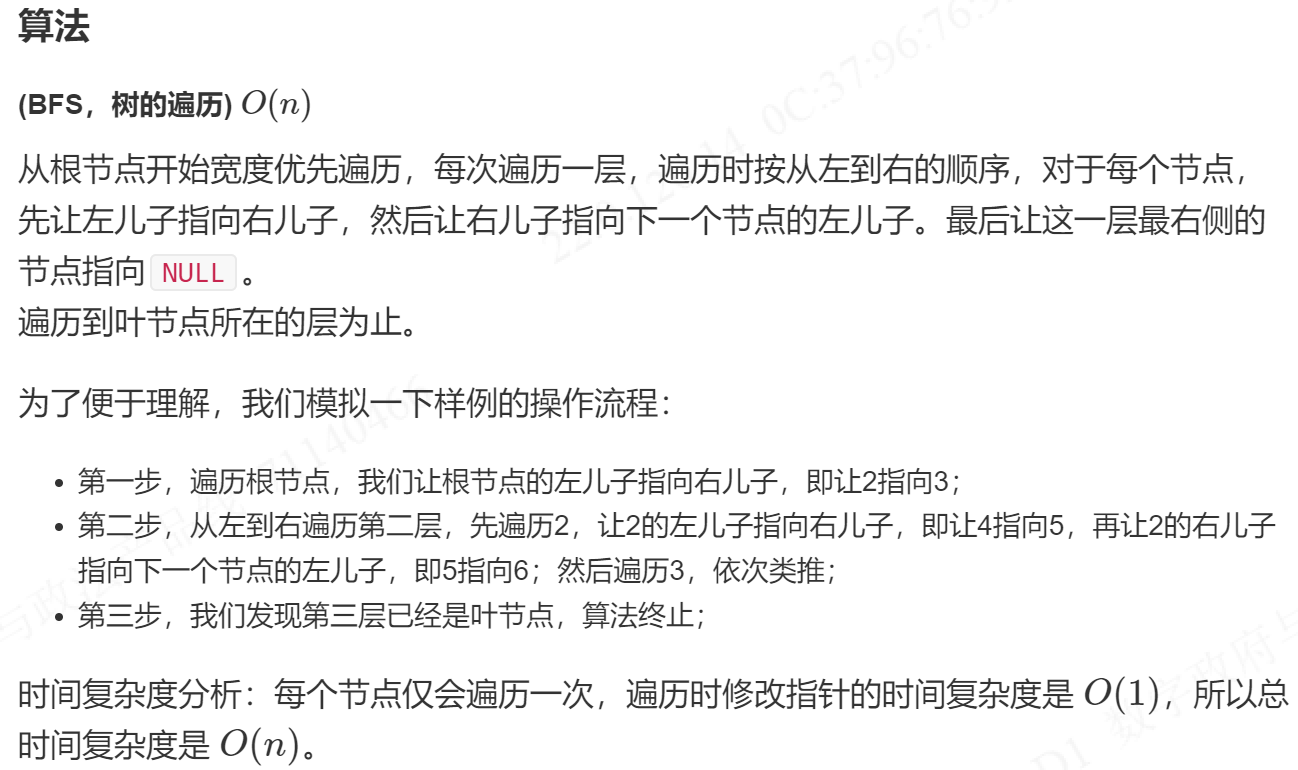
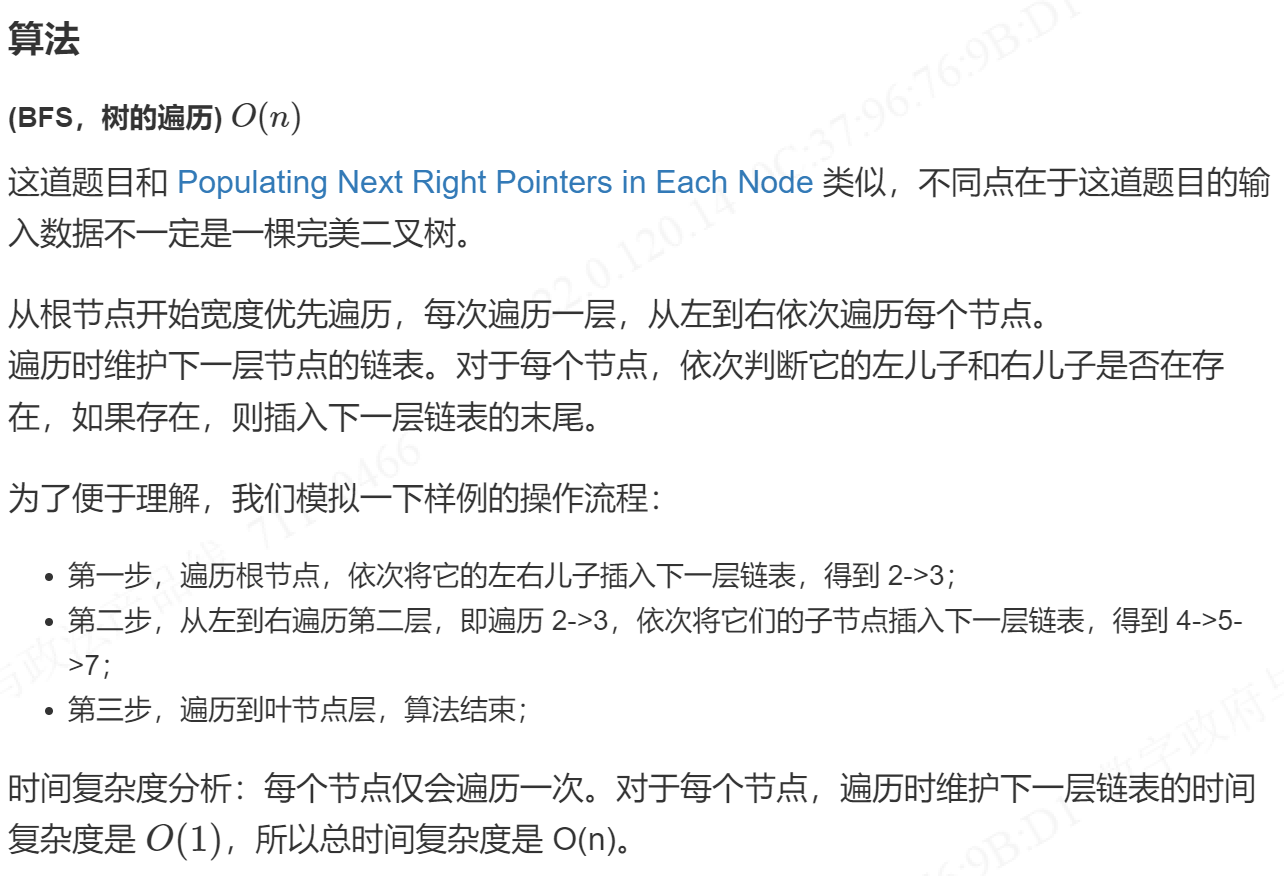
**116.Populating Next Right Pointers in Each Node**



