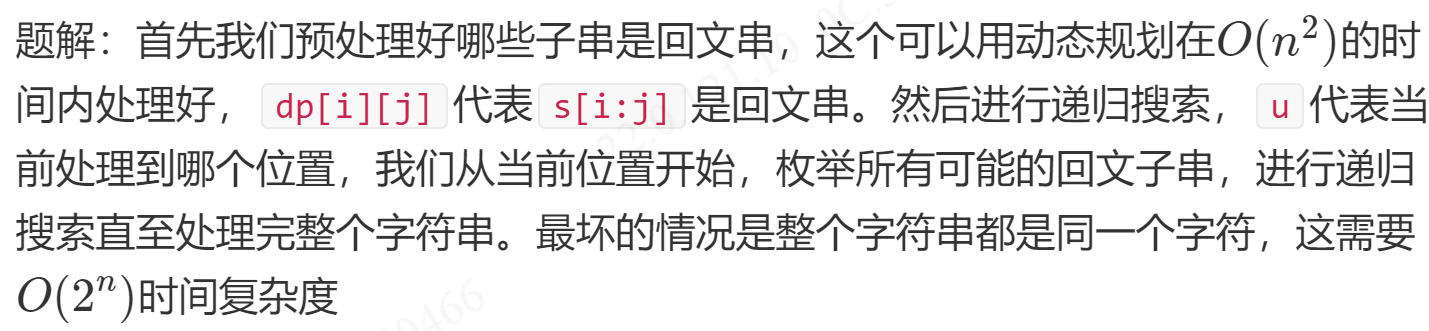
**131. Palindrome Partitioning**

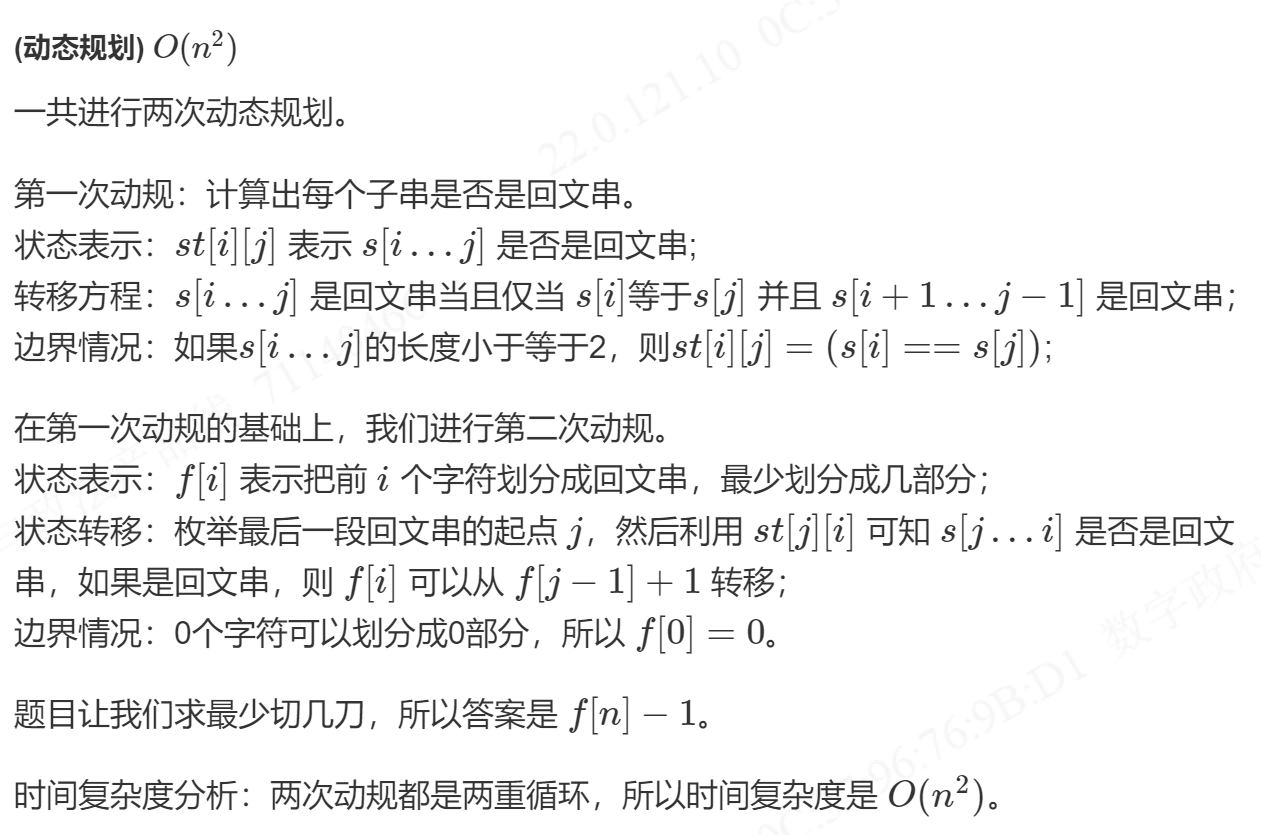


注意下C++的一个语法：

s.substr(pos, len): pos表示要截取的子字符串在字符串s中的起始位置，len表示要截取的子字符串的长度。

**132.Palindrome Partitioning II**

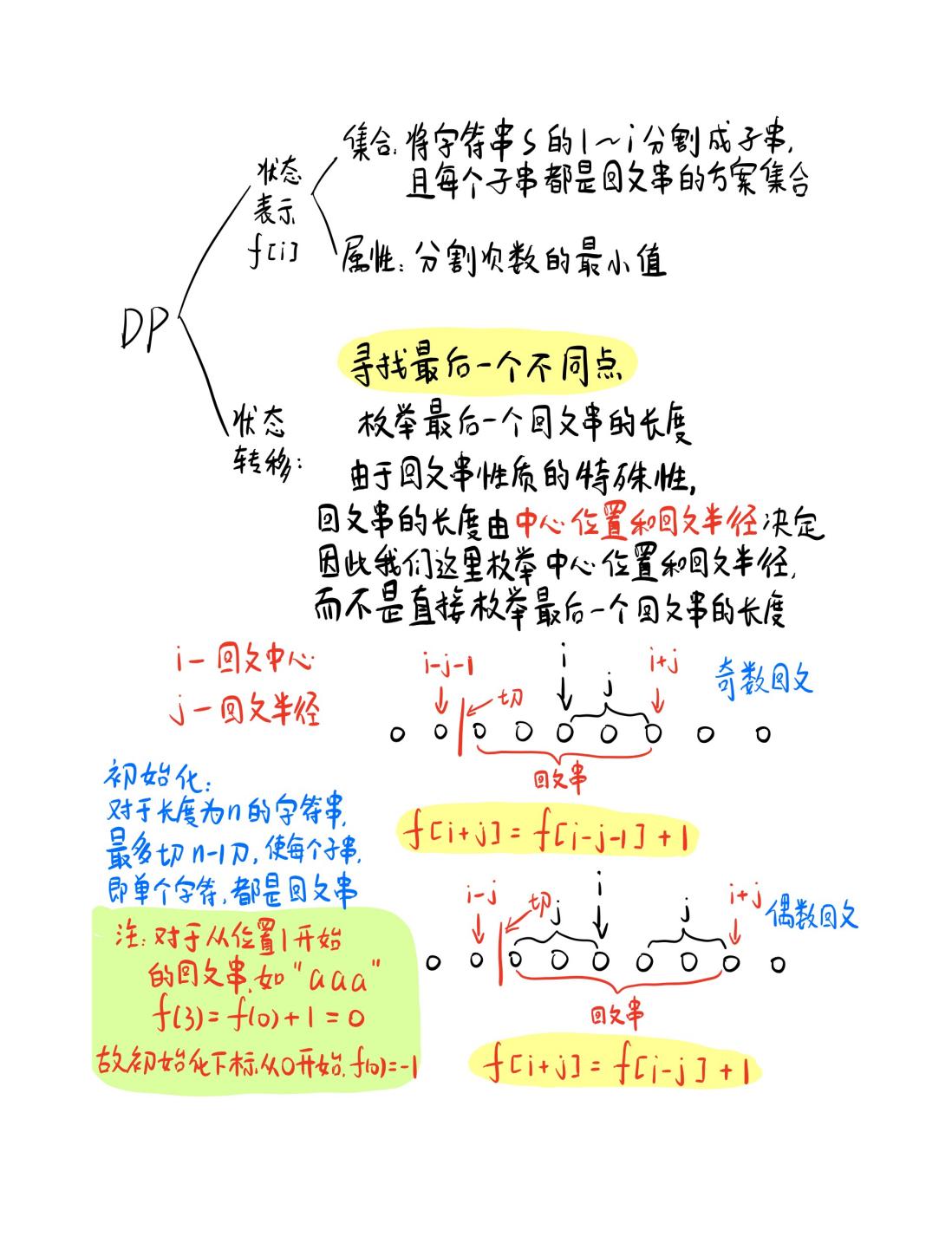
**解法一：**



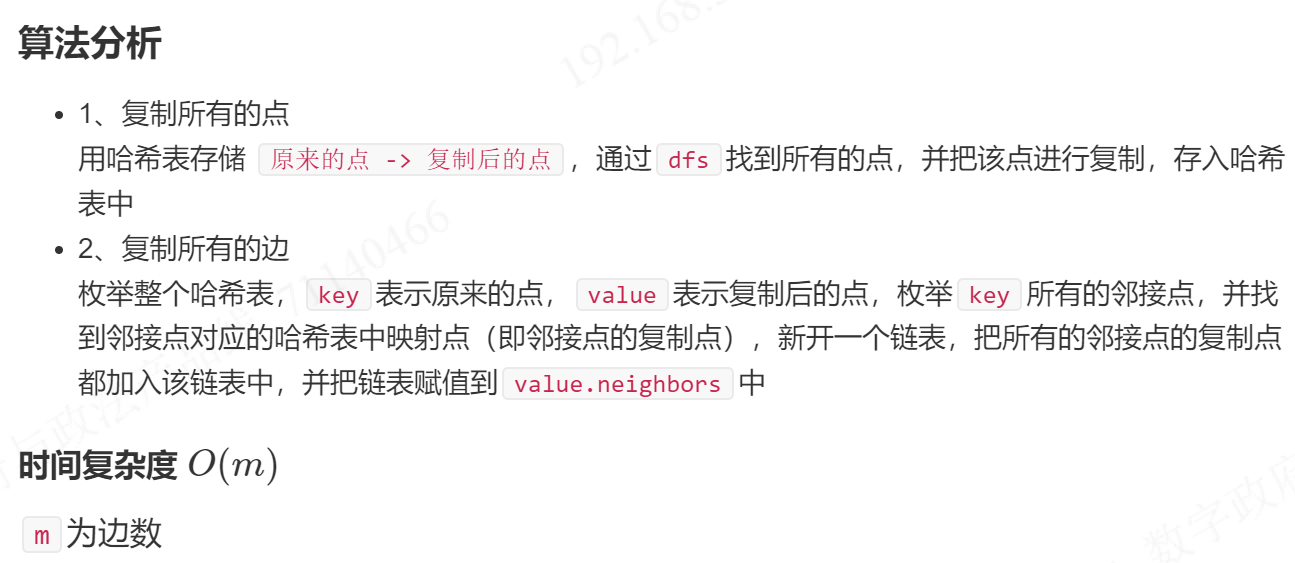
**解法一**如上。进行两次动态规划即可。此方法在第一次动归时开的空间比较多。

**解法二：**

与解法一相比，利用的空间更少。



**133.Clone Graph**



C++实现语法细节：

1.unordered\_map

①count函数：由于unordered\_map不允许存储具有重复键的元素，因此count()函数本质上检查unordered\_map中是否存在具有给定键的元素。如果Map中存在具有给定键的值，则此函数返回1，否则返回0。

②插入元素：可以用下标索引的形式插入元素，语法为unordered\_map[key] = value，如果键 key不存在，则插入键和对应的值，否则，会将键key的值更新为value。

③遍历：c++17中可以用**结构化绑定**的方式遍历哈希表，语法形式：auto [k,v]:map。

2.vector

void push\_back(const T& x):向量尾部增加一个元素x

**134.Gas Station**

使用**贪心(双指针)**算法，时间复杂度为O(n)，空间复杂度为O(1).

