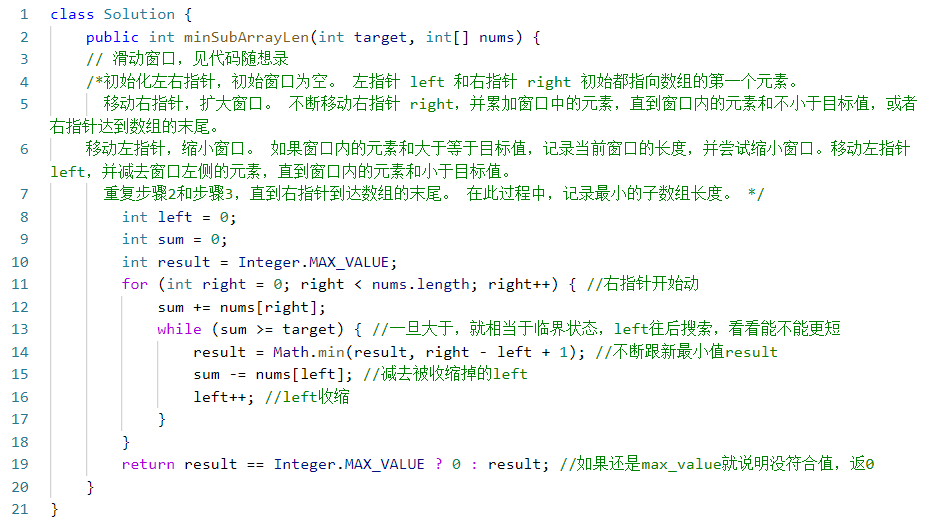
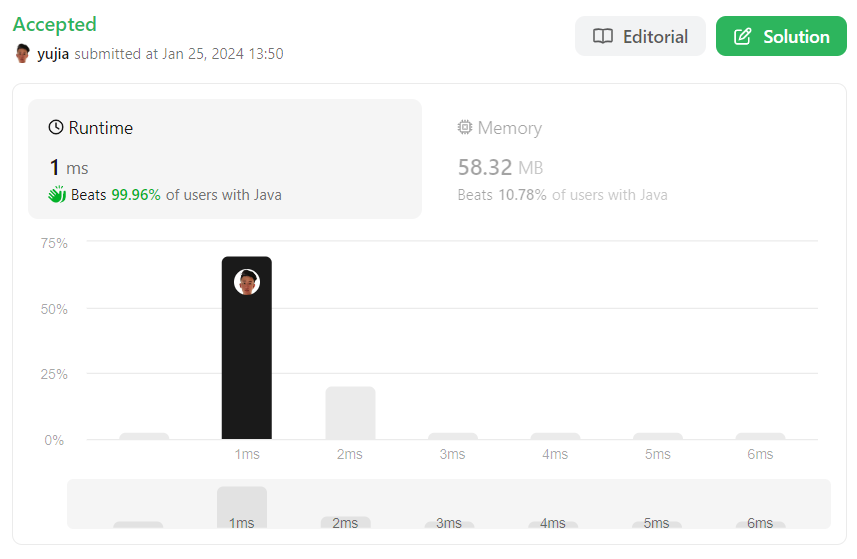
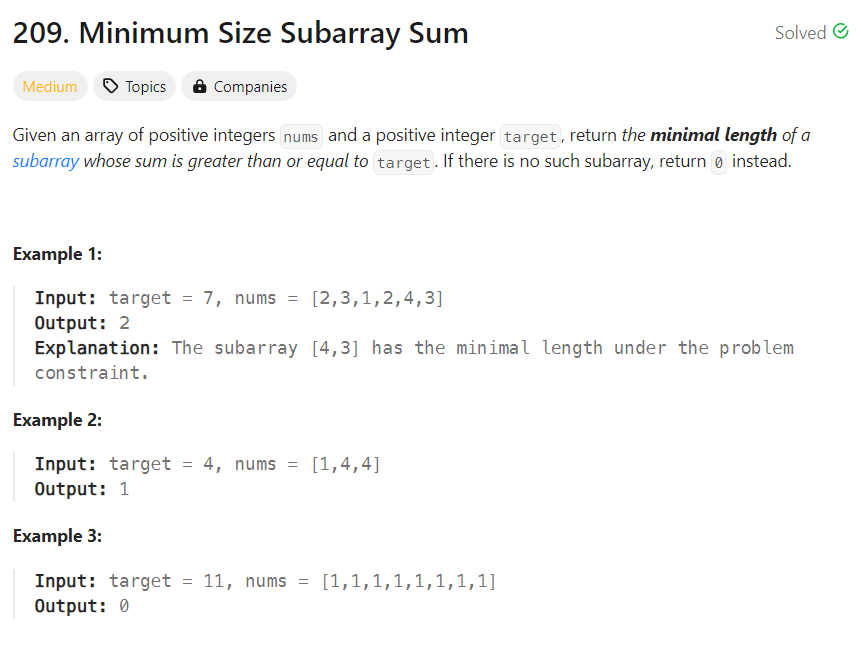


力扣笔记

Medium



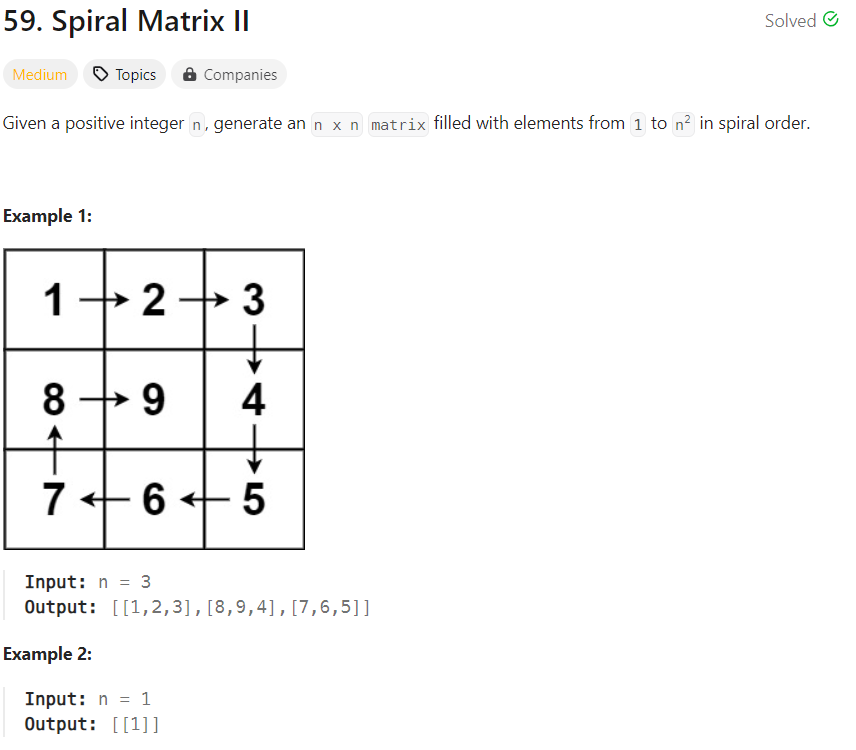
**ARRAY 数组**

下面方法是我自己做的，但是超时了

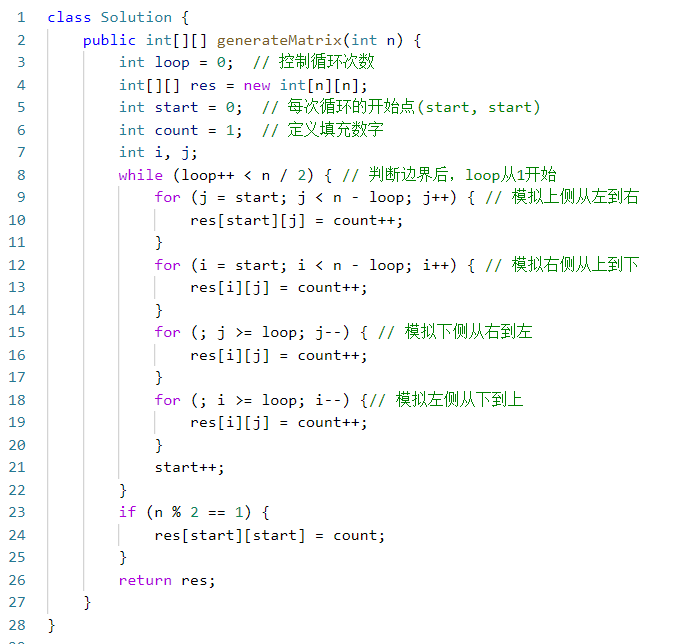
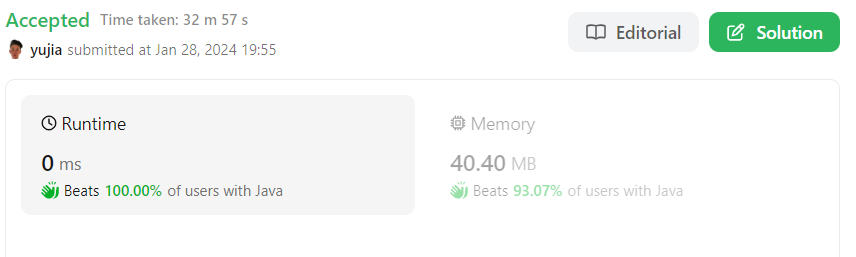


思路：新学到的**滑动窗口**，比我自己的双指针更高效，因为我的是双for循环，而滑动窗口是一个for循环里面嵌套一个while，算是2\*n，还是O(n).

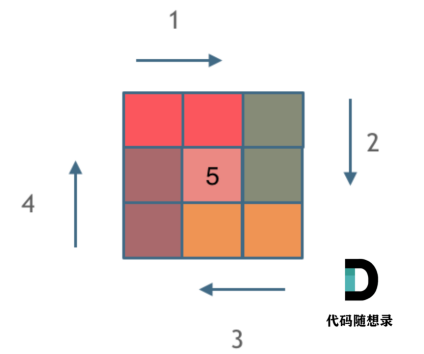
滑动窗口的核心就是新建一个左边的指针，然后在循环里设置指针作为右指针，当满足某个情况时，左++，进行窗口缩减，反正也是求跨距，最值之类的问题



**ARRAY数组**

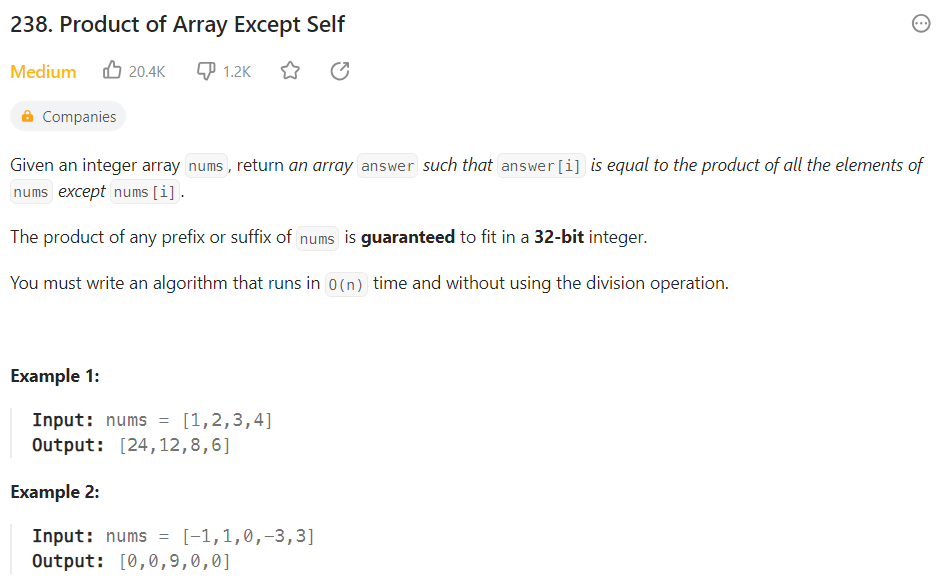


笔记：而求解本题依然是要坚持循环不变量原则。



由外向内一圈一圈这么画下去。这里每一种颜色，代表一条边，我们遍历的长度，可以看出每一个拐角处的处理规则，拐角处让给新的一条边来继续画。一层有4条边，多少层就多少个while次数

* 填充上行从左到右
* 填充右列从上到下
* 填充下行从右到左
* 填充左列从下到上

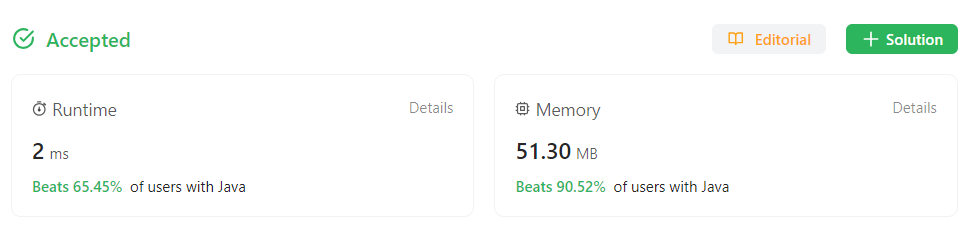


**ARRAY数组**

思路：

1. 以1为例，设置左右两个数组，来装当前元素左右测的乘积.
2. 比如当循环到2号位的1时，左侧乘积的2号位为8，右侧乘积为15
3. 最后把左右两个数组通过循环再乘一下得到结果
4. 主要难点在于左右的累乘是截至到当前位置的上一位，并且这个累乘的结果是放到左右数组的当前位置。对于循环标志i需要判断清晰。