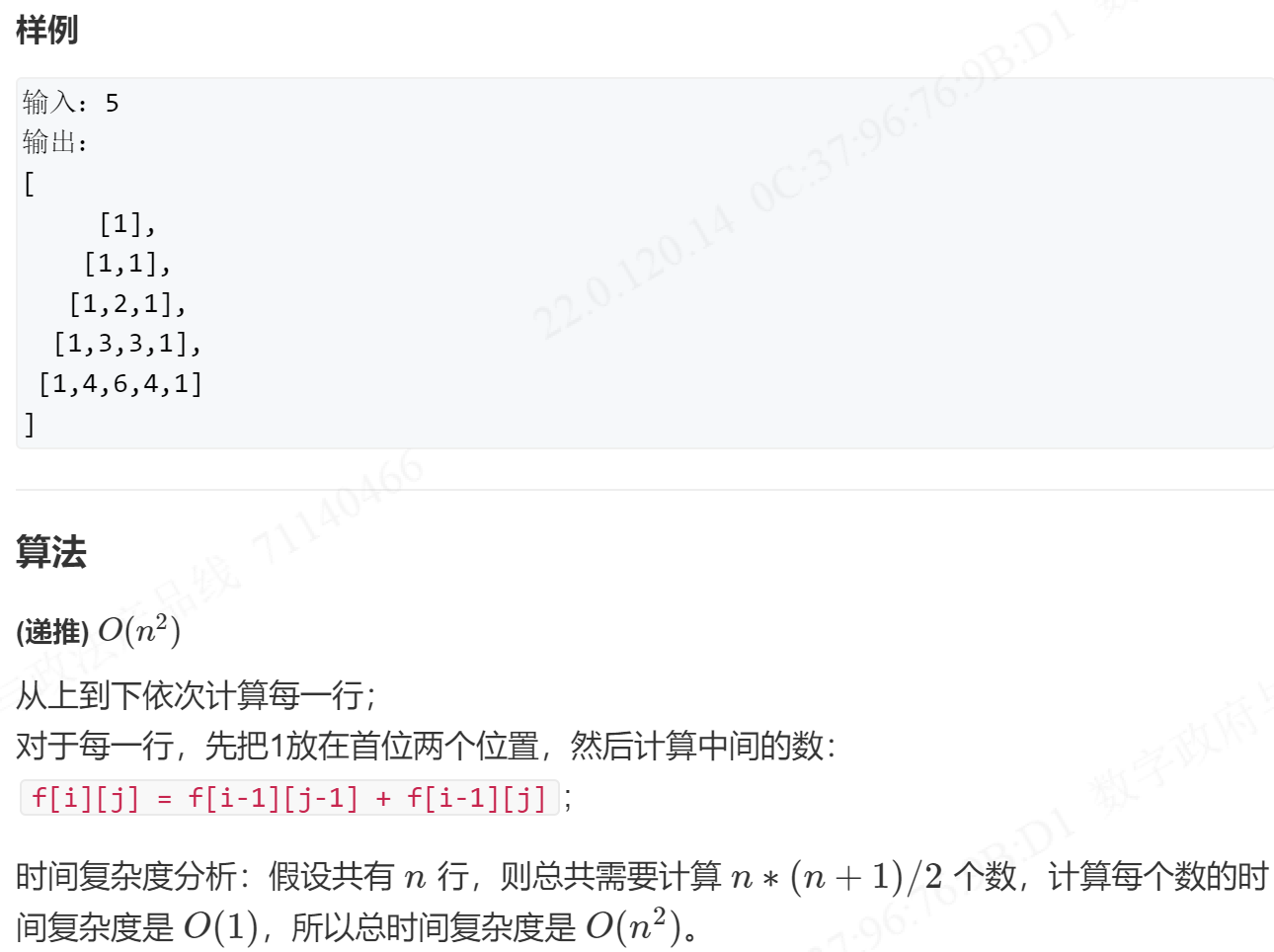


**117.Populating Next Right Pointers in Each Node II**





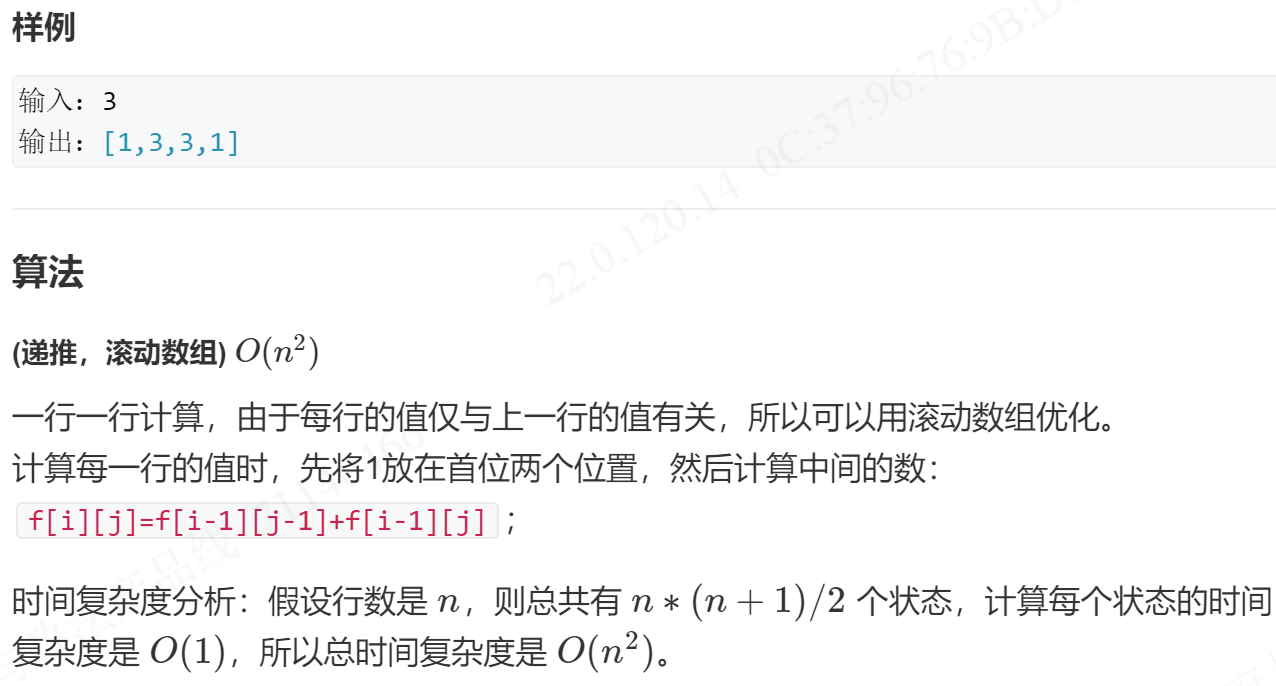
**118.Pascal's Triangle**



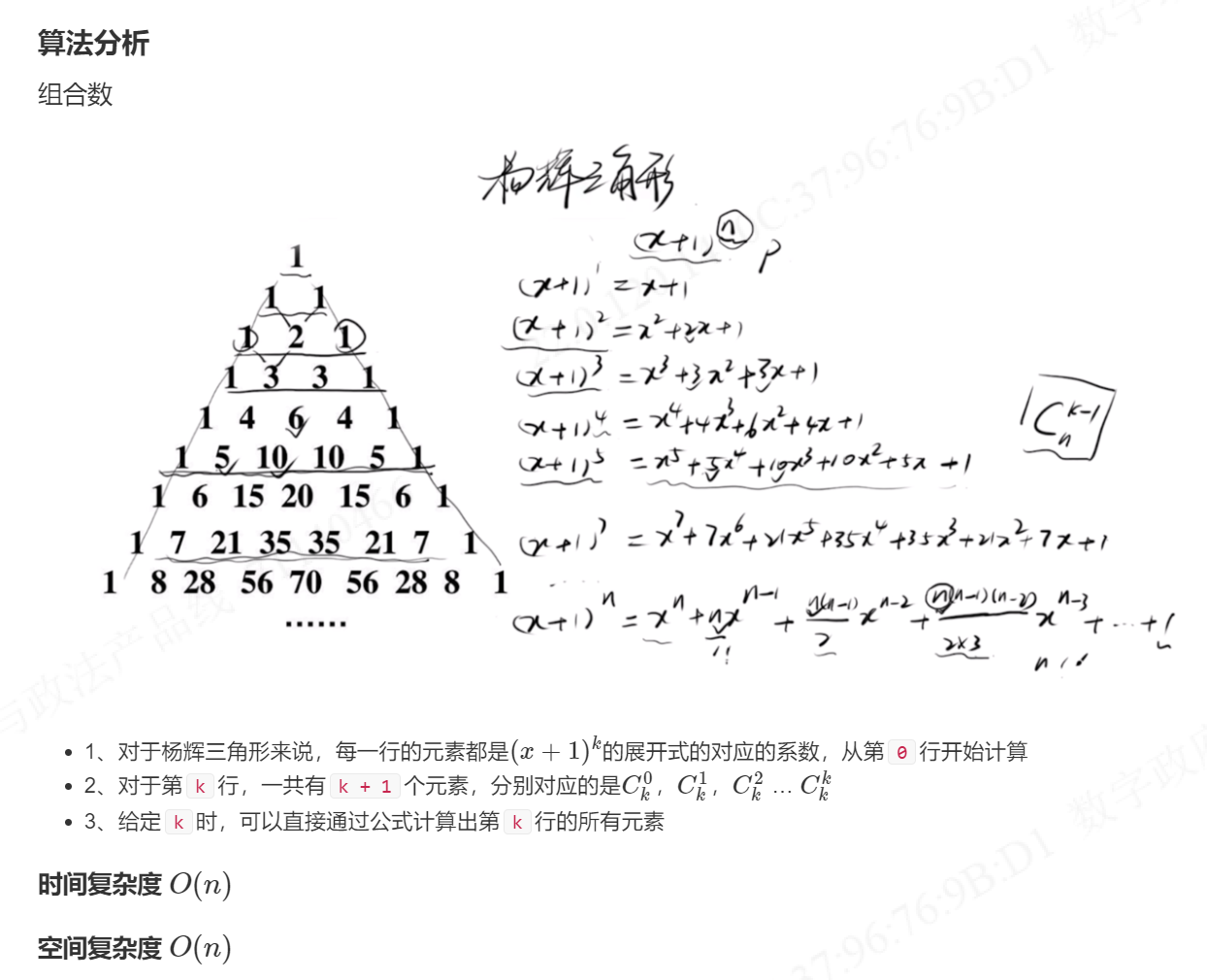
**119.Pascal's Triangle II**

使用解法二更好，时间复杂度更优，且空间复杂度的常数项更小。

解法一：

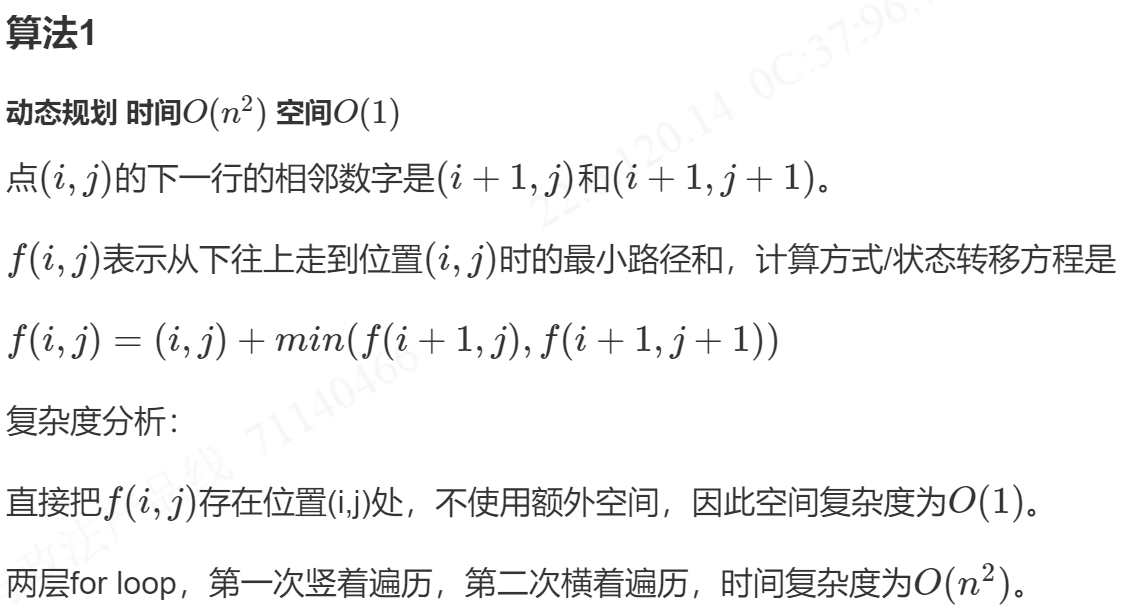


