例题分析一

LeetCode 第 336 题，回文对：给定一组唯一的单词， 找出所有不同的索引对（i, j），使得列表中的两个单词， words[i] + words[j] ，可拼接成回文串。

示例 1

输入: ["abcd","dcba","lls","s","sssll"]

输出: [[0,1],[1,0],[3,2],[2,4]]

解释: 可拼接成的回文串为 ["dcbaabcd","abcddcba","slls","llssssll"]

示例 2

输入: ["bat","tab","cat"]

输出: [[0,1],[1,0]]

解释: 可拼接成的回文串为 ["battab","tabbat"]

解题思路：暴力法

所谓回文，就是正读和反读都一样的字符串，例如"leetteel"。

检查一个字符串是否是回文，方法如下。

1. 将给定的字符串翻转，然后跟原字符串对比，看是否相等。但空间复杂度为 O(n) 。

2. 定义两个指针 i、j，一个指向字符串的头，一个指向字符串的尾巴，同时从两头进行检查，一旦发现不相等就表明不是回文，一直检查到两个指针相遇为止。

   将上述方法 2 用代码实现如下。

```
boolean isPalindrome(String word, int i, int j) {
    while (i < j) {
        if (word.charAt(i++) != word.charAt(j--)) return false;
    }

    return true;

}
```

   代码非常简单，因此不作过多讲解。

   回到例题本身，用暴力法怎么解。

   实现方法：

- 先找出所有的两两组合
- 对每种组合进行排查，看看哪种组合可以构成回文。

   时间复杂度：

- 假设一共有 n 个单词，每个单词的平均长度为 k，两两组合，有 P(n, 2) = n × (n - 1) 种；
- 对组合的字符串进行回文检查，需要 2k 的时间复杂度；
- 最终的时间复杂度是：O(n2×k)。

暴力法优化

暴力法需要检查哪些情况？

进行回文检查的时候，根据两个字符串的长度不同的程度，假设组合字符串的长度分别为 k1、k2，那么会出现以下三种情况。

- k1 = k2

  举例：字符串 s1 = "abcd"，字符串 s2 = "dcba"

  实现：同时从两边进行检查，看看它们能否构成回文，构成回文的条件就是两个指针相遇，或者同时扫描完两个字符串。

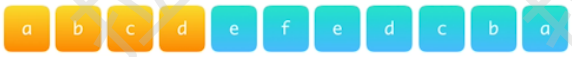
- k1 > k2

  举例：s1 = "abcdefe"，s2 = "dcba"

实现：同时从两头进行检查，由于 s2 的长度短，那么 s2 首先会被遍历完毕，此时 s1 还剩下的部分必须要满足回文。

- k1 < k2

  举例：s1 = "abcd"，s2 = "efedcba"



  实现：跟第二种情况类似，同时从两头进行检查，由于 s1 的长度短，s1 首先会被遍历完毕，此时 s2 还剩下的部分必须满足回文。

暴力法如何优化？

暴力法之所以那么慢，是因为它要对所有情况进行检查。而对于 s1 = "abcd" ，s1 + s2 的组合构成回文的一个条件就是，s2 的最后一个字符必须是 a，如果 k2>=2，它最后两个字符一定是 ba。不满足条件的字符串，不需要理会。

那么，如何能快速地知道哪些字符串以 a 结尾，哪些字符串以 ba 结尾呢？如果反看 s2 ，这个问题相当于，怎么能快速地找出所有以 a 开头或者以 ab 开头的字符串？第 2 节课里介绍的 Trie，正是快速查找以某个字符串开头的数据结构。

注意：此处要对每个字符串反着构建 Trie。

Trie

一个 Trie 一般都是由很多个 TrieNode 节点构成的，最普通的 TrieNode 节点一般有以下的结构。

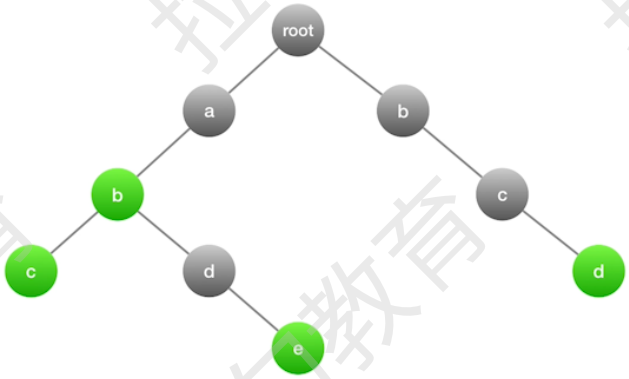
```
class TrieNode {
    boolean isEnd;
    HashMap<Character, TrieNode> children;

        TrieNode() {
        isEnd = false;
                children = new HashMap<>();
    }
}
```