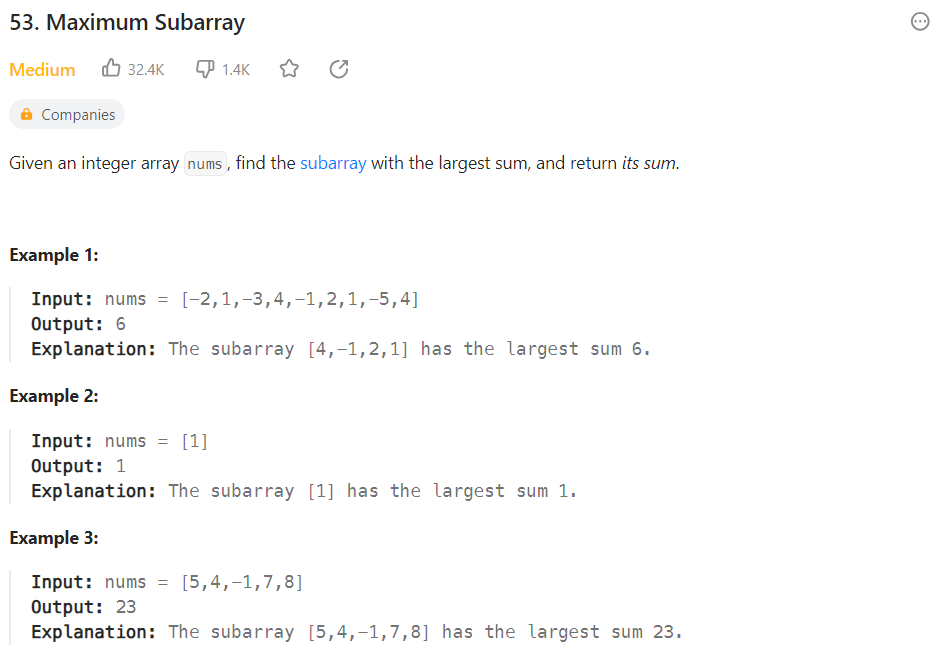
时间复杂度：O(n), n代表遍历数组nums[]里有n个元素

空间复杂度: O(1),在运算过程中，只有常数个变量空间产生，无循环产生n个内存空间

**方法一code：**



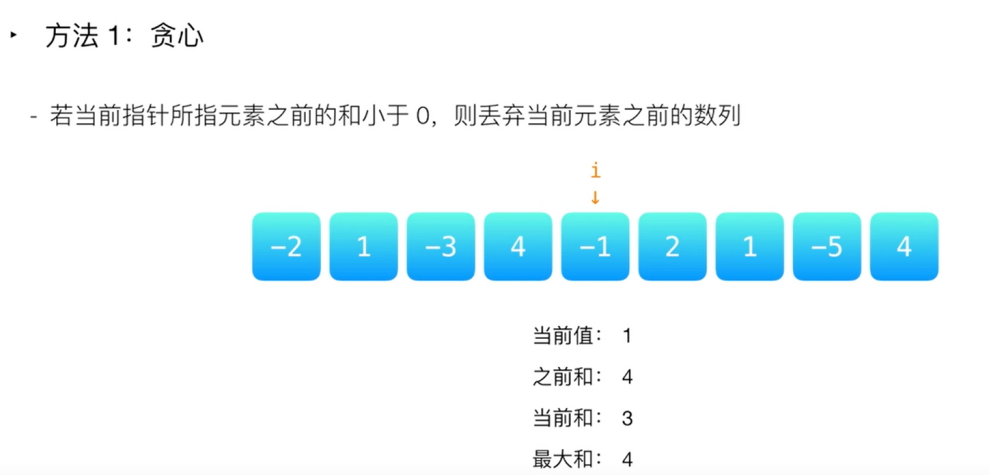
笔记：



**ARRAY 数组**

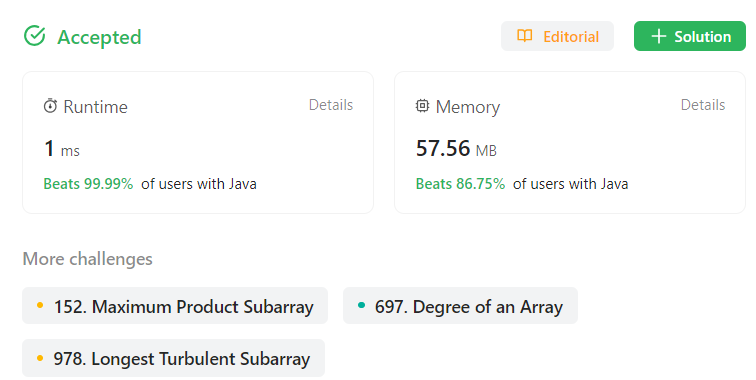
思路：

以贪心为例，主要思路是如果一旦当前循环到的元素要大于其与之前累计值的和。那么立即舍弃之前的累计值，其作为最新的累积值往后继续。

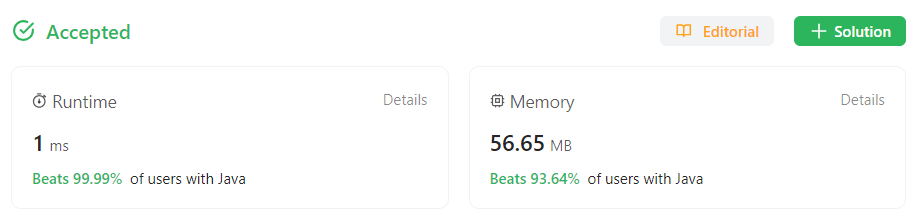




方法一：贪心算法



方法二：动态规划

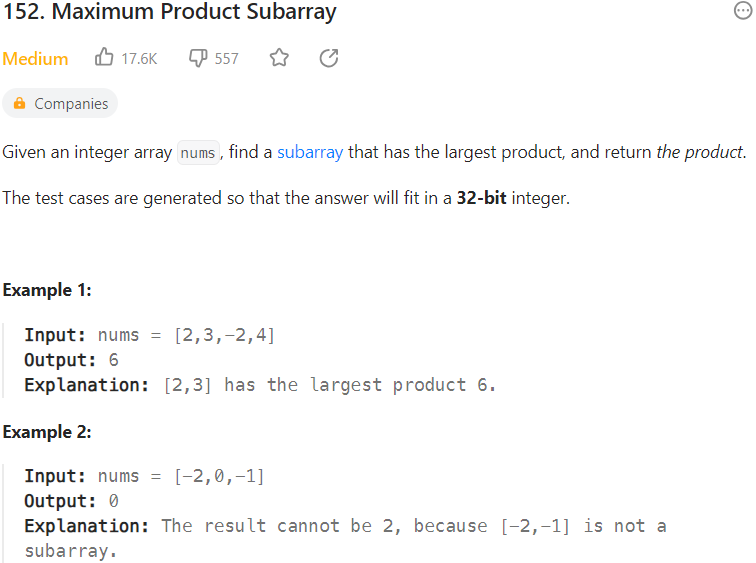


时间复杂度：O(n),其中n是输入数组nums[]的长度 [1,2,5,-4,9,0,2]

空间复制度：O(1),因为只用常数个变量空间，循环里面没有挨个创建新内存空间



笔记：如果问题涉及到最大值、最小值、累计值等，那么**贪心算法**可能是一种有效的思路。核心是通过不断更新一个变量（通常是一个最值变量），找到满足问题要求的最佳解。



**ARRAY 数组**

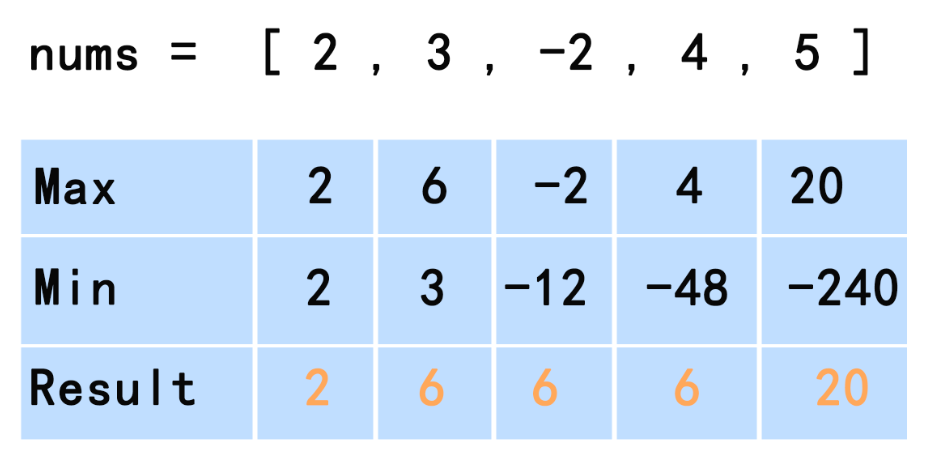
思路：

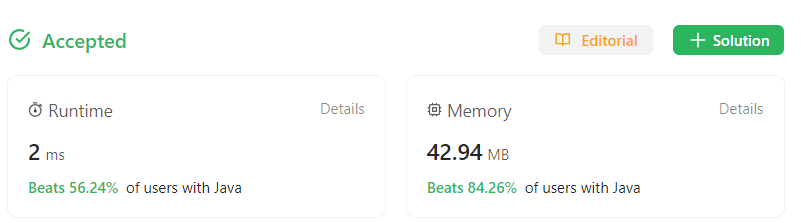
1.思路和力扣53很像，但是注意53中是累积，也就是当到某一个位置，加了还不如不加的时候，就可以舍去之前的，从当前位置开始。而这个存在负负得正，暂时的负数无法判断最终的走向。

2.因此考虑创建Max和Min以及最终result，目的是得到极值，因为最初的Max/Min都有可能因为nums[i]的正负性，得到完全扭转的值。

3.所以，max和min的赋值应该在两者与nums[i]的乘积中产生

4.除了上面2者比较，还要加入与nums[i]的比较。以-2为例，比较-12，-2，-6三个数，从中赋值给max。此时应该选择nums[i]代表的-2。



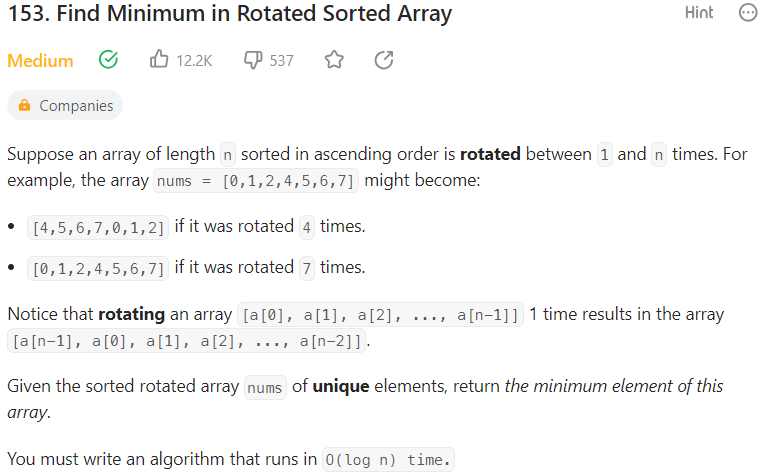


时间复杂度：O(n),其中n是输入数组nums[]的长度

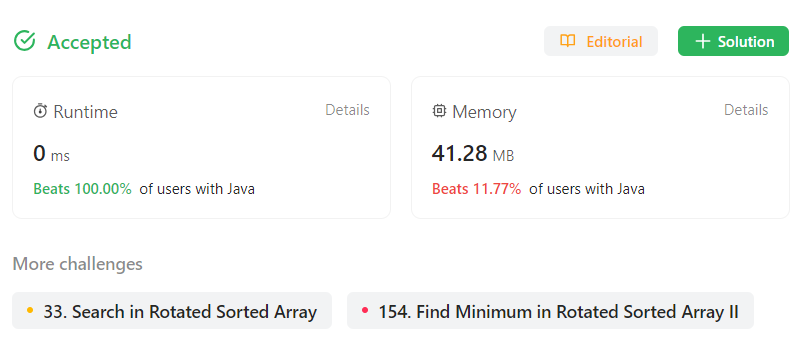
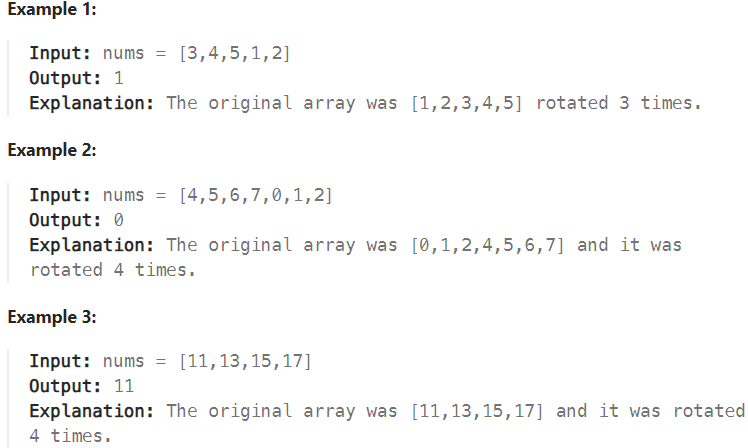
空间复制度：O(1),因为只用常数个变量空间，循环里面没有挨个创建新内存空间