利用如下结论：

如果从x最远能到y点，y + 1点到不了，那么从x到y的任一点k出发都不可能到达y+1。

证明:

因为从k点出发的话，相当于从0开始加油，而如果从x出发到k点的油量一定是>=0的，即还有剩余的油或正好没油。既然在k点时油量>=0都到不了y+1，那么从0开始就更不可能到达y+1了。

**135.Candy**

**方法一**：两次遍历，时间复杂度O(n)，空间复杂度O(n)。

首先从左向右遍历，如果当前孩子评分比左边的高，将当前孩子分的糖果数置为左边孩子的糖果数加1，经过此次遍历可以保证，对于每个孩子，如果他的评分比左边的孩子高，则他分的糖果比左边的孩子多。

然后从右向左遍历，如果当前孩子评分比右边的高，先计算右边孩子糖果数加1后的数值，然后和当前孩子原来的糖果数比较，取较大值即可。

在分析完算法2后，可以看出，方法一相当于将递减序列利用从右向左遍历的方法，归结为一个递增的序列。

**方法二**：一次遍历，时间复杂度O(n)，空间复杂度O(1)。

首先观察各种情况下孩子分得的糖果数，得出结论(考察下面图中的前六种情况即可，后面重复了)：

1.如果某一段的评分是严格单调递增的，则孩子分得的糖果数严格单调递增，且山脚处分得的糖果数为1。

2.如果某一段的评分是严格单调递减的，则孩子分得的糖果数严格单调递减，且山脚处分得的糖果数为1。

3.如果先严格单调递增，紧接着就严格单调递减，则山峰处的糖果数取决于递增和递减中，长度较长的那个。

4.如果某几个相邻的孩子评分相同，可以分以下几种情况：

i)严格单调递增段和平缓段的连接点，归入情况1

ii)平缓段和严格单调递增段的连接点，分得的糖果数应为1

iii)平缓段和严格单调递减段的连接点，归入情况2

iv)平缓段内的点(即不是连接点)，分得的糖果数应为1

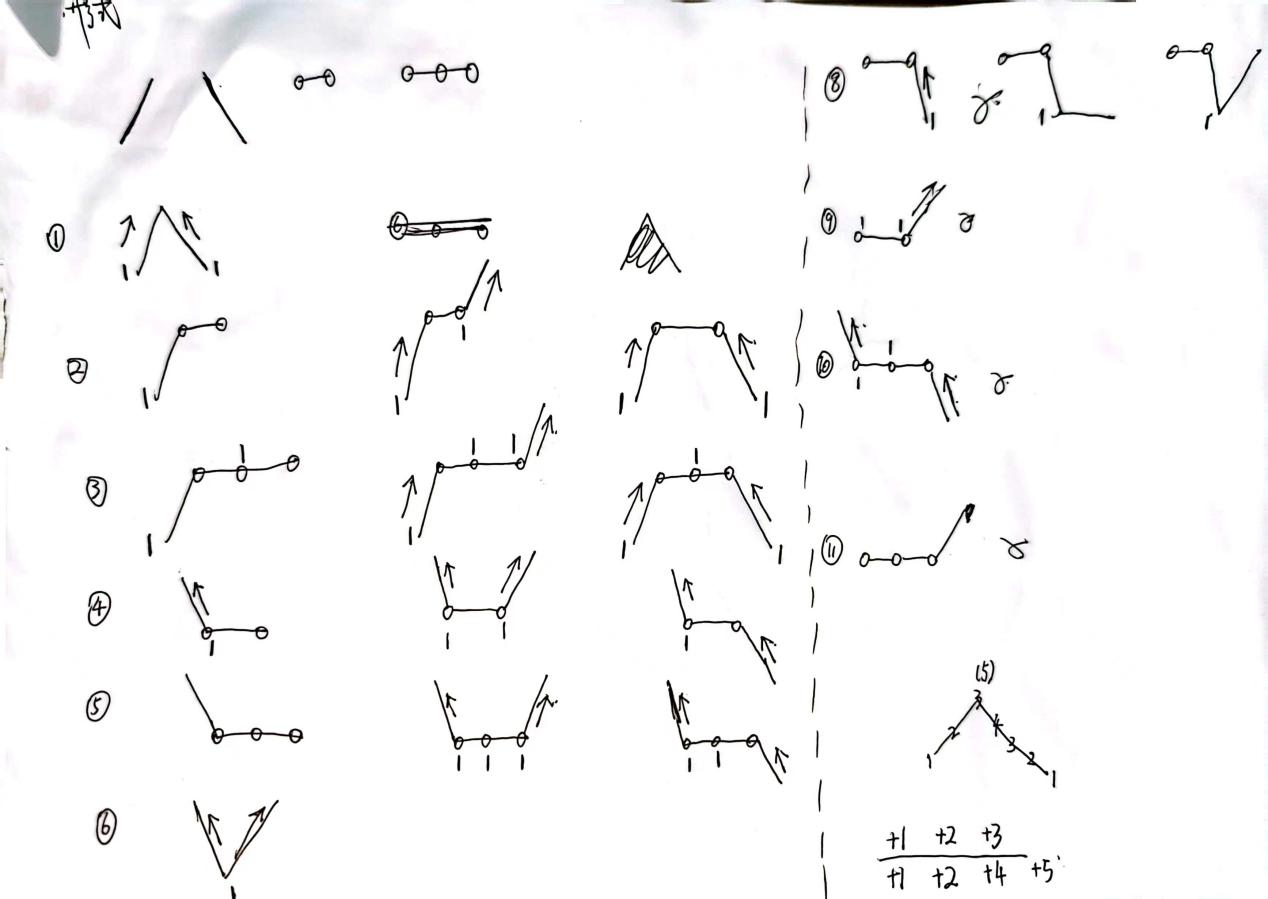
v)严格单调递减段和平缓段的连接点，分得的糖果数应为1

在具体实现时，从左向右进行一次遍历，实现以上规则：

1.如果是一个递增的点，其分配的糖果数只需要简单的比前一个点多1即可。

2.如果是一个递减的点，在实现时，其分配的糖果数为当前递减段的长度。此时还需要考虑结论3，也就是说，如果递减段的长度大于递增段的长度，需要对糖果数进行修正。举例来说(下面图片右下角就是这个例子)，如果评分序列是1,3,5,4,3,2,1，则实际分配的糖果数应为1,2,5,4,3,2,1;而程序运行时，首先根据实现规则1，分配糖果数为1,2,3,此时可得递增序列长度为3；然后根据规则2，接下来的孩子分配糖果数为1,2，再接下来，递减序列的长度将超过递增序列，而3个糖果在之前的递增序列中已经算过，因此直接跳过3，为接下来的孩子分配糖果数为4,5.