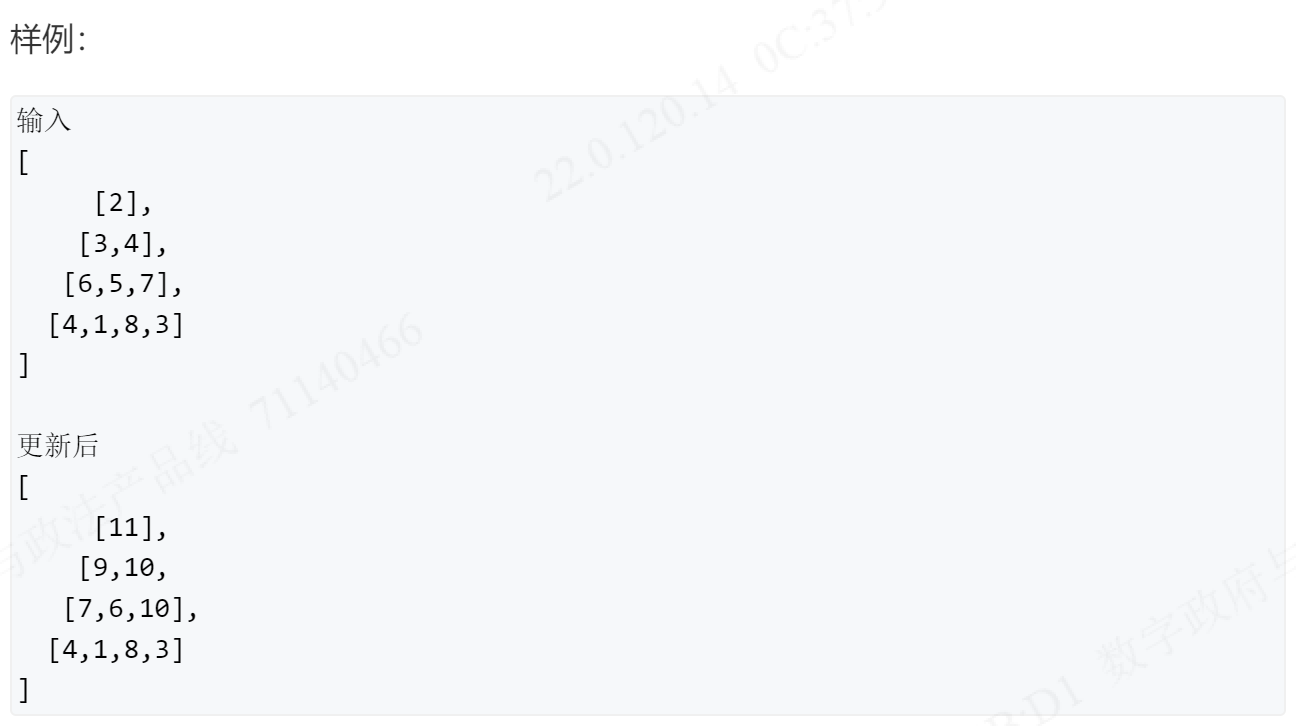
解法二：



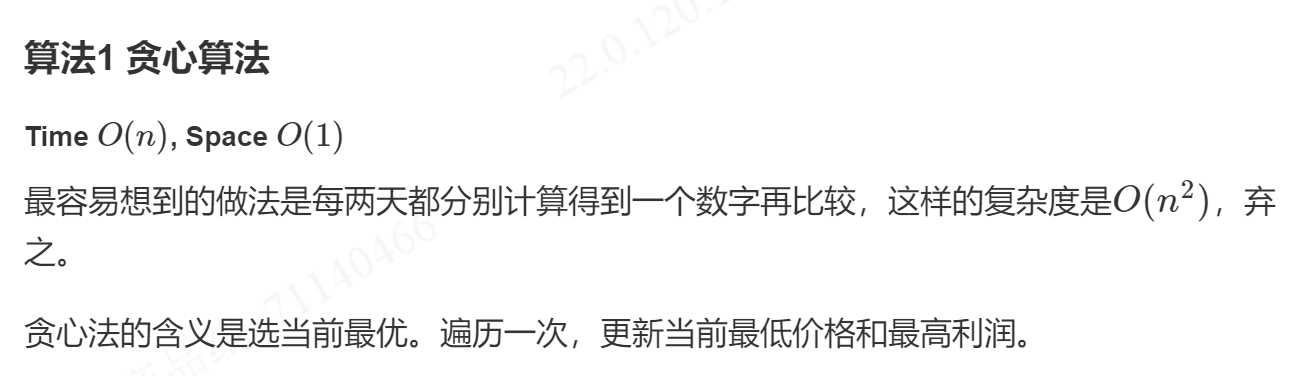
**120.Triangle**





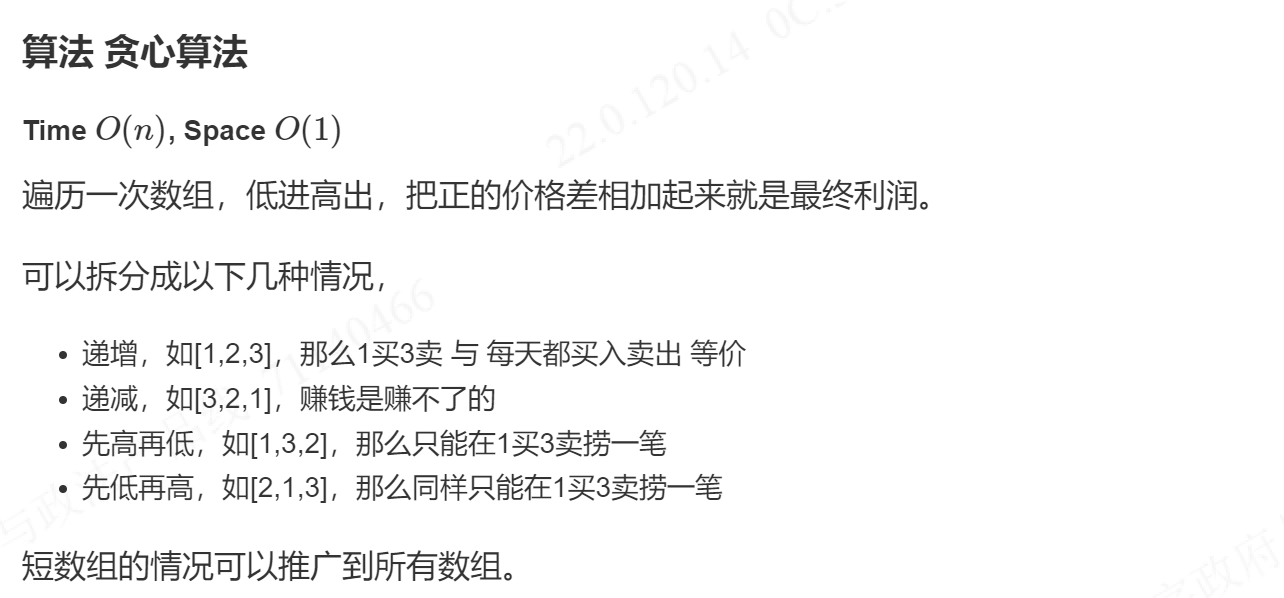


**121.Best Time to Buy and Sell Stock**



实际上是一种dp的思想，详见代码。

**122. Best Time to Buy and Sell Stock II**