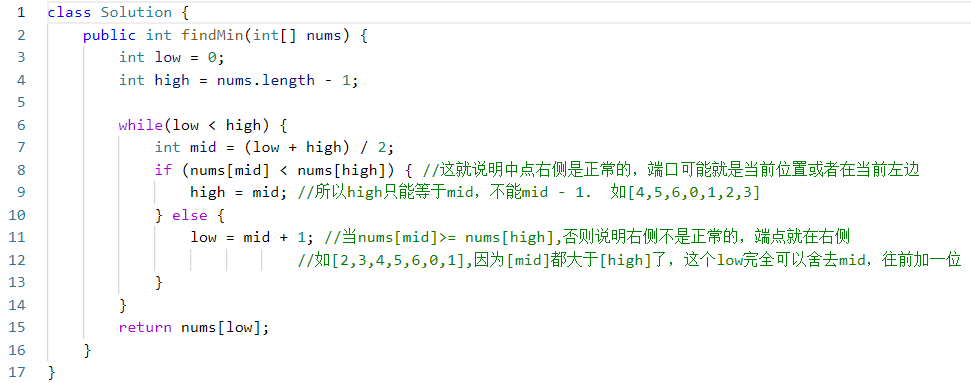


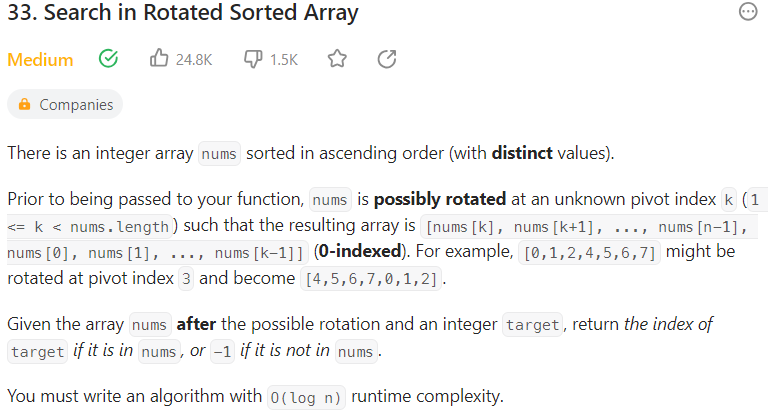
笔记：



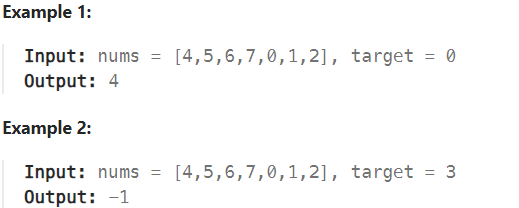
**ARRAY 数组**





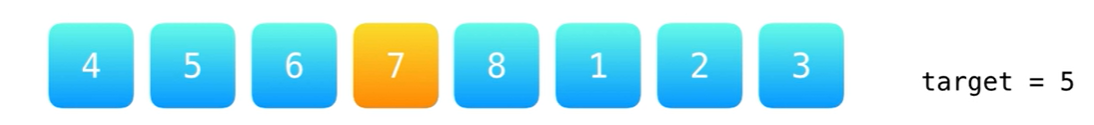


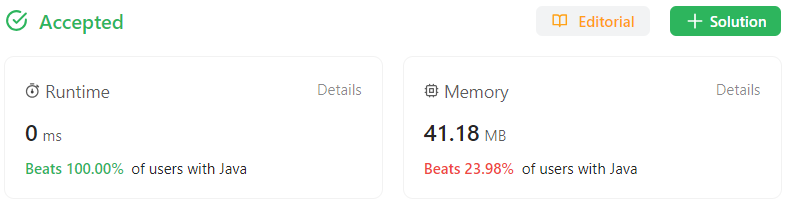
**ARRAY 数组**



分析：

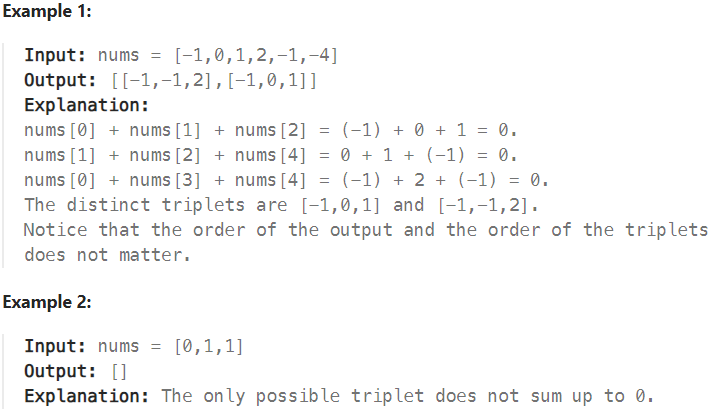
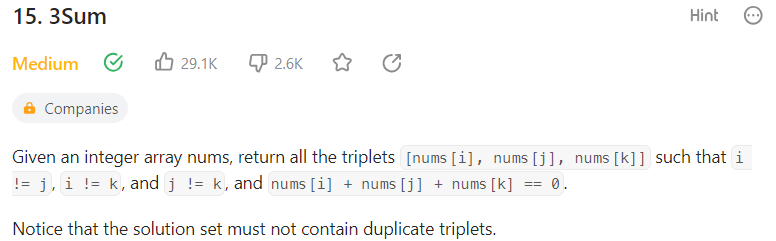
1. 就像上一题153那样，需要用到二分法。一说logN的时间复杂度肯定就考虑二分法
2. 还是同上一题的思路。二分出mid点，mid左右肯定有一边是有序数组
3. 如下，7为mid点，如果nums[left] <= 7,那么从left-mid减一之间肯定是有序的
4. 情况一：如果target就是7，那么返回mid就是要求的下标
5. 情况二：当左边是有序时，当target在左边时，即nums[left] <= target < nums[mid]，则right= mid - 1。否则target在无序的右边，left = mid + 1.
6. 情况三：当右边是有序时，如果判断nums[mid] < target <= nums[right]推出target在右边有序区时，则left = mid + 1, 否则相反，right = mid - 1；
7. 最终mid不断收缩到等于target，然后返回，如果没有则返回-1







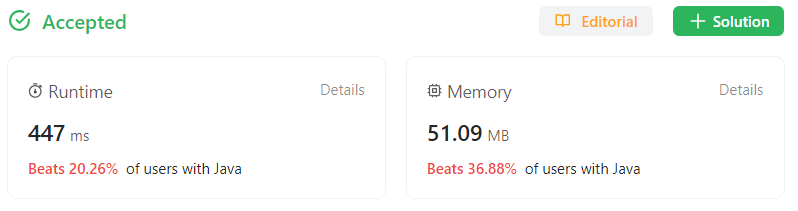
笔记：



**ARRAY 数组**

分析：

1. 用双指针来确定数据，用set来去重。把合适的都加进hashSet自动去重，最后加到result返回
2. 在for循环内，设置两个指针，因此就存在for循环当前i和左右两个指针
3. 这三个加起来可以判断是不是等于0.
4. 等于0就添加到set里面，重复的添加不进去，就相当于去重
5. 最后set赋值给题目要求的返回值





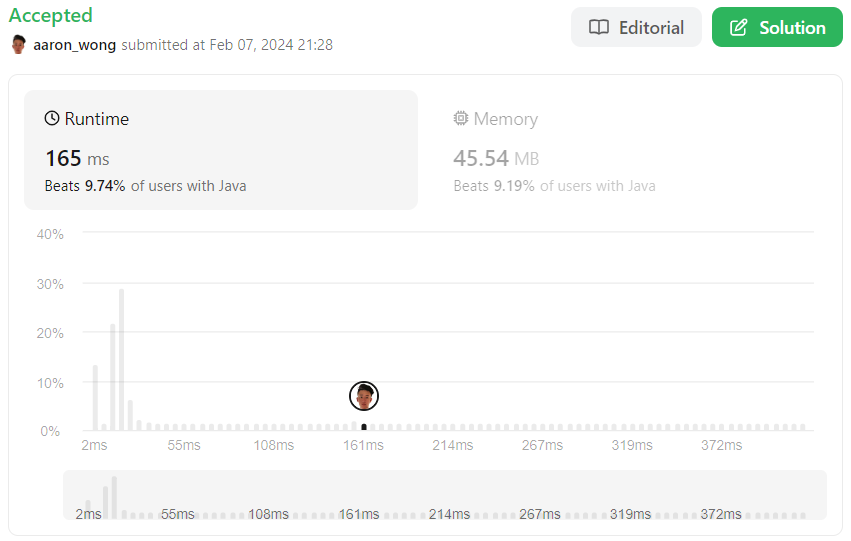
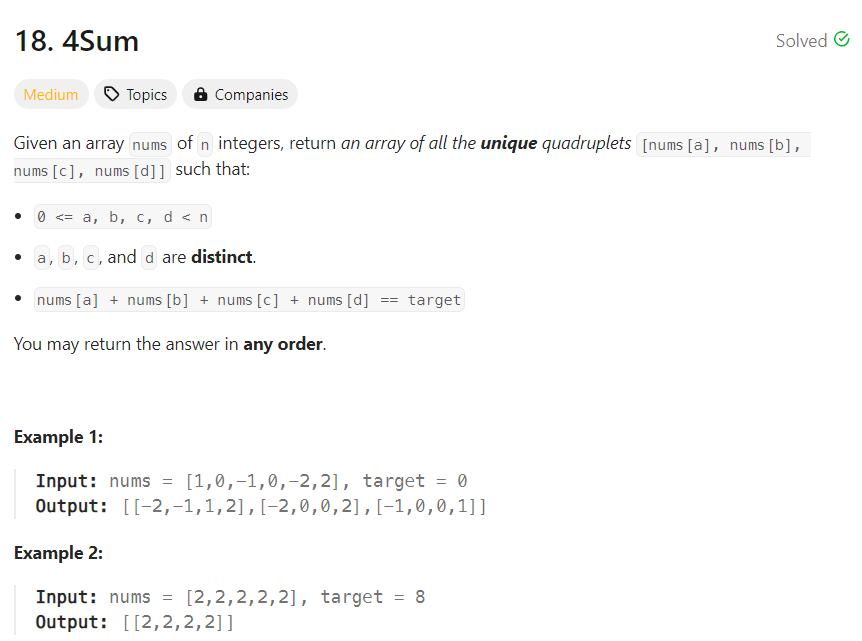
笔记：注意11行使用了Array.asList把他们转成了list集合。arrays.aslist是数组array里的一个方法，参数可以传数组名也可以传几个数字，最后生成集合list，但是这个集合是固定长度的。

Ex：数组[1,0,-1]和[1,-1,0]的地址值不一样，都会被加入HashSet里

但是集合<> [1,0,-1]和[1,-1,0]是会被判定一样，因此只有一个加入hashSet

*对的，你理解得很正确。在Java中，数组的equals方法默认行为是比较引用地址，即两个数组只有在引用相同的情况下才被视为相等。因此，即使数组中的元素相同但顺序不同，它们的地址是不同的，将它们作为不同的元素加入HashSet是可行的。*

*相反，List的equals方法是根据元素的equals方法来判断的，所以如果两个List包含相同的元素，即使顺序不同，它们在equals比较中会被视为相等。这也是为什么通过Arrays.asList将数组转换为List后，排序不同但包含相同元素的List会在Set中被视为相同的原因*



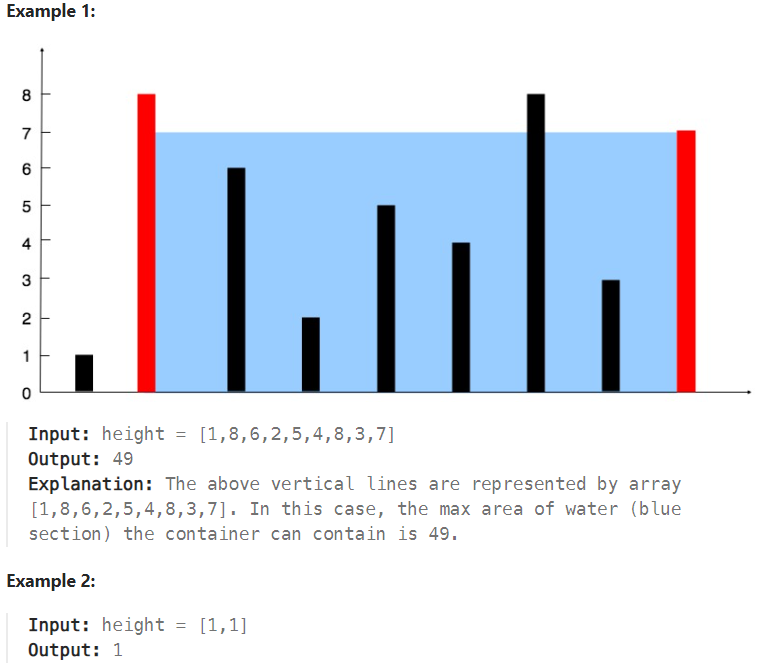
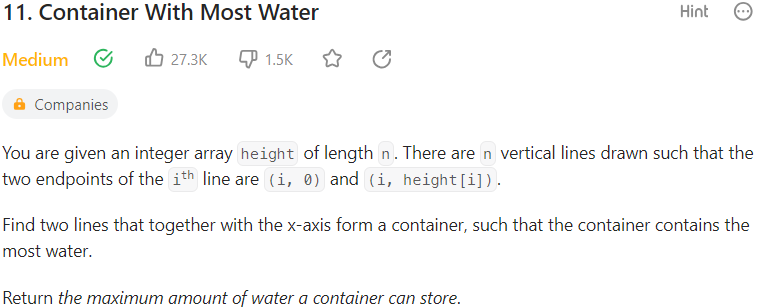
**ARRAY 数组**