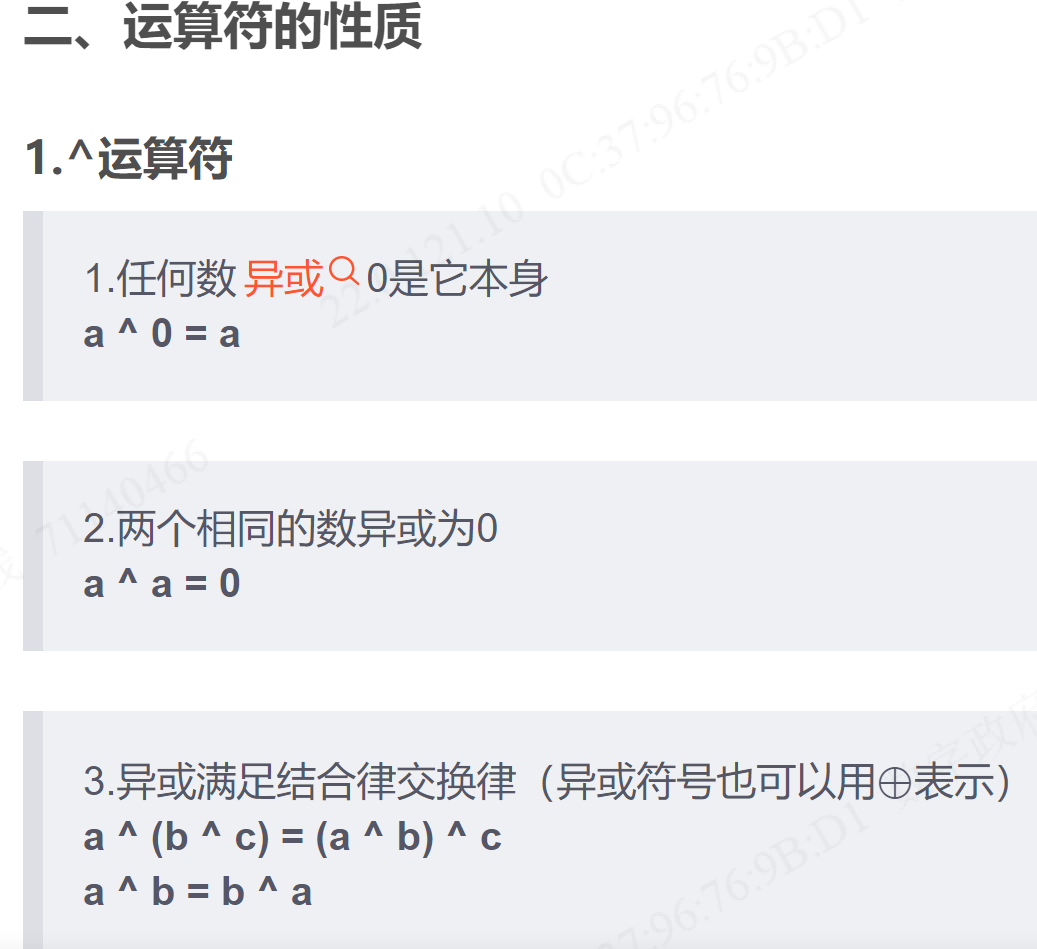
3.如果是一个平缓的点，为其分配糖果数为1即可。



**136.Single Number**



利用位运算即可解决该问题，另外注意位运算具有以下性质：

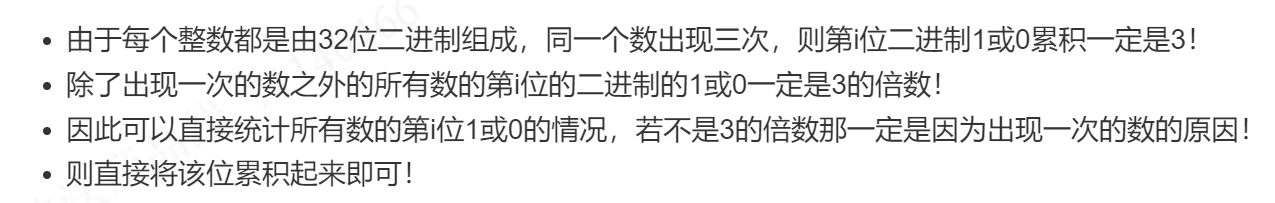


**137.Single Number II**

有两种方法，但这两种方法其实思路相同。

方法一：

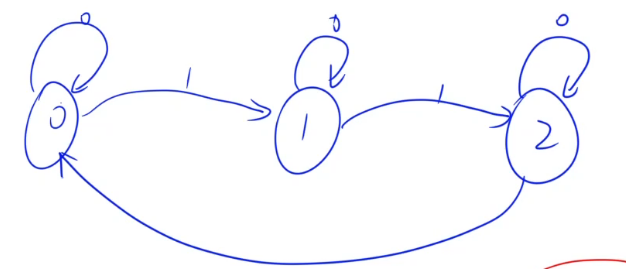
首先，由题目给出的限制-231 <= nums[i] <= 231 - 1，可知数字最长为32位。因此最朴素的思路如下所示。



方法二:

