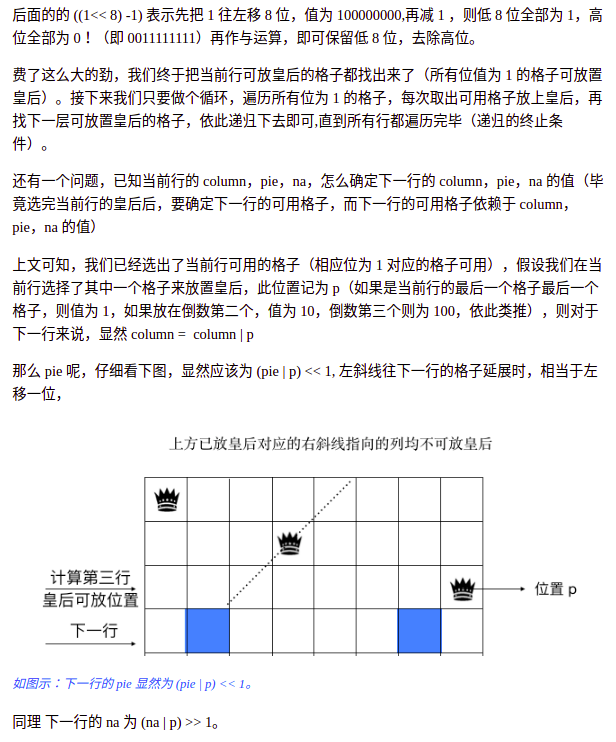
你可以在每一行放置一个皇后，放置的位置取决于上面的行放置的皇后的位置，如果没有摆放的位置后，那么就需要回溯，那么本题的难点在于，如何找到皇后能够拜访的位置，这个时候位运算就能解决这个烦恼。

Column表示当前行的某个位置的该列有无皇后

Pie表示当前行的某个位置的右斜线有无皇后

Na表示当前行的某个位置的左斜线有无皇后





将三者想或即得到了该行能够放置皇后的位置pos。

那么放置过后如何能够确认下一行能够放置皇后的位置呢？

假设你放置的皇后的位置是p，那么你将pos的p位置之外的位置置0即可，然后你用p|column得到新的column，用p|pie << 1得到新的pie，用p|na >> 1得到新的na。

相关伪代码如下：



这道题的难点在于如何设计回溯的条件。

因为bits &-bits能找到最右边为1的格子

且bits & bits - 1能将最右边为1的格子置0.

所以bits中的1代表的必须是能够放置皇后的位置p。

但是column\pie\na中的1代表的是放置了皇后的位置，也就是不能放皇后的位置。

所以p|column得到的1是不能放置皇后的位置，所以需要取反。

## 链表

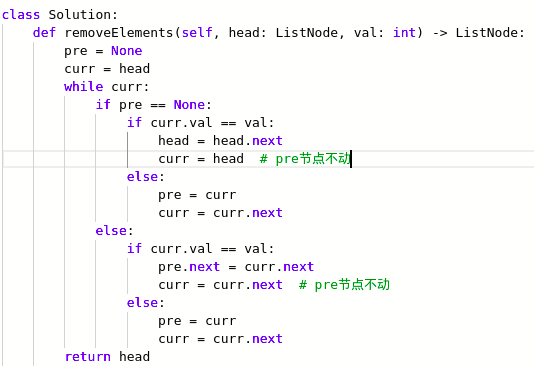
### 链表框架



链表框架最好加上newHead，newHead.next = head， 然后pre = newHead，这样就不用解决pre == Null的问题了。

### 移除链表元素

这个不能照搬链表框架，因为当删除元素时，pre会指向那个删除的元素



### 反转链表的k个元素：

这是一道运用原来的反转链表思路会超时的题目：还是运用我之前的解法解法比较好，因为要求时间复杂度为O（n）。运用改良过后的链表框架解题，值得注意的几点是：

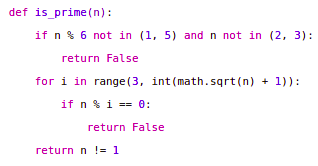
1. 要记录开始反转时的链表的节点
2. 当前节点cur通过pre.next来获得。
3. 当index == n时，直接返回newHead.next即可

## 经典数据结构

### 1、找出n以内的素数（素数筛）

**普通人的三部曲：**

1. 将偶数去掉
2. 只需判断n % math.sqrt(n) == 0
3. 数学定理：只有形如6n-1和6n+1的自然数可能是素数，n>=1.（所以2,3是例外）



最终代码如上：

其实是第2\3个条件的组合，第3个条件可以取代第一个条件。

**埃式筛法（素数筛）**

**就是用筛选出来的素数去过滤所有能够被它整除的数。**

**这些素数就像是筛子一样去过滤自然数，最后被筛剩下的自然数就是不能被前面素数整除的数，根据素数的定义，这些剩下的数也是素数。**