笔记：和上一题3sum是一样的思路。

3sum是一个i指针，然后分别左右指针。三个指针来凑target

4sum就是除了i指针还有j指针（i+1）,然后在是左右指针凑target。

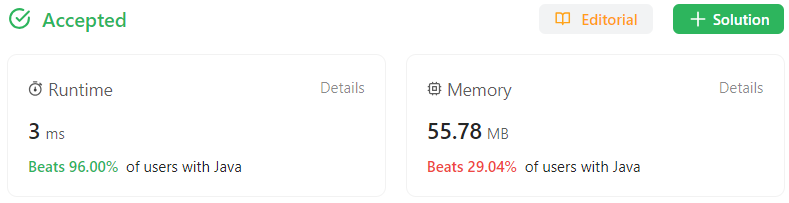
所有如果还有5sum，大抵也是i,j,k,然后左右指针凑target

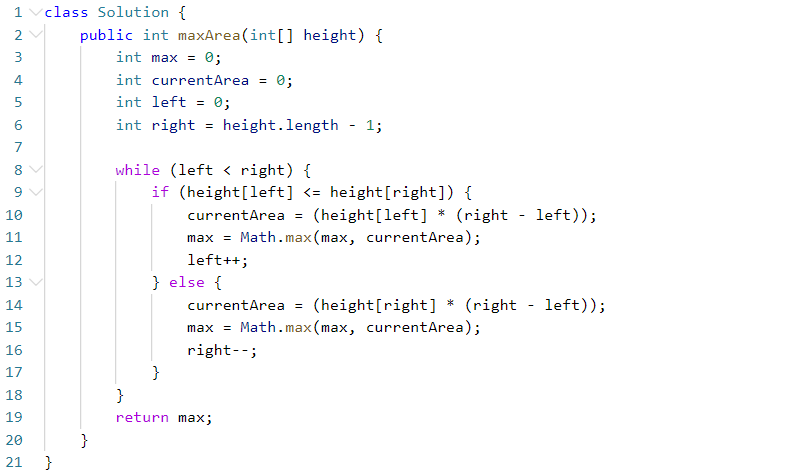


**ARRAY 数组**

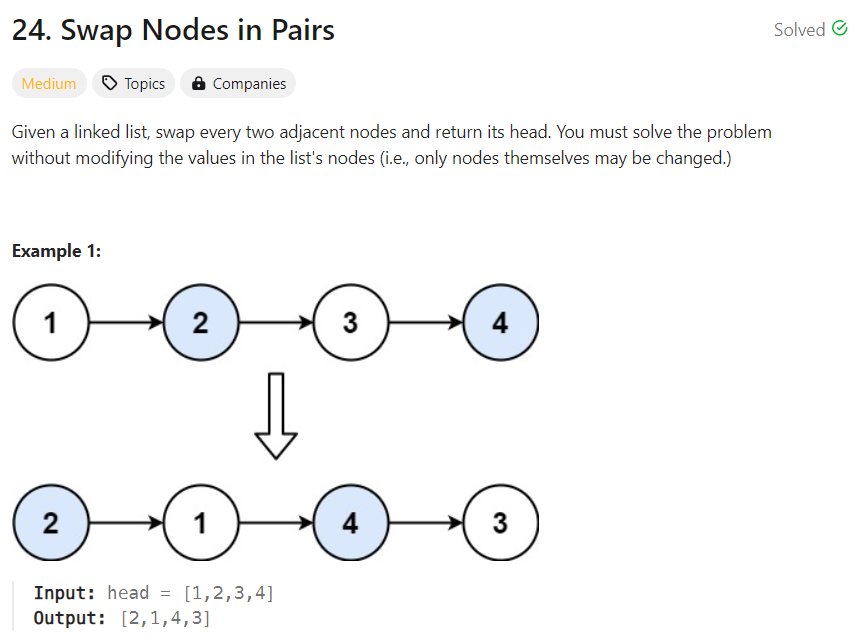
思路：

1. 采取的双指针和贪心算法
2. 左右两边各设置指针，每次固定较长的那条边，移动较短那边，使得底减少的同时尽可能面积大。
3. 再用max来实现贪心算法，逐步突破max的天花板，最后保留得到最大值。

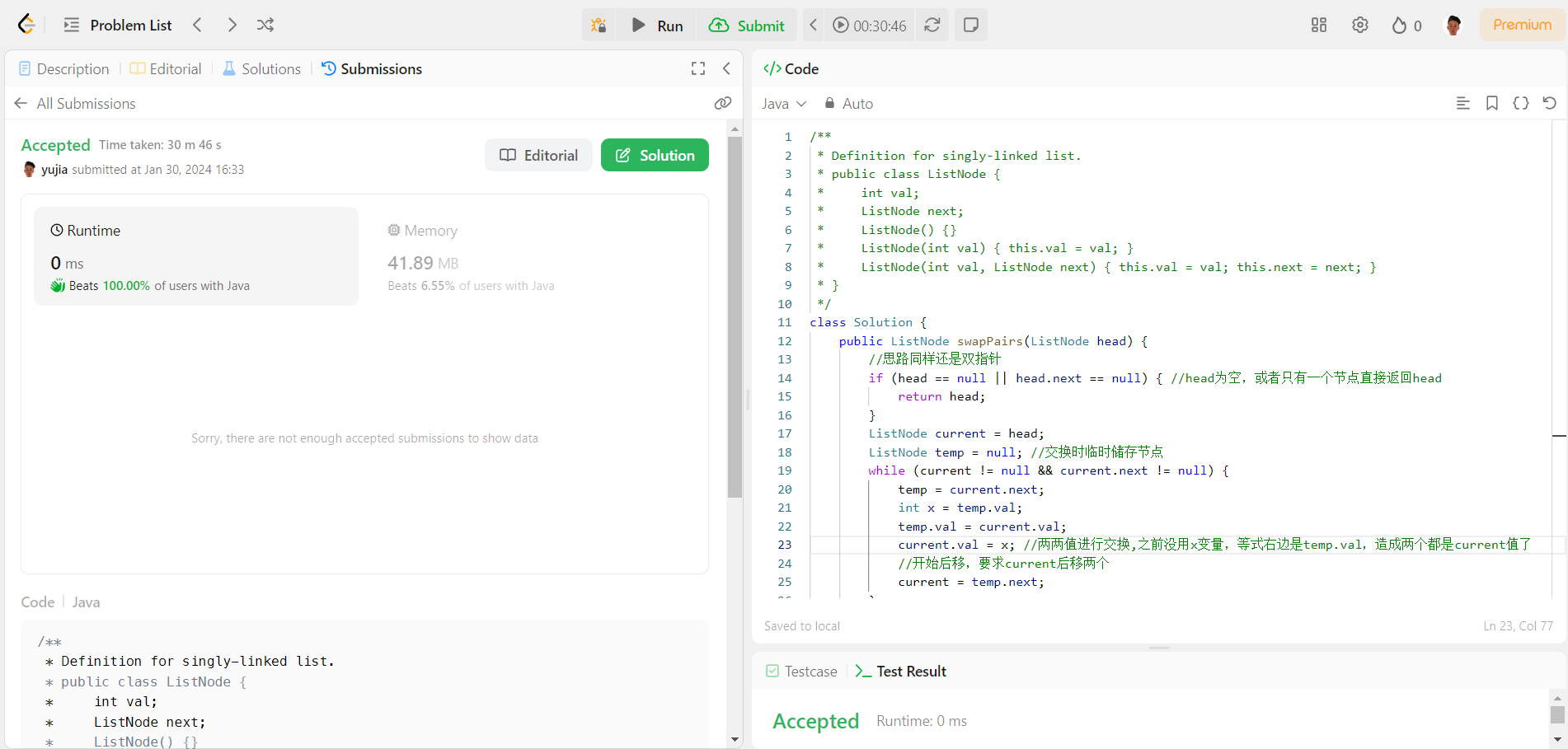


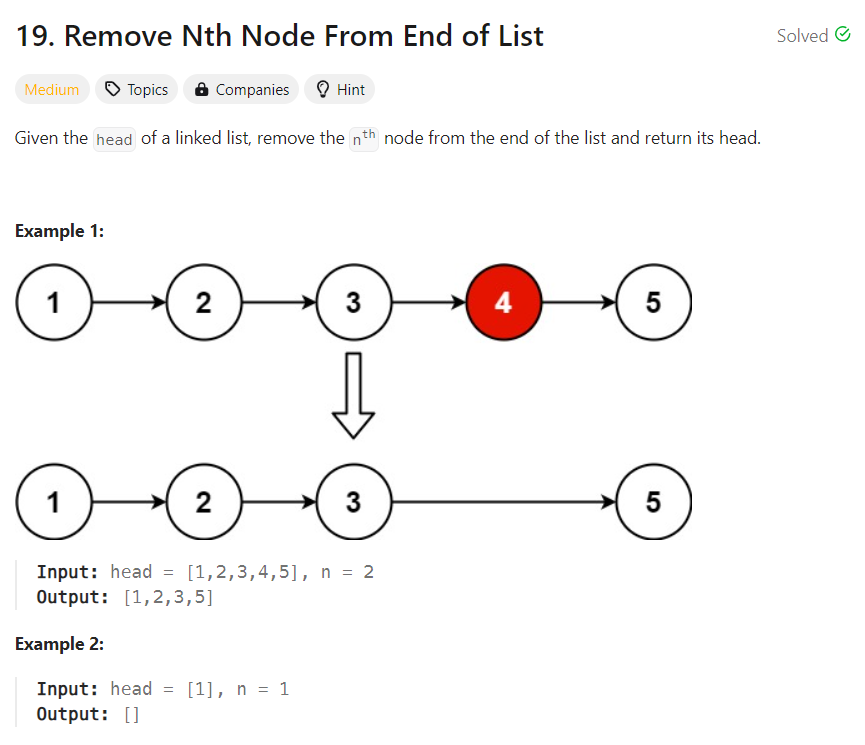
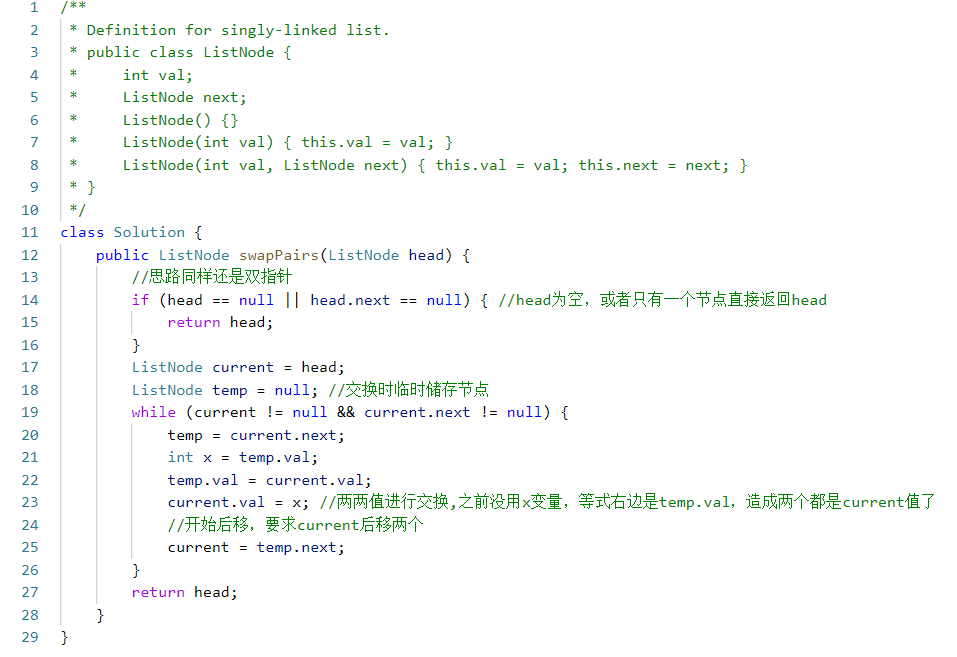


笔记：

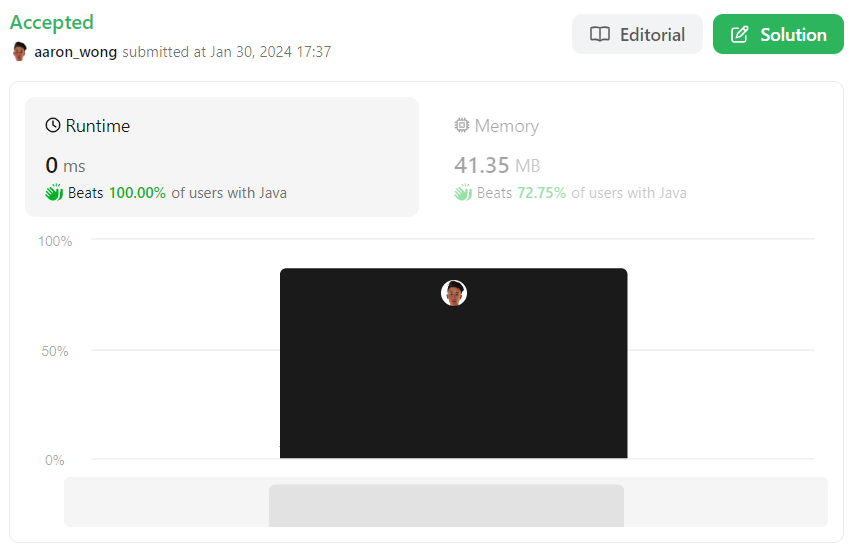


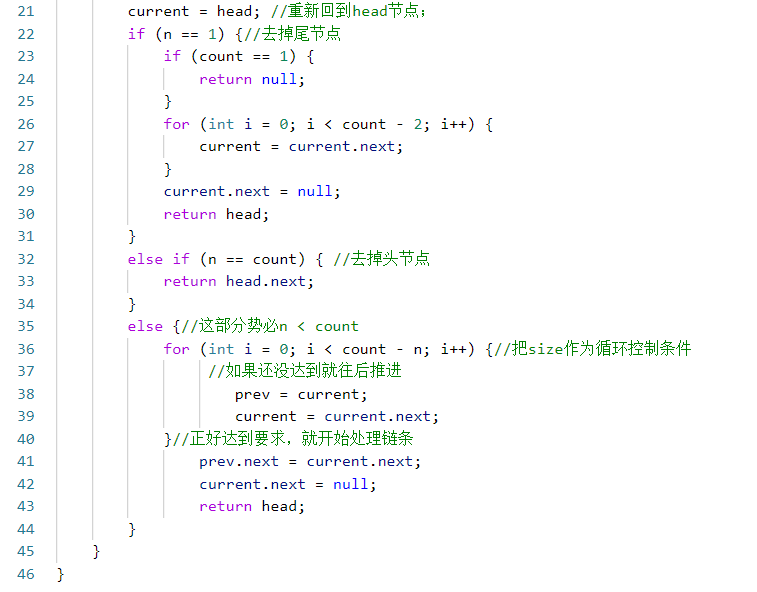
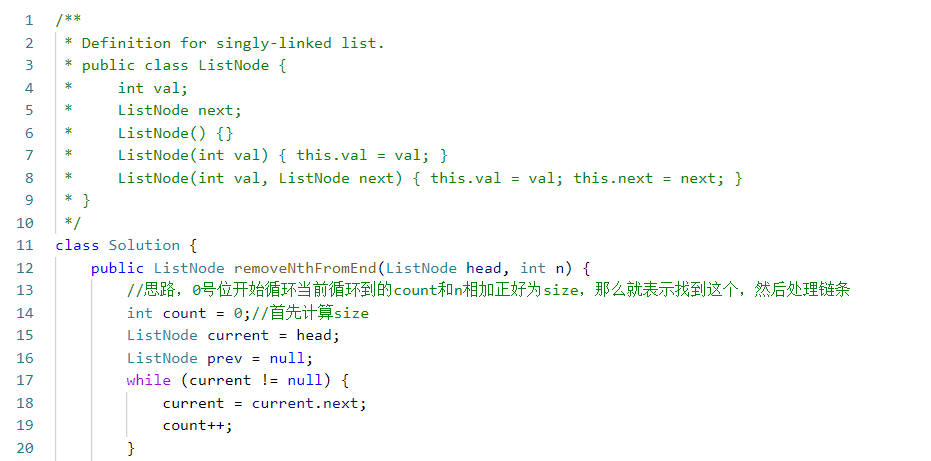
**LinkedList 链表**