实际上思路和方法一相同，只是模拟了一个状态机，减少了循环的次数。针对所有数的二进制表示的第i位之和对3取余后是几这一步，可以做如下优化。

利用状态机(十进制)表示，就是当第i位之和是3的倍数时，就将和重置为0，这样当整个数组遍历完时，该位留下的二进制数字(0或1)就是落单那个数的二进制表示。



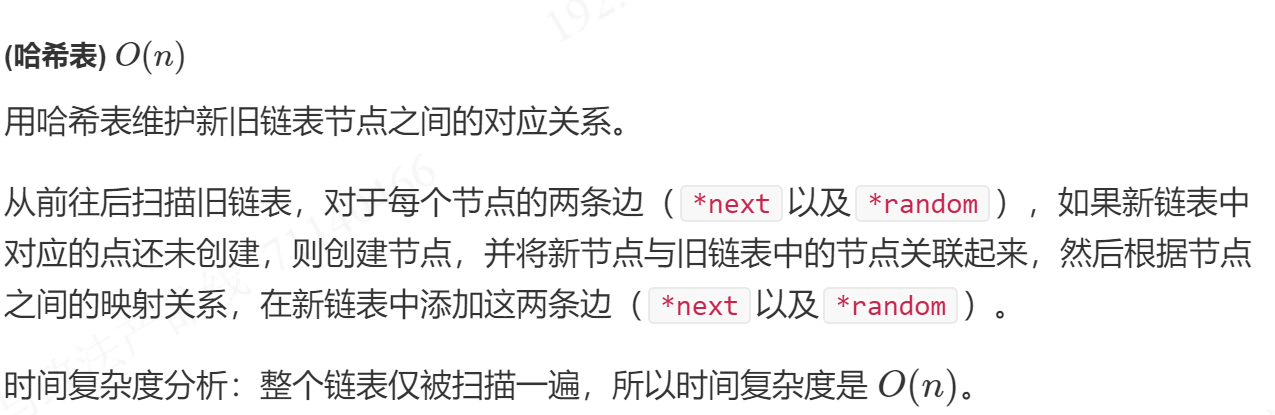
为了利用二进制计算，我们需要两个标志位one和two，用于表示上面状态机中的状态(圈中的数字)的二进制表示，而将状态转移用x表示，便可以得到如下判定表。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| two | one | x | -> | two | one |
| 0 | 0 | 0 |  | 0 | 0 |
| 0 | 0 | 1 |  | 0 | 1 |
| 0 | 1 | 0 |  | 0 | 1 |
| 0 | 1 | 1 |  | 1 | 0 |
| 1 | 0 | 0 |  | 1 | 0 |
| 1 | 0 | 1 |  | 0 | 0 |

而该判定表可以用如下两个步骤表示:one=(one^x)&(~two),two=(two^x)&(~one)，这样问题就得到了解决。

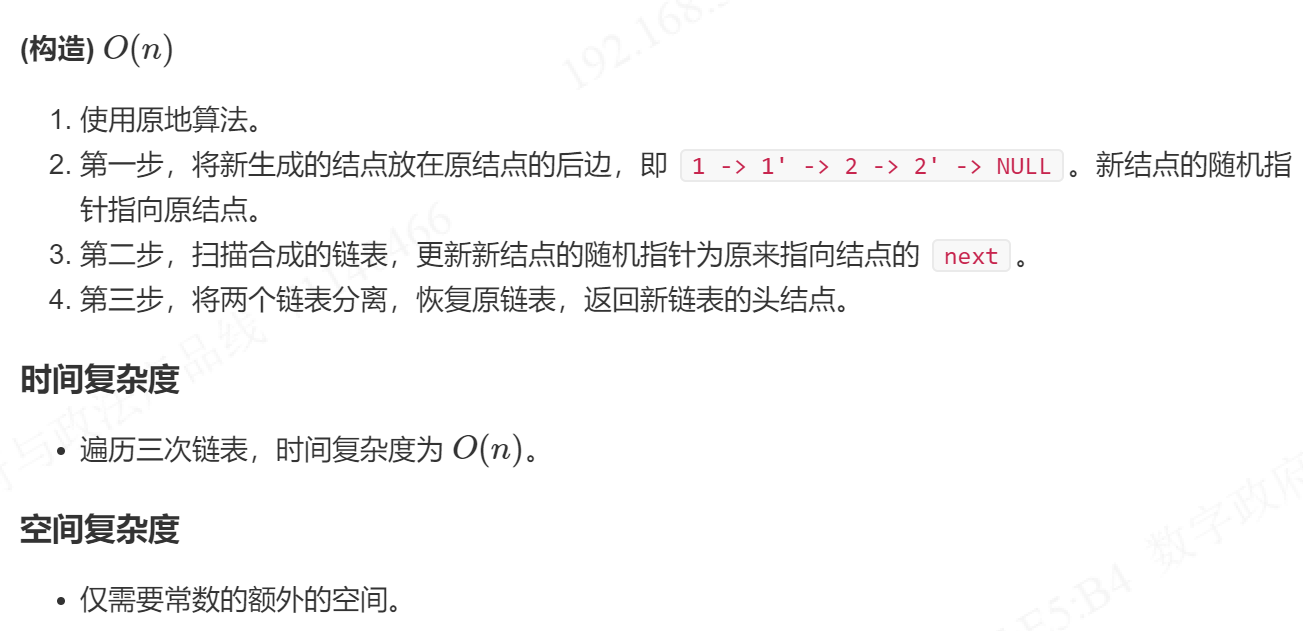
**138.Copy List with Random Pointer**

方法一：



上面方法的空间复杂度是O(n)。方法二可以实现O(1)的时间复杂度。

方法二：



**139.Word Break**

方法一：

使用动态规划的方法。

①函数f：f[i]表示s[0..i]是否可由wordDict中的词构成。

②转化规则：为了计算f[i]，将s[0..i]分割成两段s[0..i-j]和s[i-j+1..i]，可见第二段的长度为j。其中第二段被视为最后一个单词。如果第二段在wordDict中，并且第一段可由wordDict中的词构成(即f[i-j]为true)，则s[0..i]可由wordDict中的词构成。