该算法正确性用**微扰法**证明，见链接：<https://leetcode.cn/circle/article/YPuyhz/>

贪心算法的正确性证明方法见《算法竞赛进阶指南》0x07小节

P.S. 实际上并没有去看该算法正确性的证明...没心情...但还是先记录在这里相关的证明方法在哪里可以找到。

**123. Best Time to Buy and Sell Stock III**

注意该题目不能用贪心法，**贪心法是错误且无效的解法**。像122题那样用贪心法计算会出现错误。其中一个反例为：

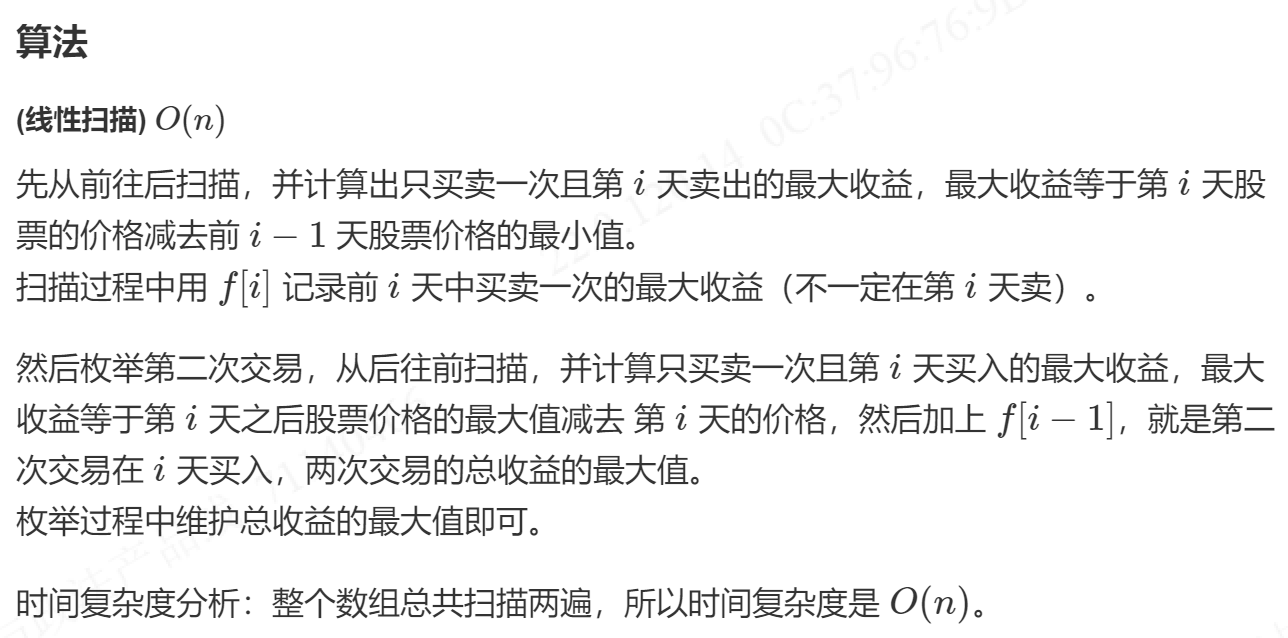
[1,2,4,2,5,7,2,4,9,0]