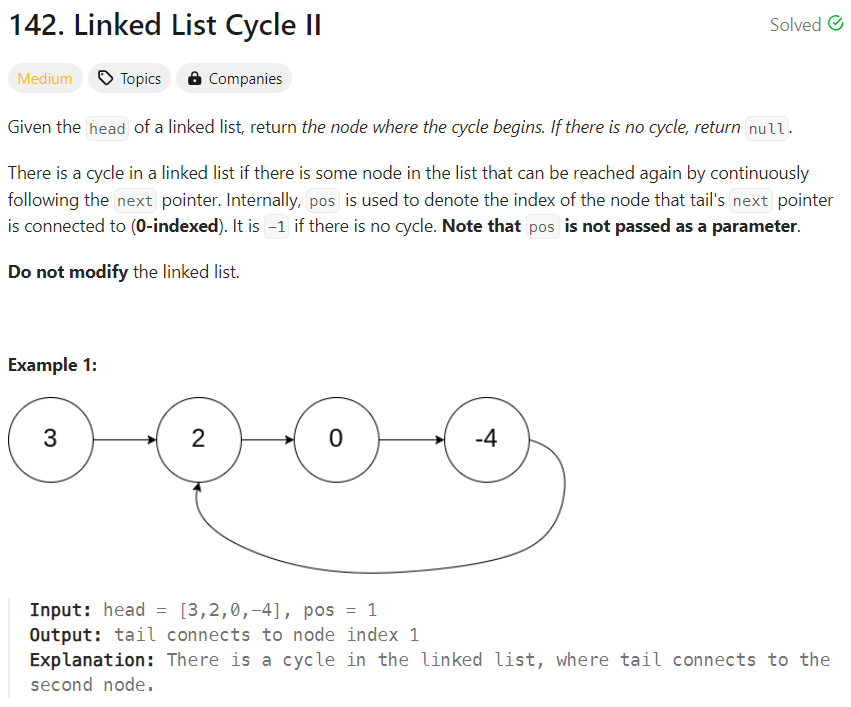




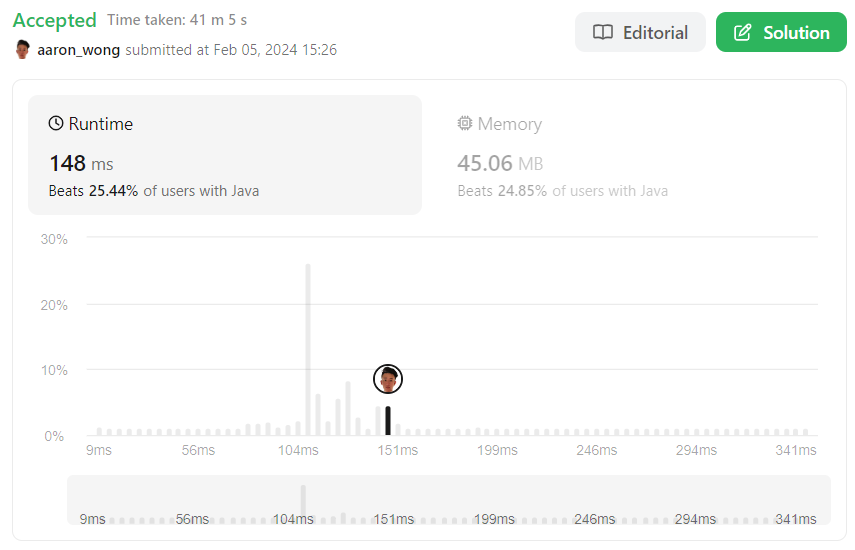
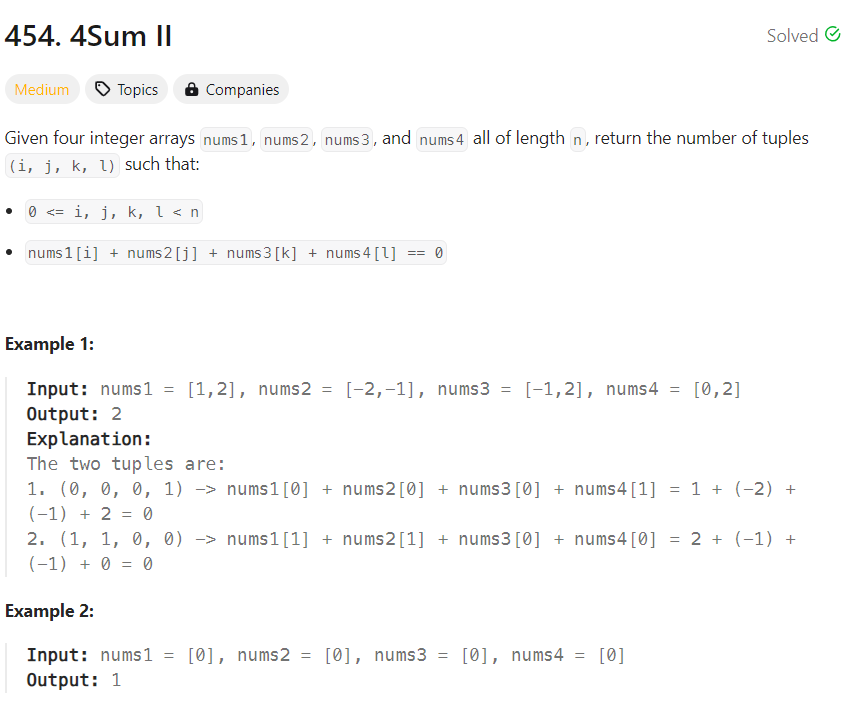
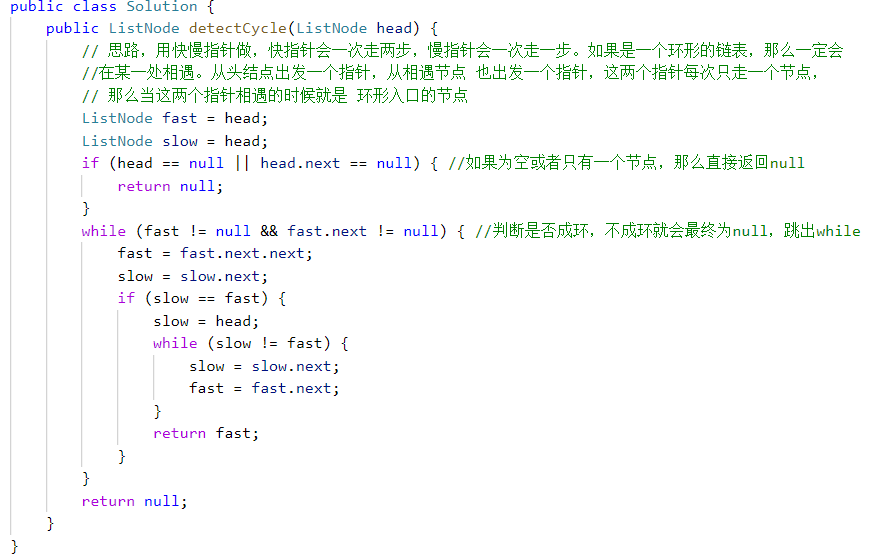
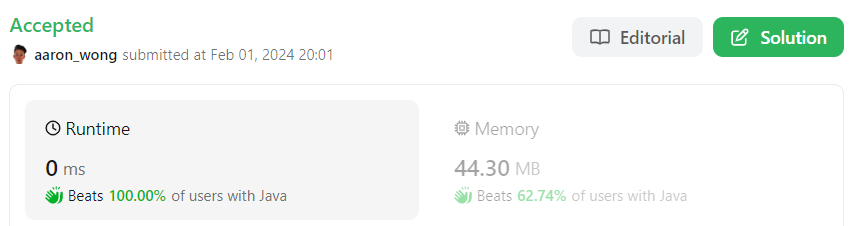
**LinkedList 链表**



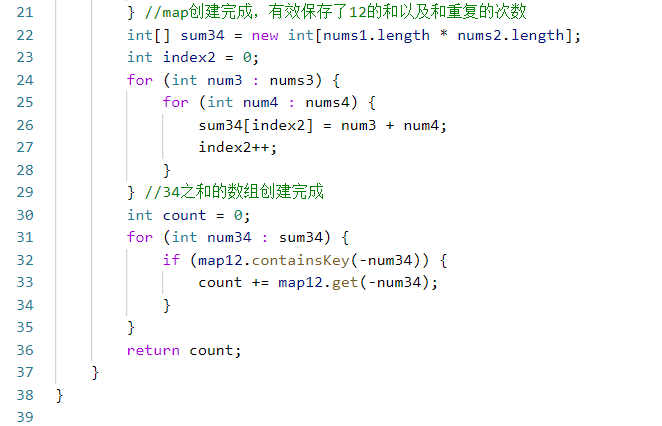


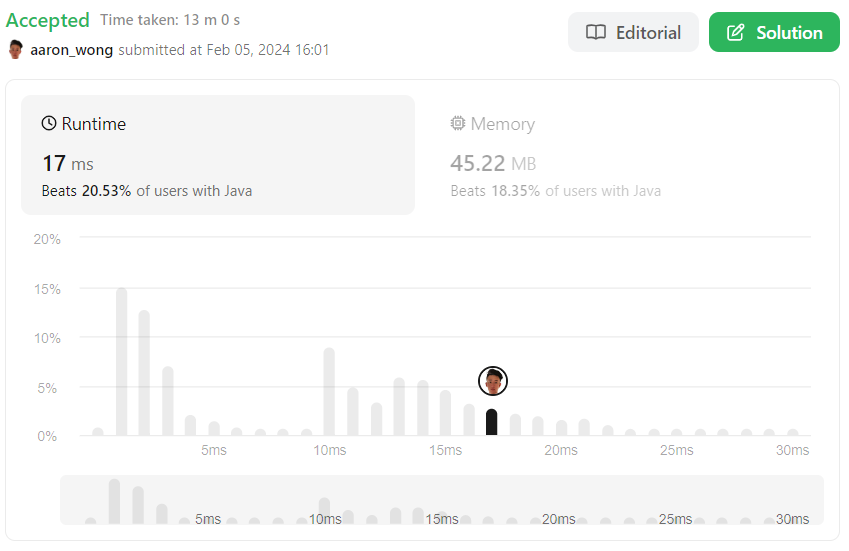
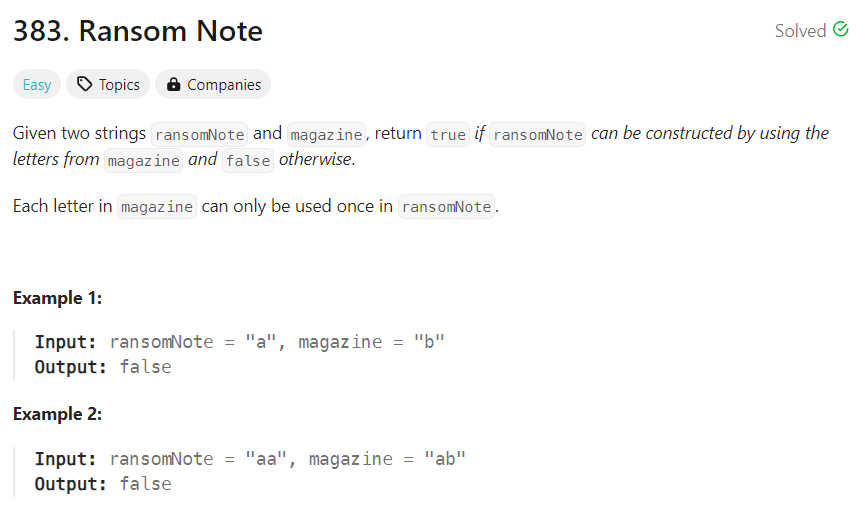