注意到1 <= wordDict[i].length <= 20，因此可以进行剪枝。

注意，①substr(pos,len)返回从pos开始，长度为len的字符串，时间复杂度为O(len)。

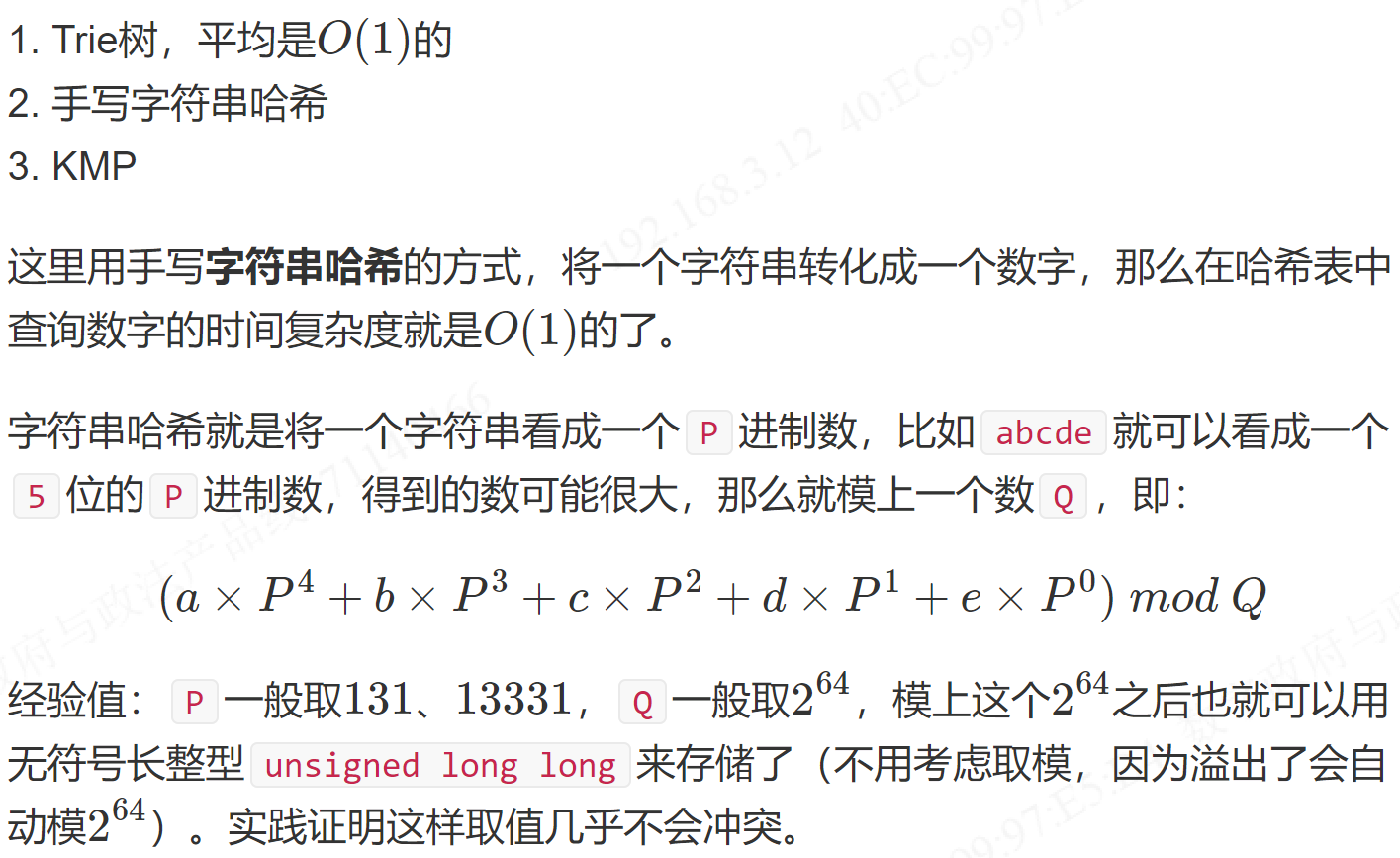
②unordered\_set插入一个字符串的时间复杂度也是O(len)。

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

方法二：

整体思路仍然是方法一的思路，考虑如何对substr这一步进行优化。