若采用122题中方法，取两个差值最大的上升区间，则这两个区间分别为2,5,7和2,4,9，则ans=(7-2)+(9-2)=12

但实际上，ans=(7-1)+(9-2)=13

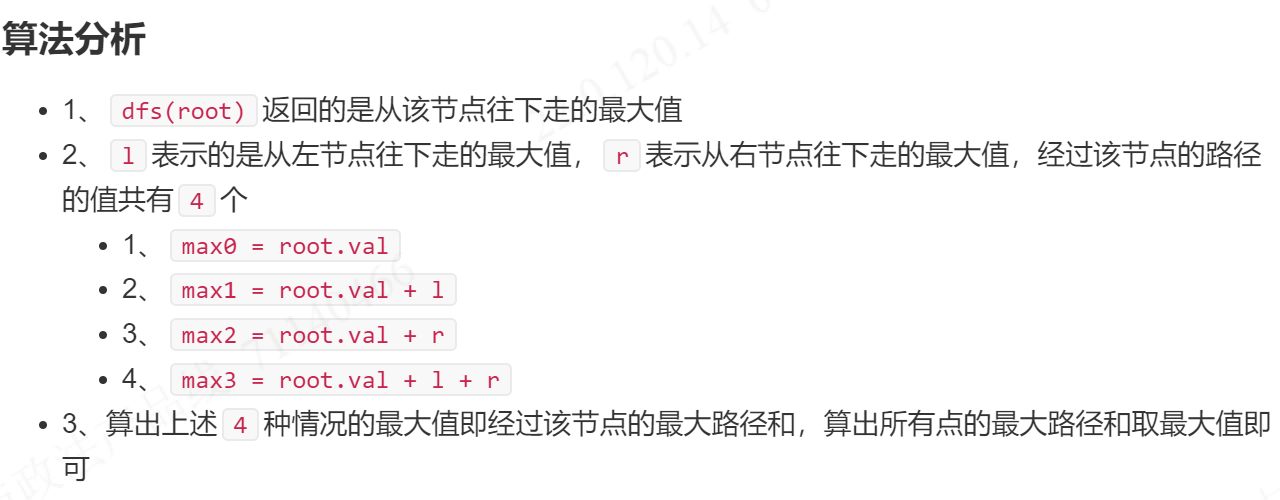
**正确解法：动态规划**

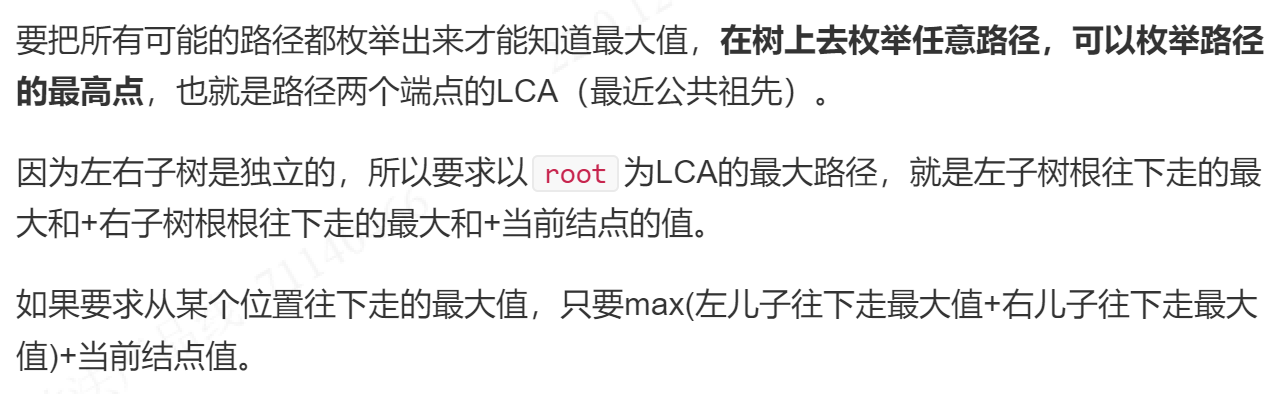




和121题的解法类似。只不过是需要前后扫描两次。

**124. Binary Tree Maximum Path Sum**

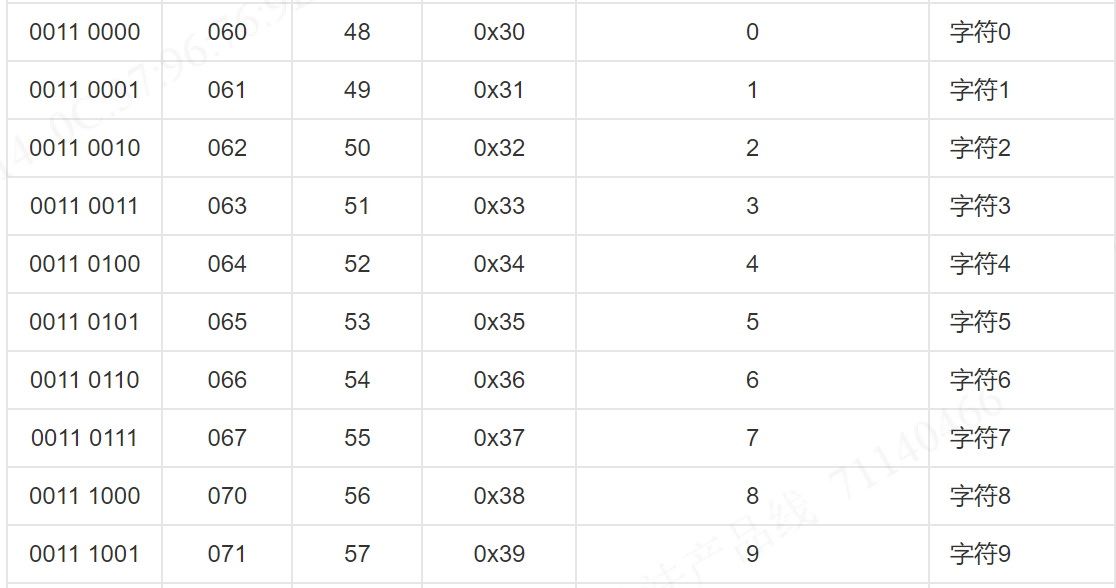


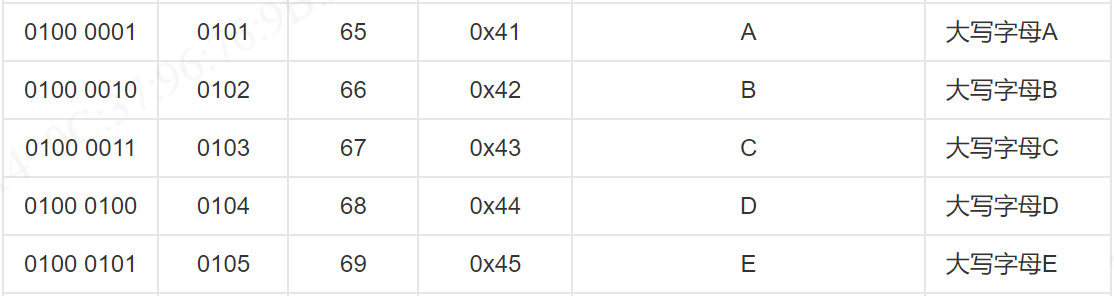


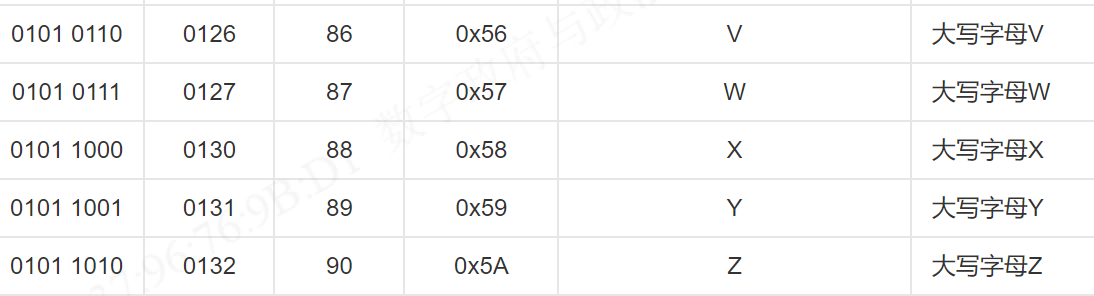
**125. Valid Palindrome**

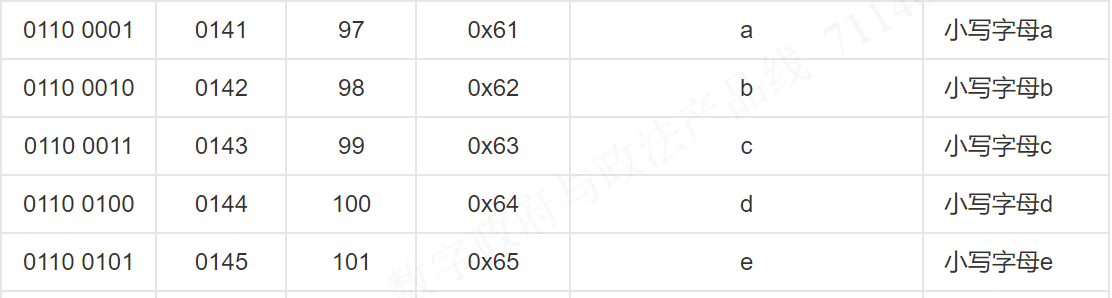
需要了解ASCII表：

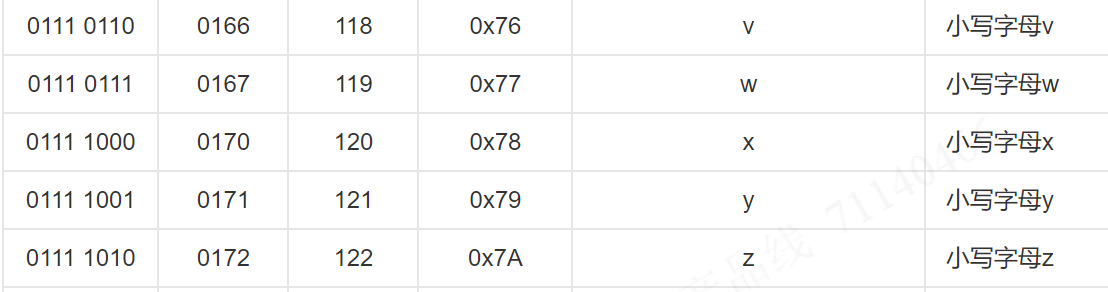