**LinkedList 链表**

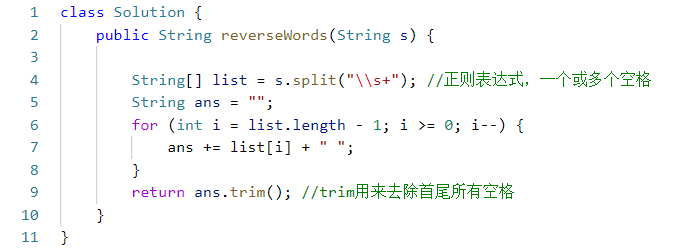
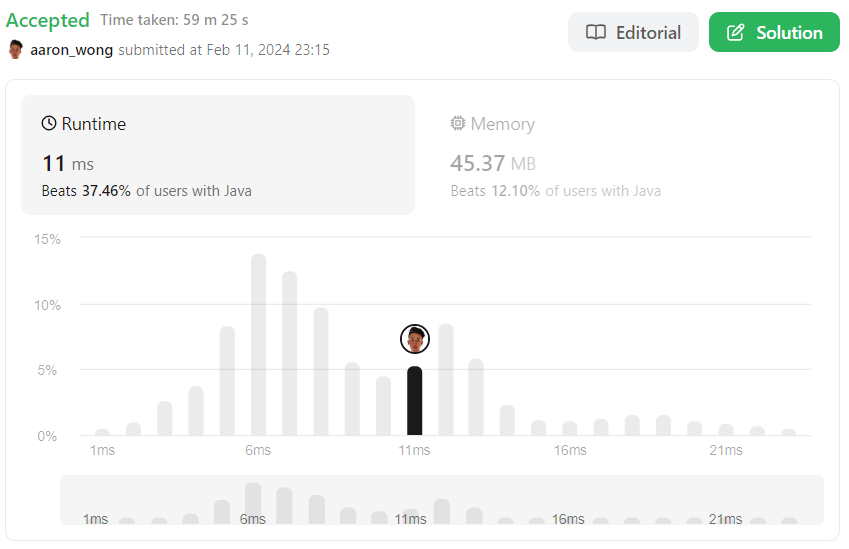
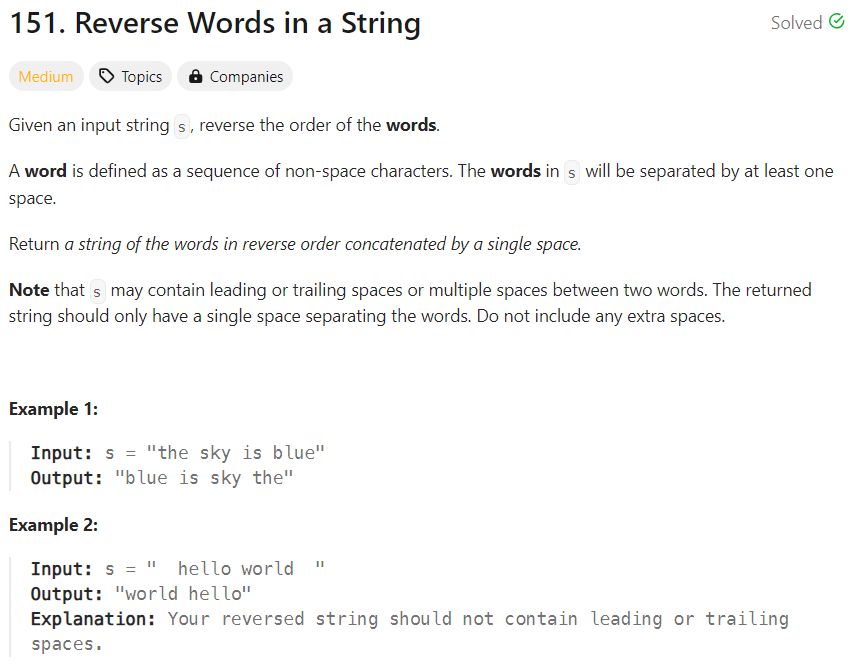


**Hash 哈希表**

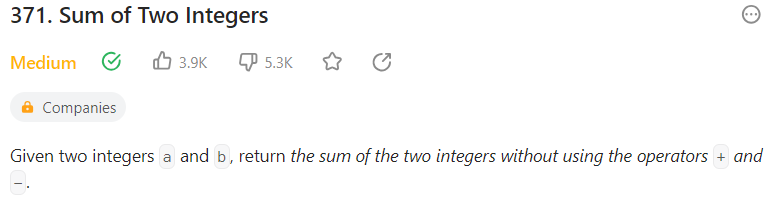




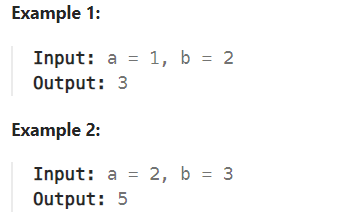
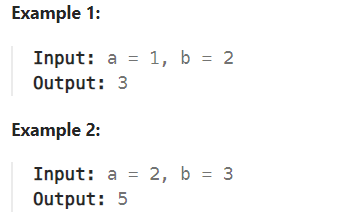
**Hash 哈希表**



**String字符串**

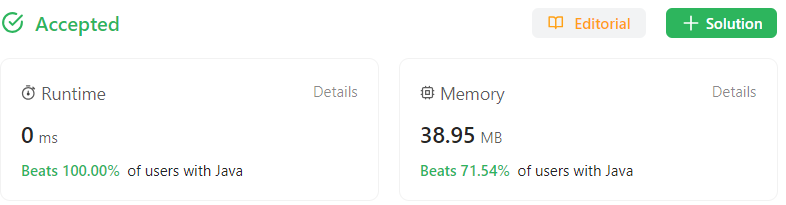


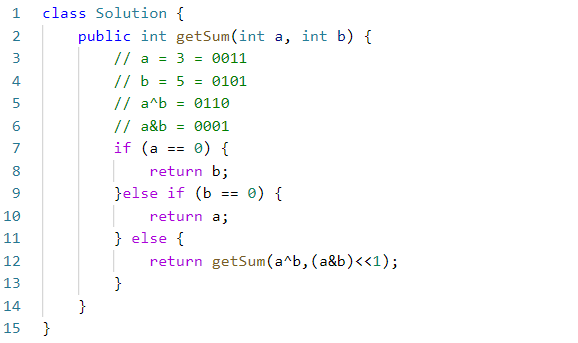
**BINARY位运算**

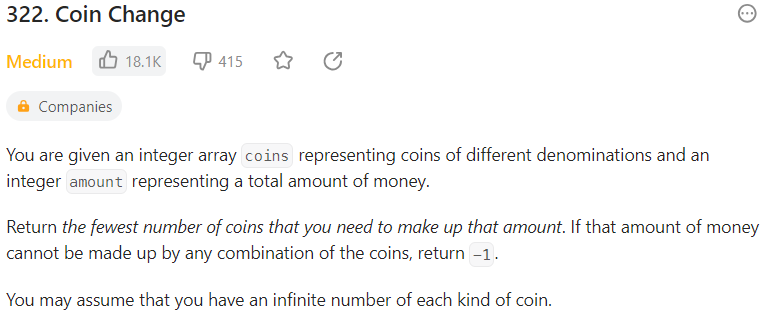


思路：

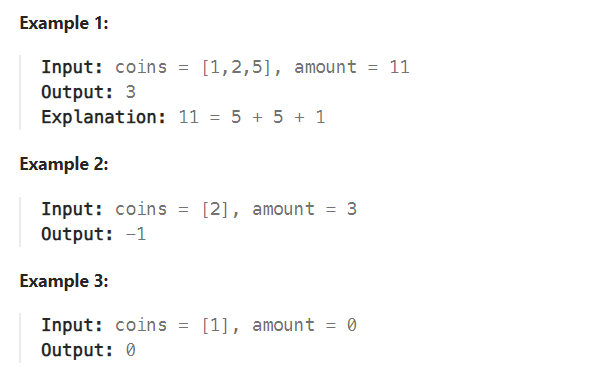
1. 不用等常规十进制运算，首先考虑二进制位运算（与，或，异或，取反，左右移）
2. 规律一：与&运算，同位次两者都为1则得1，否则为0。而1加1就得进为，是不是相当于答案得出1的位次表明了是要进位的位次，为规律一。
3. 异或就是不进位的加法，因为异或^运算时，相同为0，不同为1，本来二进制形式只有10，所以当然理解为不进位加法。
4. 第2，3点综合来看，加上左右移来补一位0就能得出答案
5. 总结，异或完成不进位加法，bug是相同为0，即1+1也得0。而与运算则1+1才得1，同时左移一位0，正好和异或互补。







**Dynamic Programming动态规划**



思路：

**状态定义：**

dp[i] 表示凑成金额 i 所需的最少硬币数量。

**状态转移方程：**

dp[i] = Math.min(dp[i], dp[i - coin] + 1)：在遍历每个金额 i 的过程中，不断更新 dp[i] 的值为使用不同硬币面额时的最小硬币数。这里的状态转移方程表明了一个重要的思想：当前问题的最优解依赖于子问题的最优解，通过子问题的最优解来更新当前问题的解。

**初始化：**

dp[0] = 0 表示凑成金额为 0 时所需的硬币数量为 0。

其他元素初始化为一个足够大的值，这里是 amount + 1，表示初始时无法凑成。

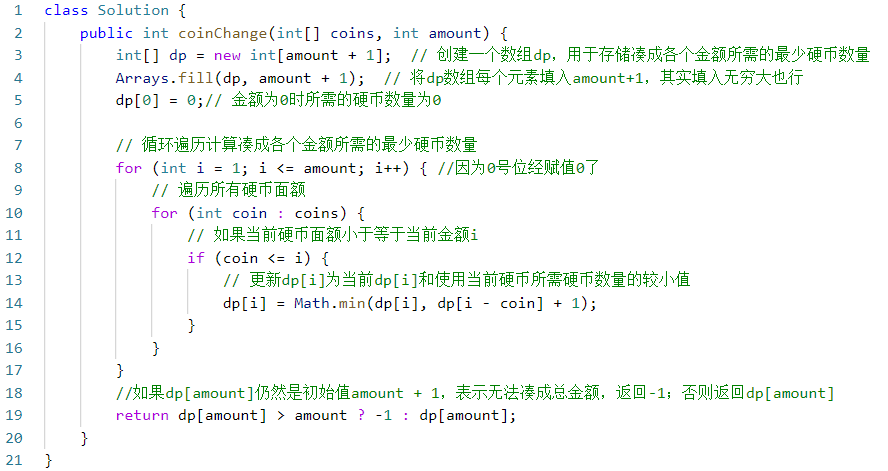
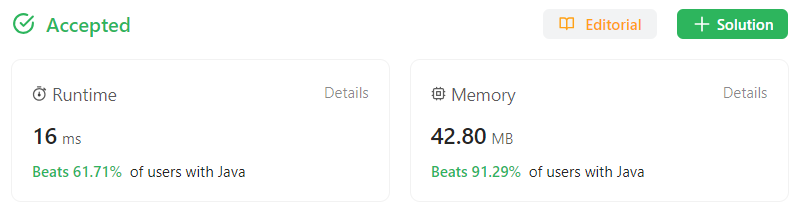
**循环求解：**

通过两层循环，外层循环遍历金额 i，内层循环遍历所有硬币面额。通过状态转移方程更新 dp[i] 的值。

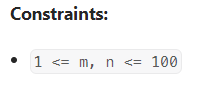
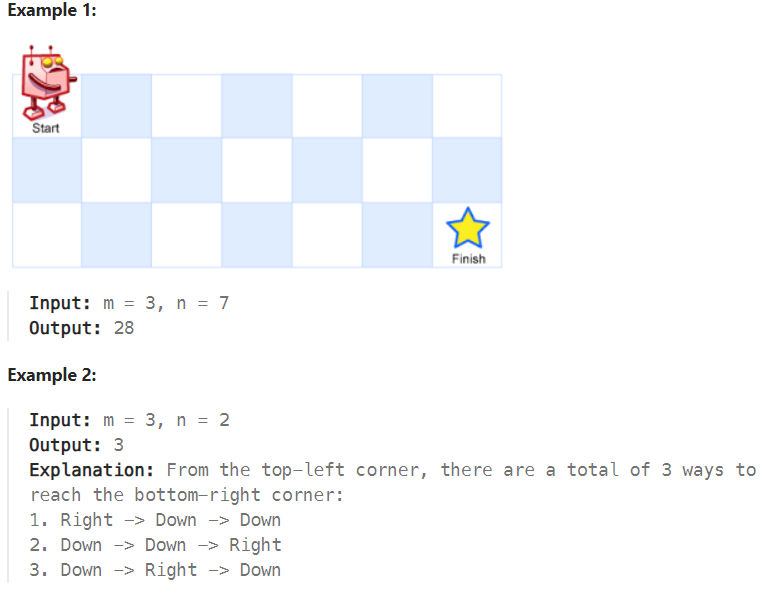
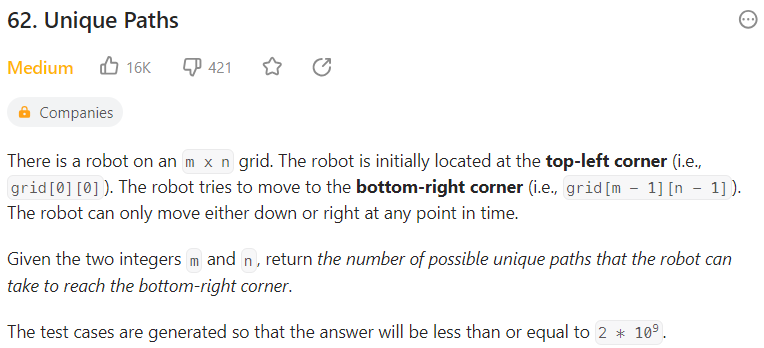
**最终结果：**

返回 dp[amount]，即凑成总金额所需的最小硬币数量。

这个过程体现了动态规划的关键点：通过建立状态转移方程，利用之前计算得到的子问题的最优解来求解更大规模的问题，最终得到了凑成指定金额所需的最少硬币数量



笔记：



**Dynamic Programming动态规划**

思路：严格按照代码随想录的dp五部曲来

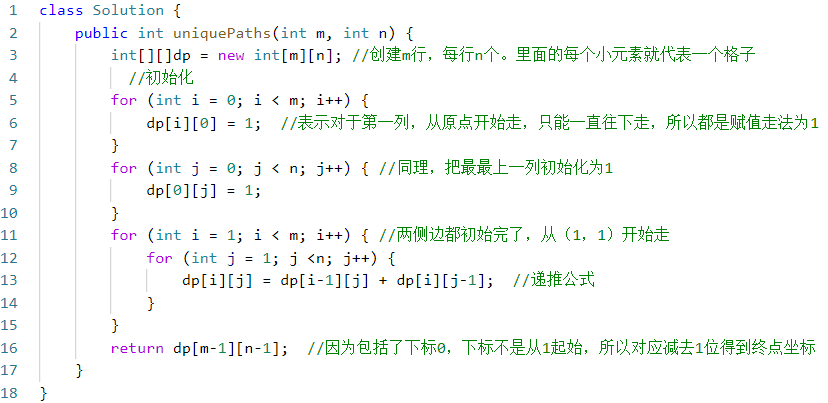
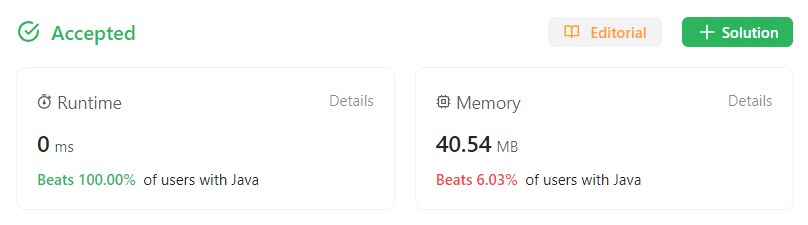
1.dp[]以及下标表示的含义