考虑优化一下查询字符串是否在字典中这件事，有几种方案：

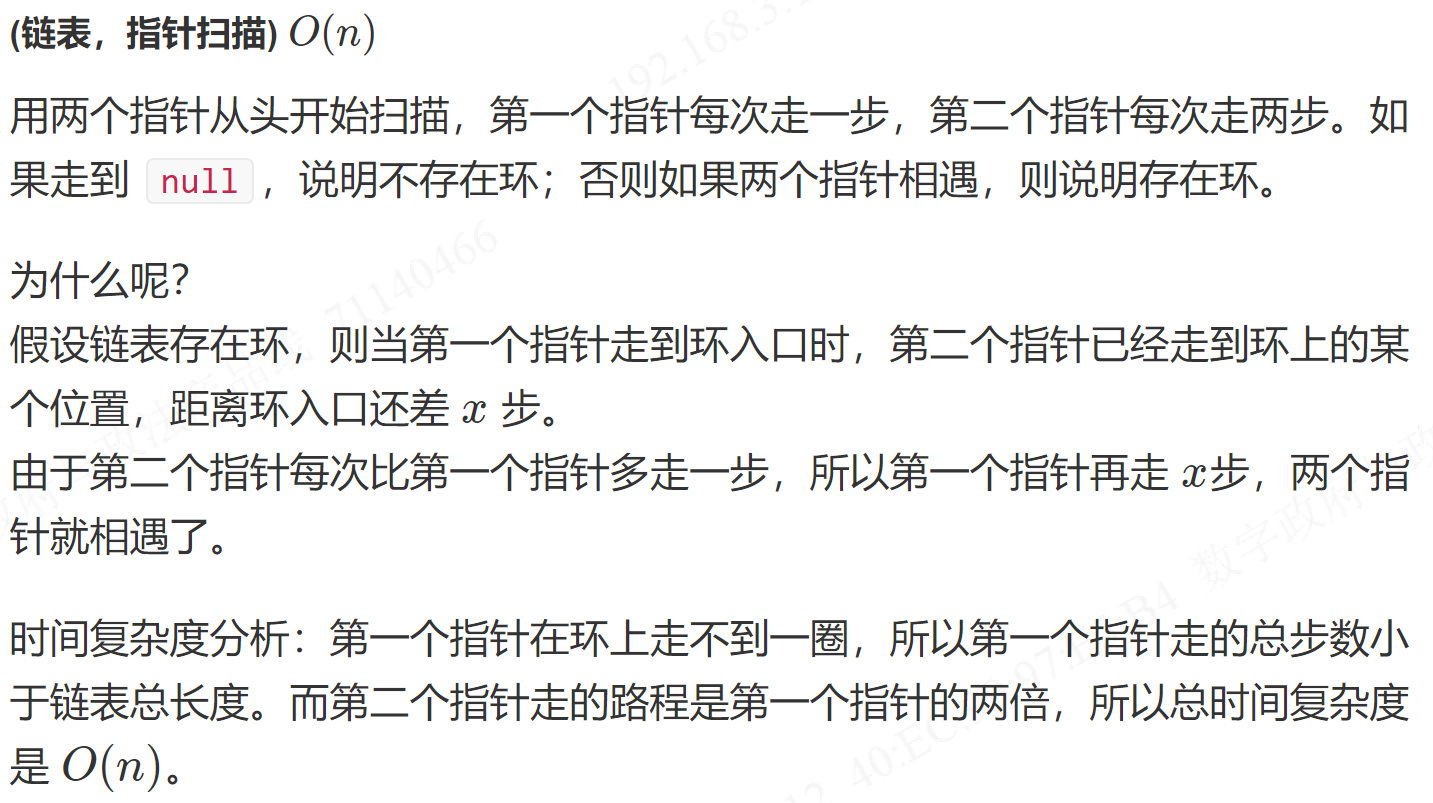


**140.Word Break II**

利用dfs解决该问题，注意需要根据题目给的数据范围进行剪枝。如果想进一步剪枝，可以像139题一样先算出f数组，以判断从某一位置继续dfs是否是必要的。

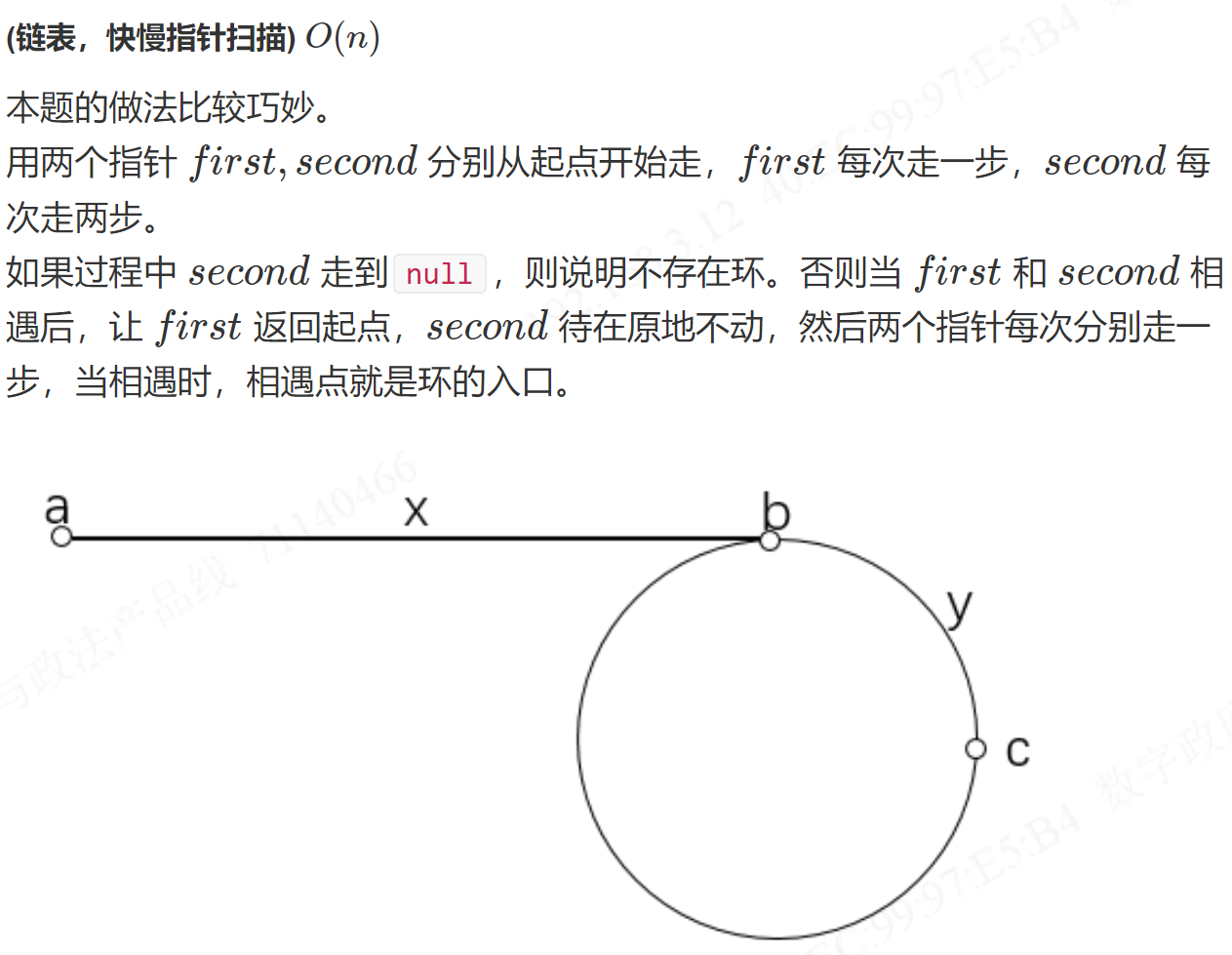
**141.Linked List Cycle**

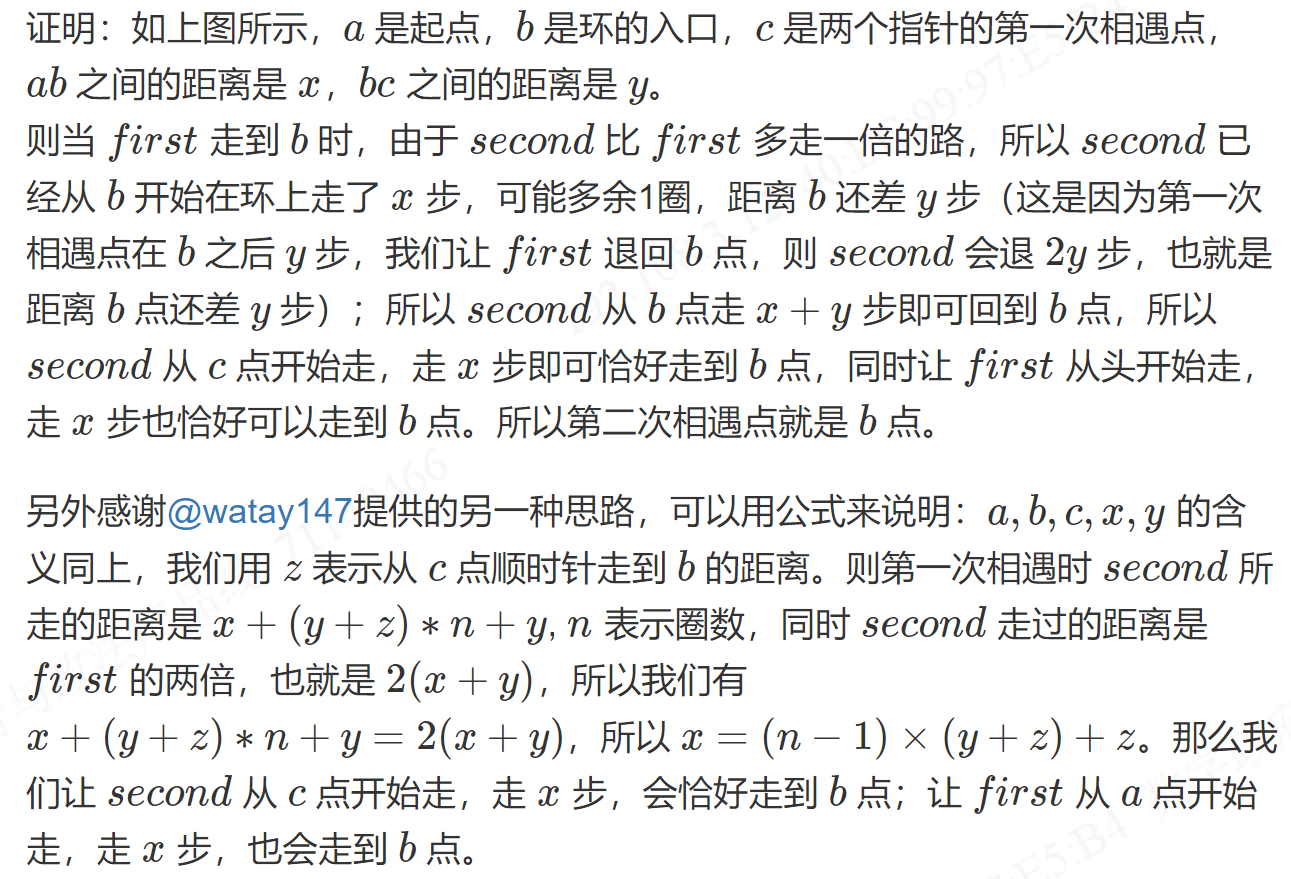
使用快慢指针解决该问题，有环时相当于追及问题。