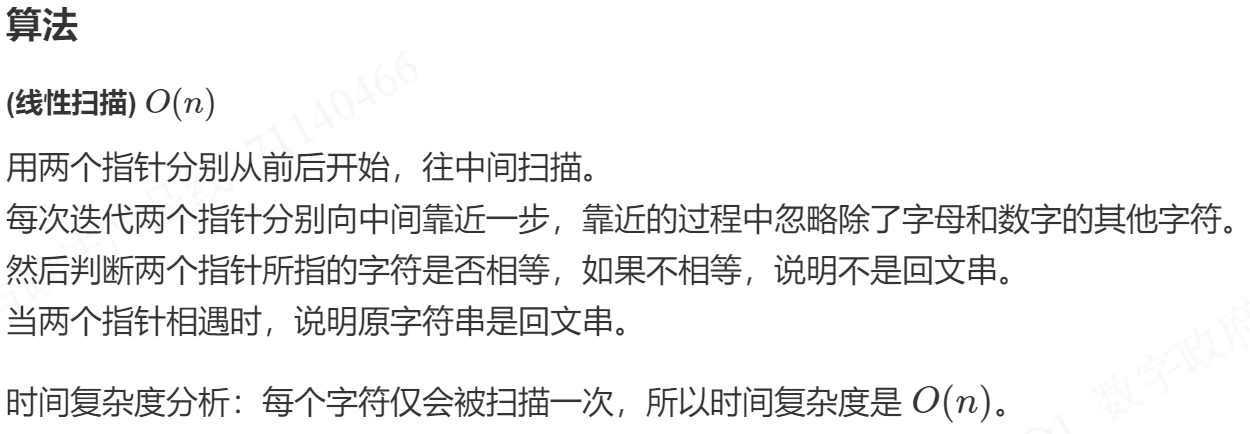












实现过程中应注意好边界条件。此外，大写字母和小写字母的转换可以通过异或32实现，这和大小写字母的ASCII码的二进制表示有关。在相同字母的大小写的二进制表示里，只有第5位不同，异或32可以起到只改变第5位的效果，从而实现大小写的转化。

**126. Word Ladder II**

本质是图论中的最短路问题。

设n=wordList.length，L=wordList[i].length。有两种建图方式：

①单词两两比较，判断由一个单词能否变成另一个单词。每次需要比较O(L)次，一共需要比较O(n\*n)次，所以这种方式的复杂度为O(n\*n\*L)

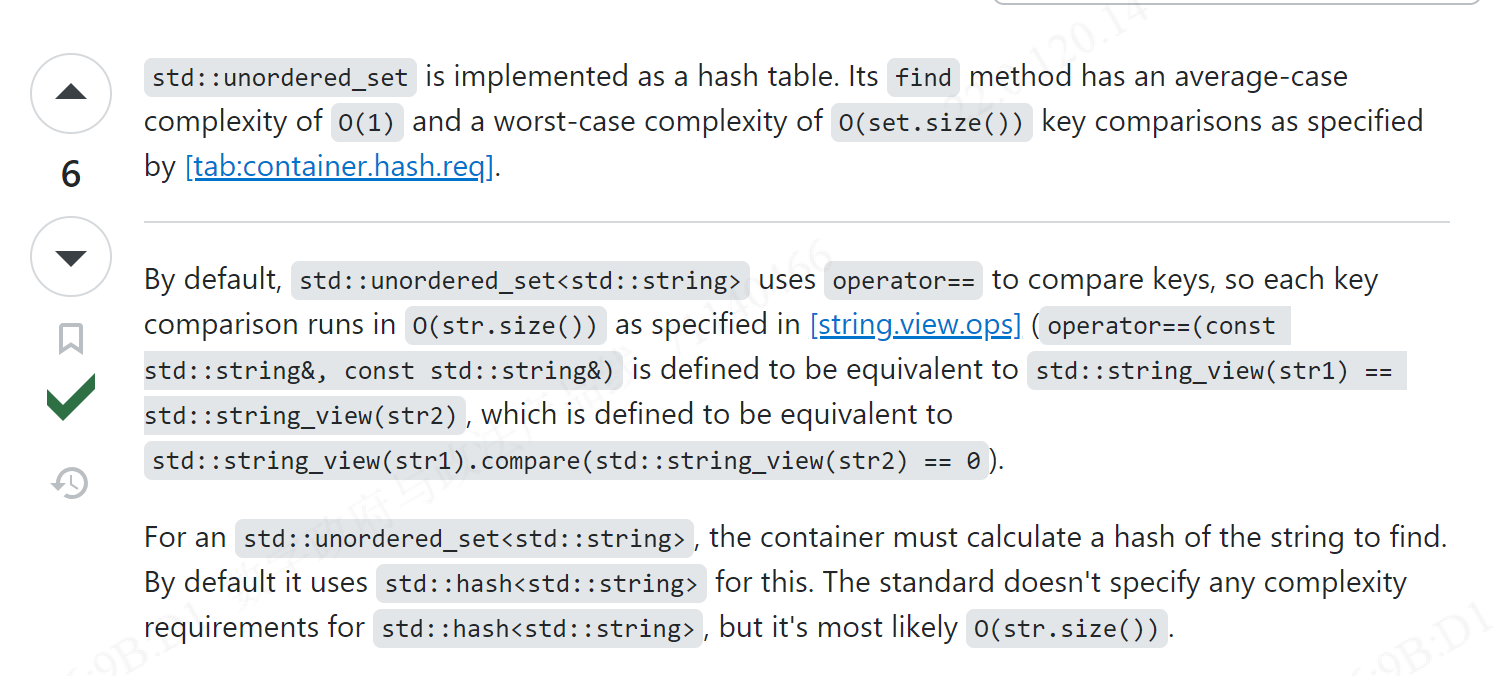
②枚举每个单词的每个字母(\*nL) 再尝试将各字母变成别的字母(\*26) 再判断变化后的单词是否在列表中出现过(将wordList中的单词存在哈希表中)(\* L) -> O(26\*n\*L\*L)