2.递推公式

3.dp数组如何进行初始化

4.遍历顺序（几层循环，循环顺序）

5.打印dp数组

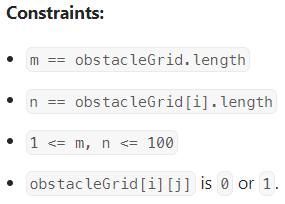
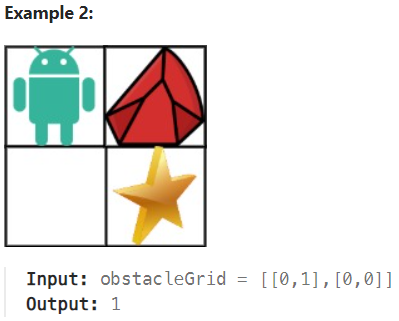
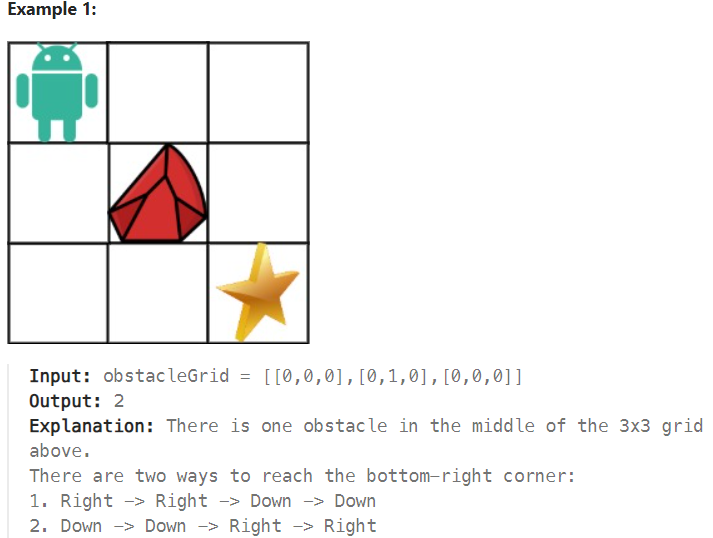
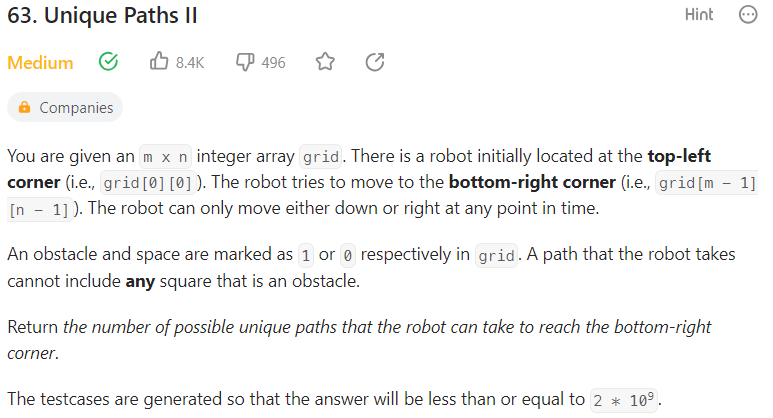


笔记：

感觉都是找到某一个点，然后相邻的点能得出这个点，通过共同规律

比如之前的爬楼梯，某级台阶可以从前一级和前两级走到

而本题，从上面一个格子和左边一个格子可以得到，格子可以按这种逻辑一直溯源到最基本的单元，也就是最上行和最左列，而这两排的没个格子从原点出发的方式都是0，即一路向右或一路向下。



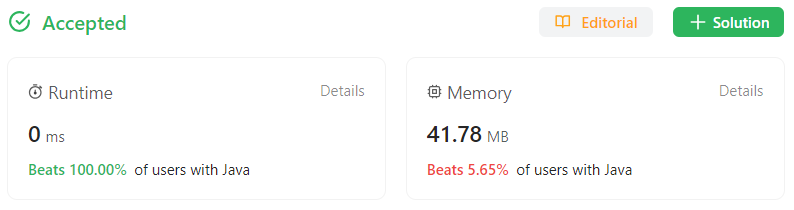
**Dynamic Programming动态规划**

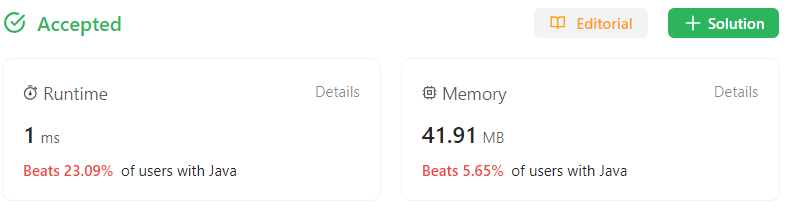
思路：

两个方法主要区别是在双重for循环那里，第一种是直接continue，如果遇到障碍物就跳过，此时dp里对应的格子的值还是保持0，不会进行赋值

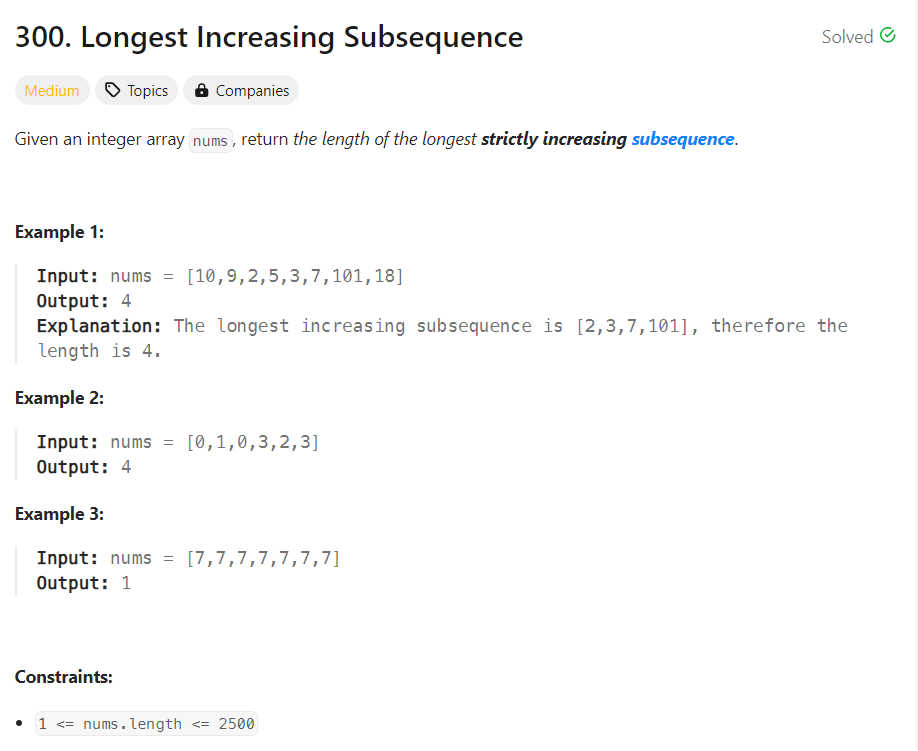
第二种是分情况讨论，上左无障碍，上有，左有，上左都有

所以区别更像是第一种从每个格子的有效性入手，而第二种是从路径方面入手

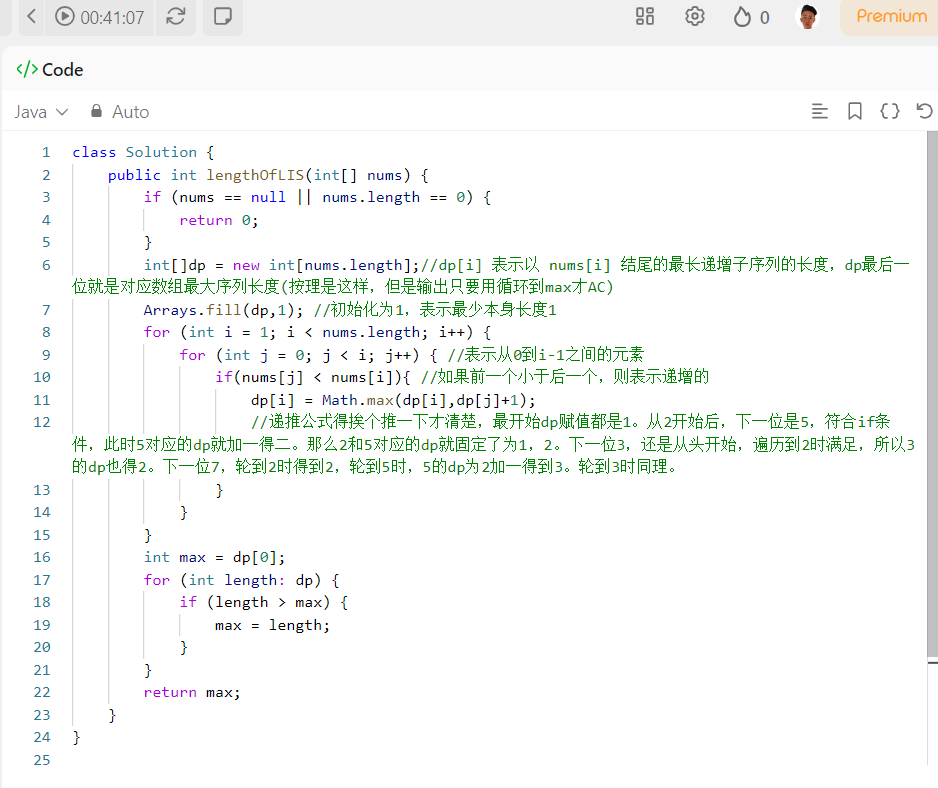
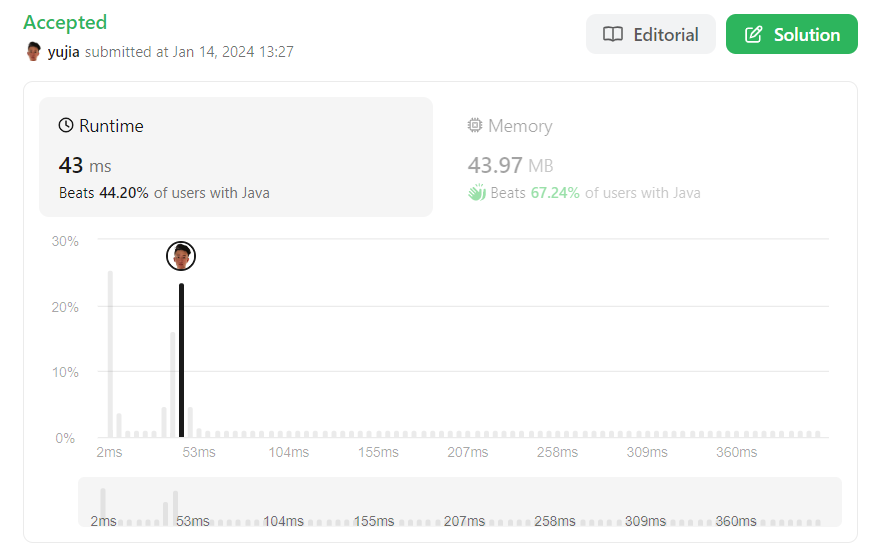