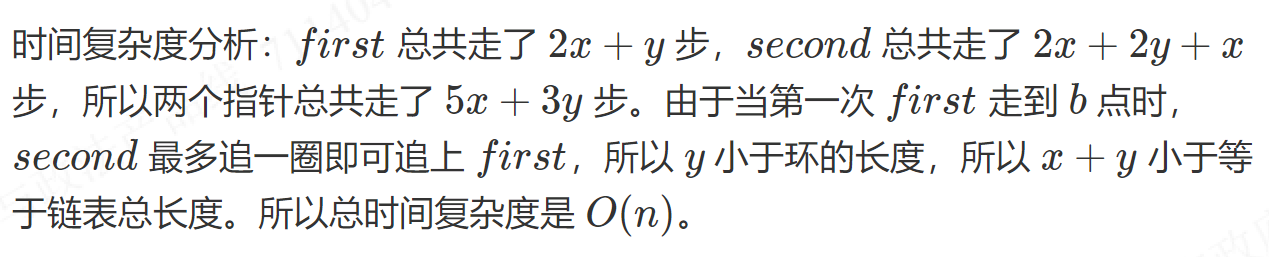


**142.Linked List Cycle II**

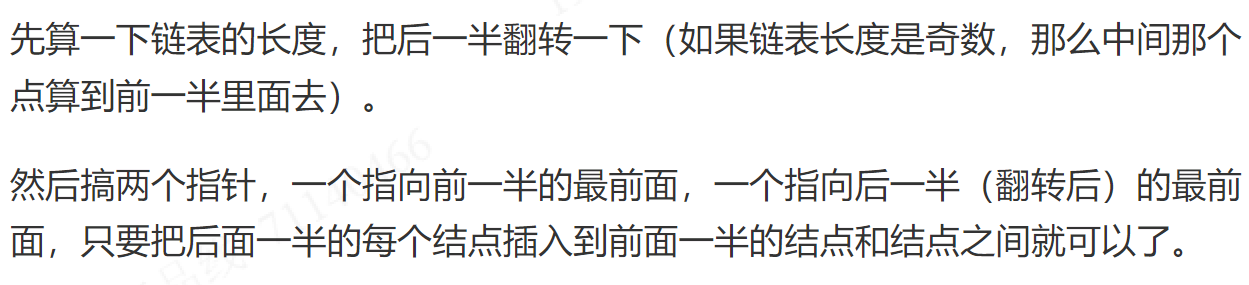
证明直接看公式法的证明即可，前面口述的证明不需要看。注意第一次快慢指针相遇相当于追及问题，在环中，fast指针可能走多圈，而slow指针走不到一圈。







**143.Reorder List**



注意，求链表长度是为了找到后半段，当然也可以不求链表长度，而是用快慢指针找到链表的中间节点。快慢指针代码类似于下面：