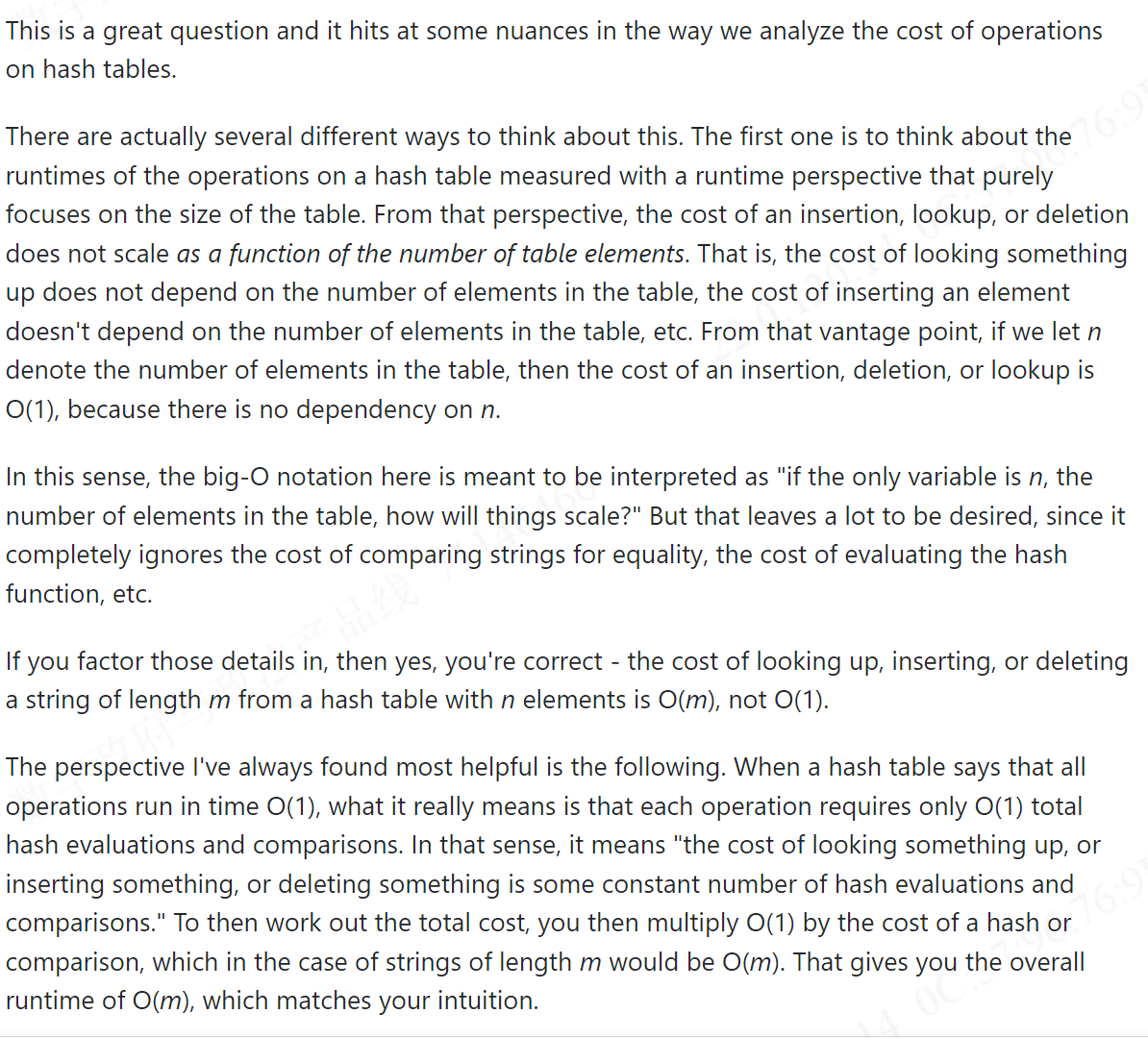
注意：在C++中，判断一个长度为L的string是否在unordered\_set中，所需的时间为O(L),对此的分析见下面两个链接和截图。

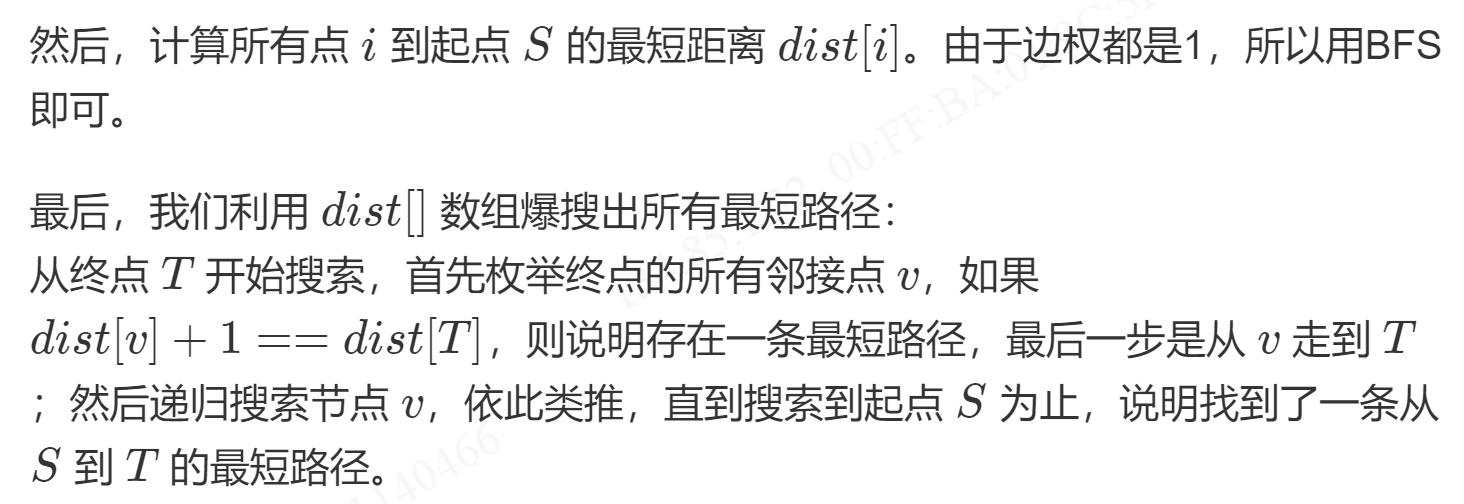
对于本题给定的数据范围，采用第二种方式建图花费的时间少。

<https://stackoverflow.com/questions/61284351/time-complexity-of-finding-a-string-in-an-unordered-set-of-strings>