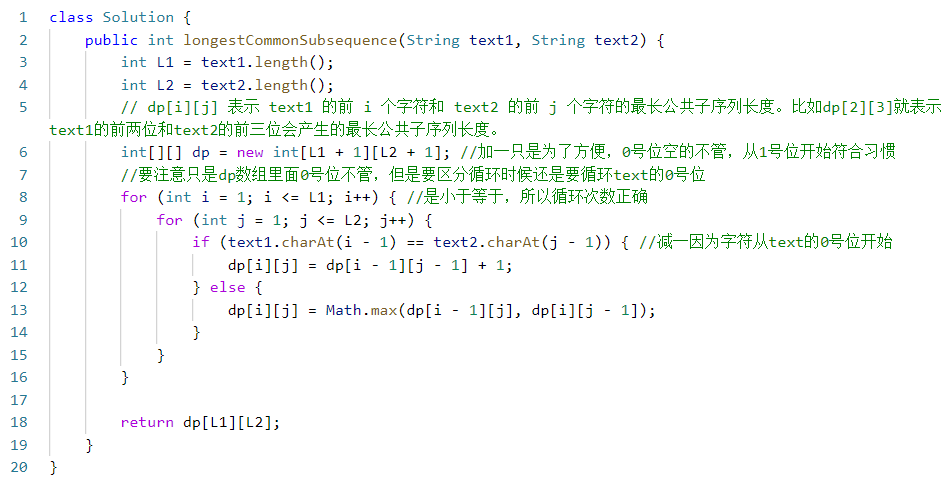
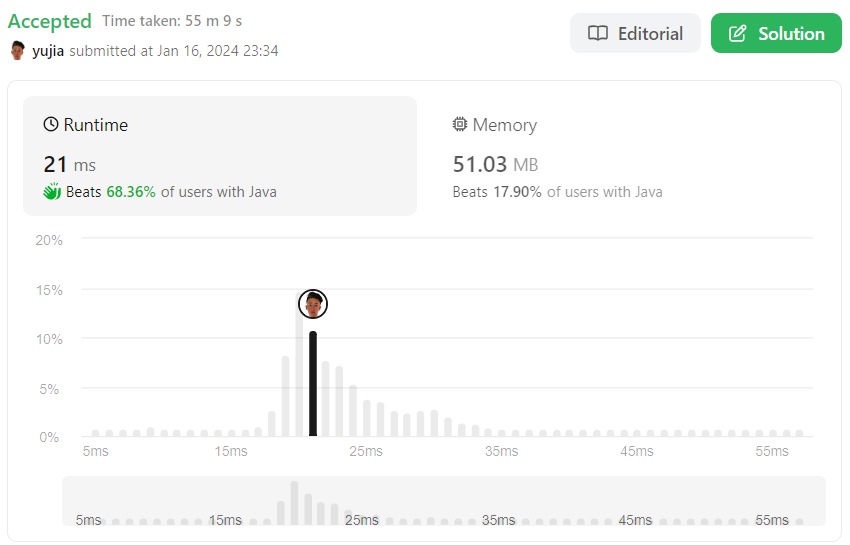
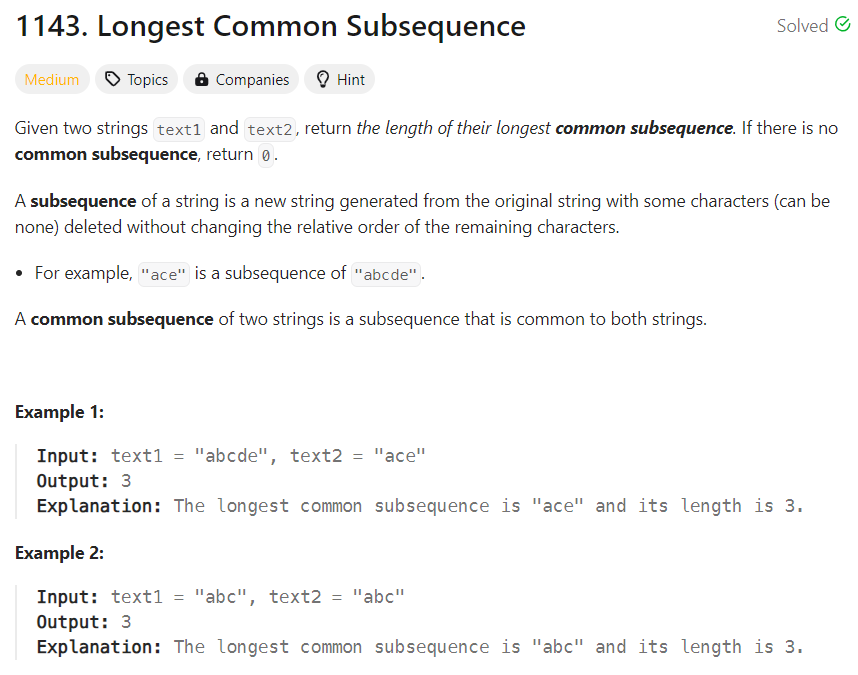
笔记：



**Dynamic Programming动态规划**

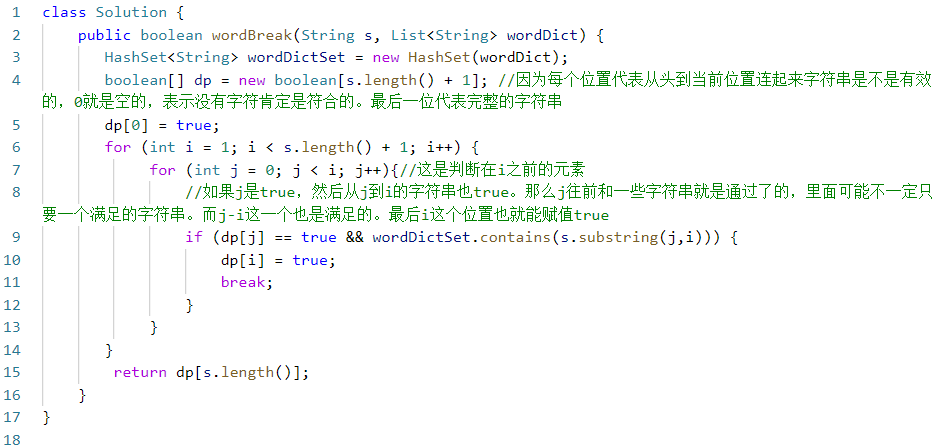
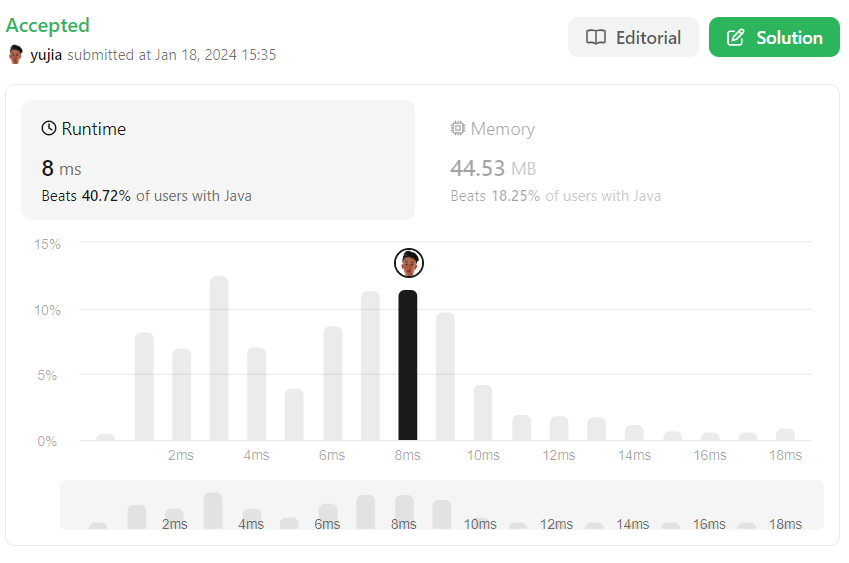
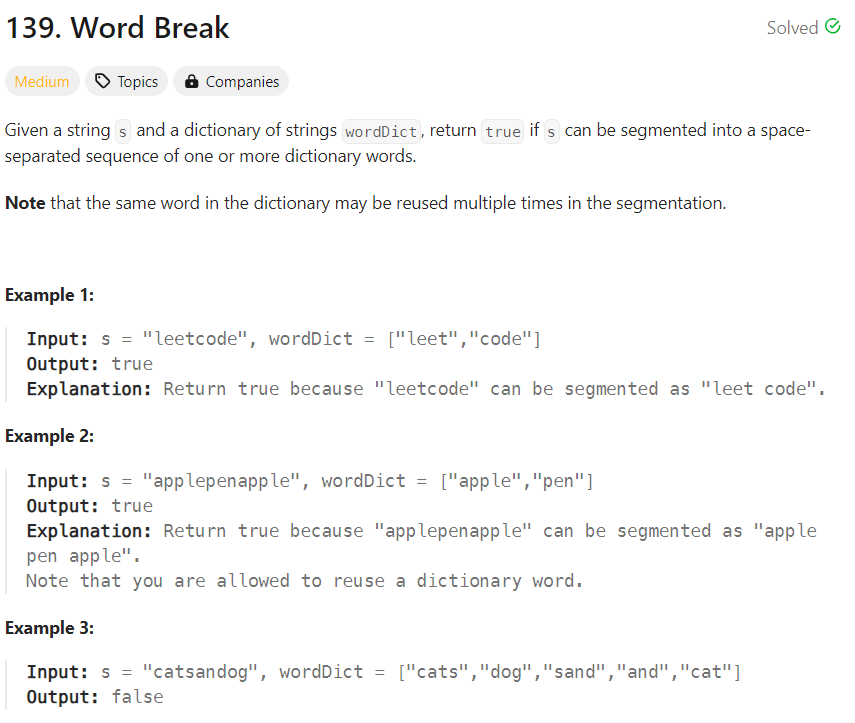


笔记：这题非常经典，是高频题。理解稍微拗口，不行就背



**Dynamic Programming动态规划**

笔记：和392有点类似

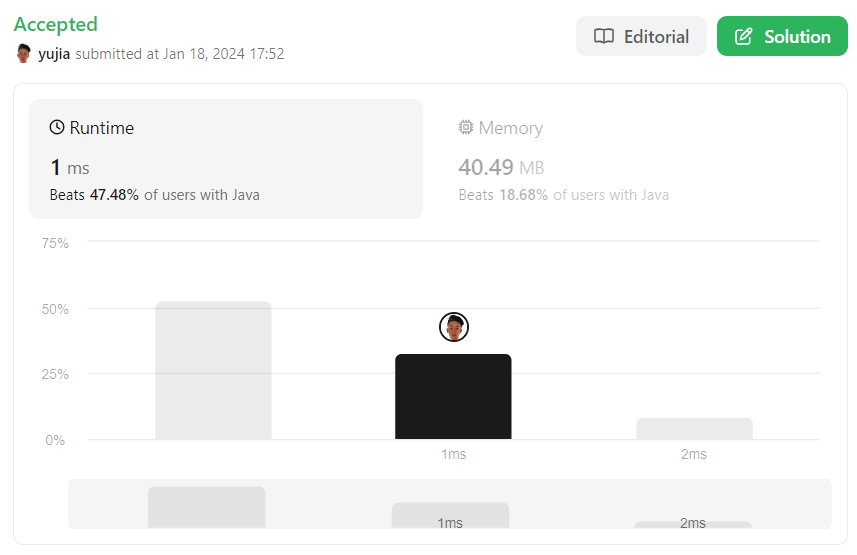
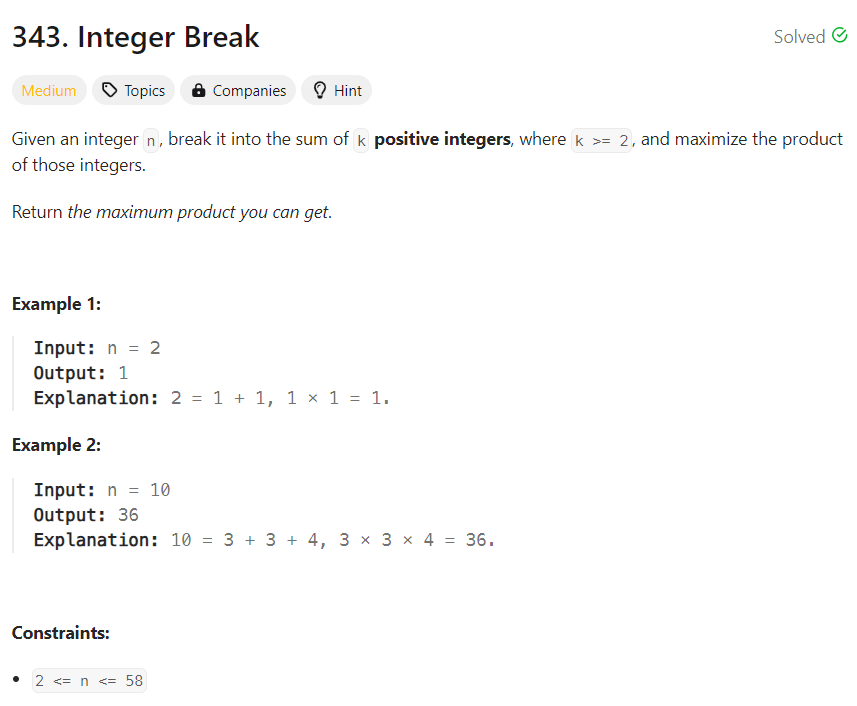


**Dynamic Programming动态规划**

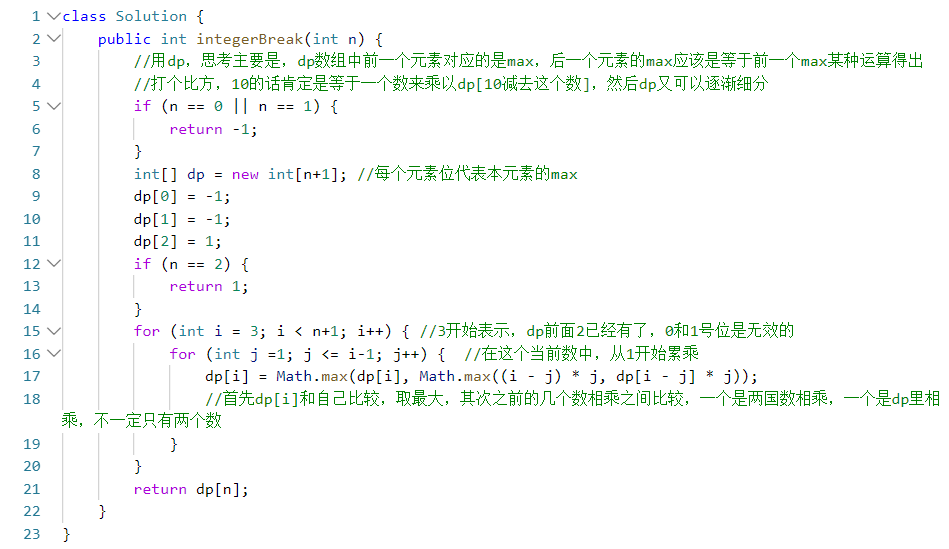
笔记：切记动态规划就是由数组之前的值来得出后面的值。

以这题来说，思路就是i在外层循环，j在0-i之间循环。j会循环到一个分界点，这个分界点是之前验证满足条件的字符串和后续新的还没验证的字符串。

总的来说就是验证okay的字符串+被验证为contain的新字符串，这两个条件相加就得到此时i点也应该赋值true



**Dynamic Programming动态规划**



笔记：