<https://stackoverflow.com/questions/72324033/c-unordered-set-string-hash-time-complexity>







**127. Word Ladder**

与126题相同，比126题简单，略。

**128.Longest Consecutive Sequence**

考察哈希表的灵活使用。

**思路**：

首先将所有数字放入哈希表，遍历哈希表中的元素，因为要找连续的数字序列，因此可以通过向后枚举相邻的数字（即不断加一），判断后面一个数字是否在哈希表中即可。

为了保证O(n)的复杂度，避免重复枚举序列，因此只对序列的起始数字向后枚举（例如[1,2,3,4]，只对1枚举，2，3，4时跳过），因此需要判断一下当前遍历元素num是否是序列的起始数字（即判断一下num-1是否在哈希表中）。

**C++实现说明**：

1.unordered\_set存储的各个元素的值都互不相等。

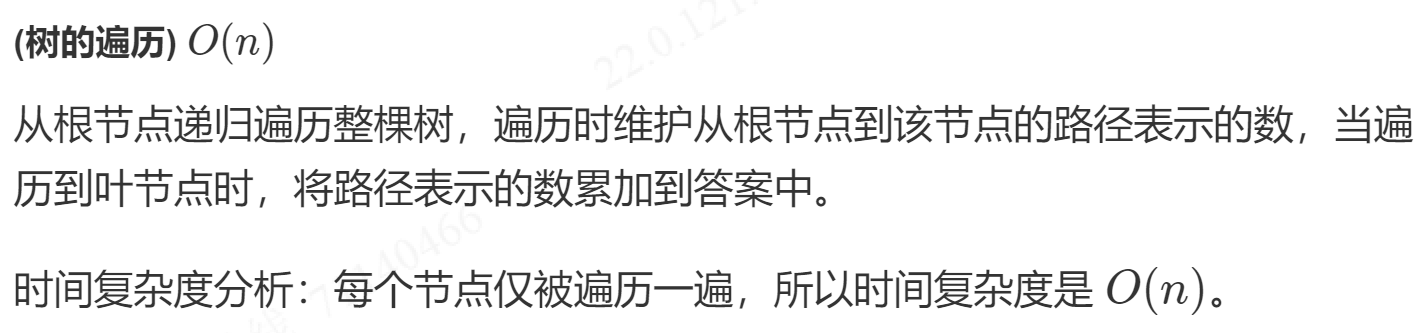
2.unordered\_set类模板成员方法：

begin() 返回指向容器中第一个元素的正向迭代器。

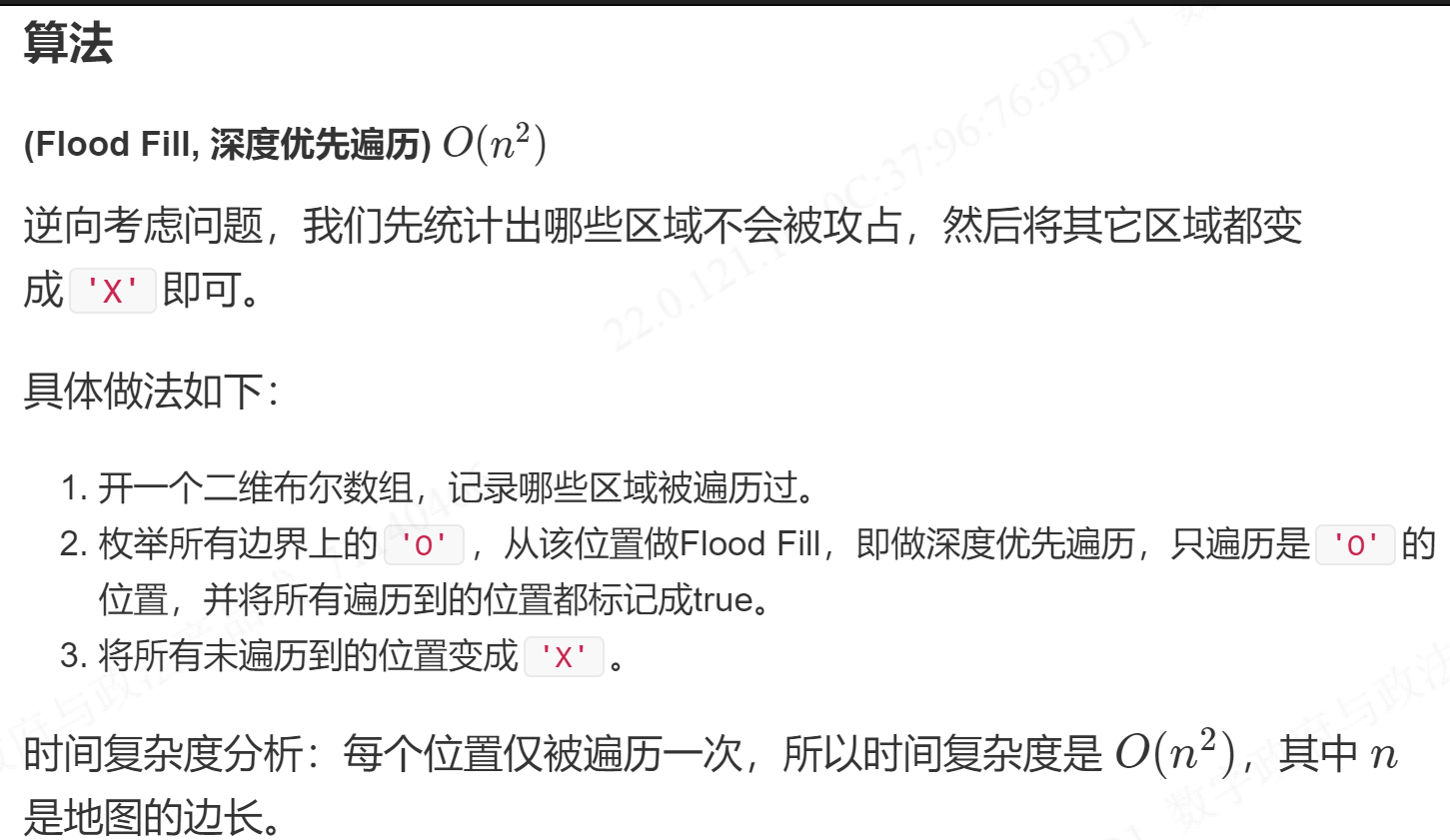
end() 返回指向容器中最后一个元素之后位置的正向迭代器。

find(key) 查找值为key的元素，如果找到，则返回一个指向该元素的正向迭代器；反之，则返回一个指向容器中最后一个元素之后位置的迭代器（即end()方法返回的迭代器）。

**129.Sum Root to Leaf Numbers**



**130.Surrounded Regions**



利用泛洪算法解决该问题。该问题的关键是注意到一个性质：

从边界的O开始，包含该O的连通分量不会被X包围，也就是不会被攻占，而其它包含O的连通分量将会被X包围，即会被攻占。

因此利用上图中算法解决问题即可。