其中，

- children：数组或者集合，罗列出每个分支当中包含的所有字符
- isEnd：布尔值，表示该节点是否为某字符串的结尾

由上可知，Trie 是一种通过字符链接起来的树状结构，且 Trie 一定有一个根节点 root，它的 children 集合包含了所有字符串的开头那个字符。

举例：给定一系列字符串："ab”，"abc”， “abde", “bcd"，用 Trie 表示的结构如下。

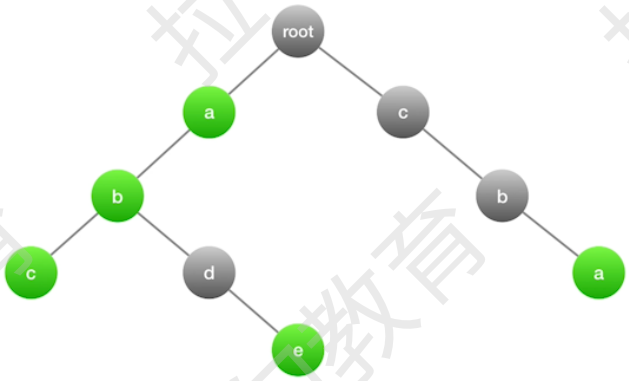


其中，

- 字符作为链接每个节点的边，这些字符也是哈希表里的 key。
- 这些 key 对应的 value 是节点，绿色的节点表示节点里的 isEnd 布尔值为 true，也就是这个节点表示了一个字符串的结束。
- 要利用这个 Trie 来查找所有以 b 字符开头的字符串时，可以避开左边三个以 a 字母开头的字符串。

构建 Tire

将给定的字符串变成 "ba" ，"cba" ，“edba"，“dcb"，它们其实就是之前的字符串的翻转。对它们逆序进行 Trie 的构建，也得出了相同的结构。为了能让给定的字符串能组合成回文，再添加两个字符串：” a “，” abc “，同时，把” dcb” 删除，Trie 变成了下面的结构。



就之前提到的三种情况来分析如何利用 Trie 判断合并两个字符串能否构成回文。基本上是同时遍历字符串和 Trie。

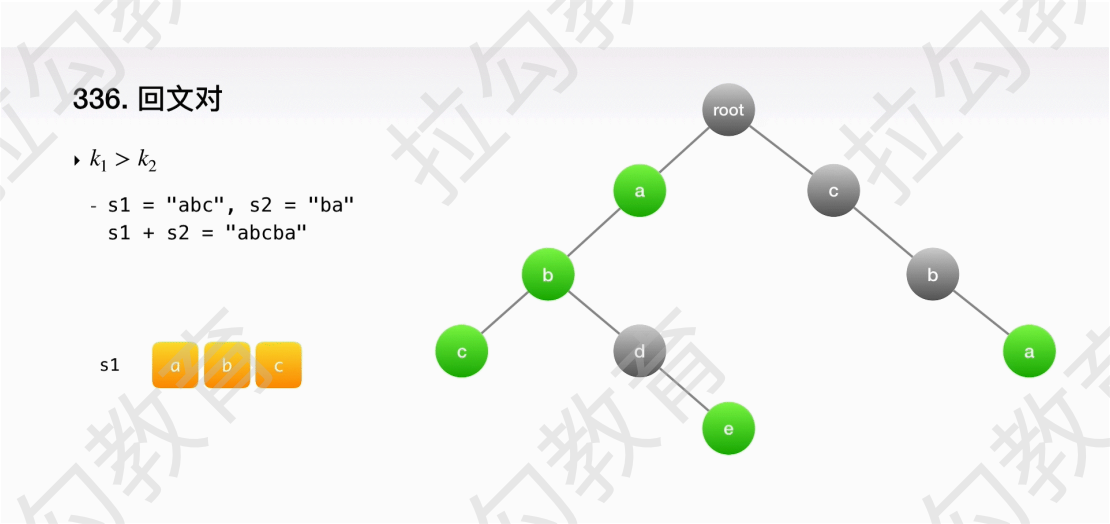
- k1 = k2，即两个字符串的长度相同并且能够构成回文

举例：s1 = “abc”，s2 = “cba”，s1+s2 = “abccba”。

1. 从 s1 的第一个字符 a 开始，Trie 里有记录以 a 结尾的字符串，其他那些不是以 a 结尾的字符串不予考虑。
2. 第二个字符 b，那么从 a 节点开始，看看有没有以 b 作为键值 key 的节点，有，继续。
3. 第三个字符 c，在 Trie 里，从 b 指向的节点开始，看看有没有以 c 作为键值的节点，有，继续。那些不是以 c 作为键值的分支可以不必考虑。

4. 字符串遍历结束，在 Trie 里，当前节点是 c 指向的节点。由于该节点恰好表示字符串“cba”的结束，因此，得出两个字符串合在一起可以构成回文串。

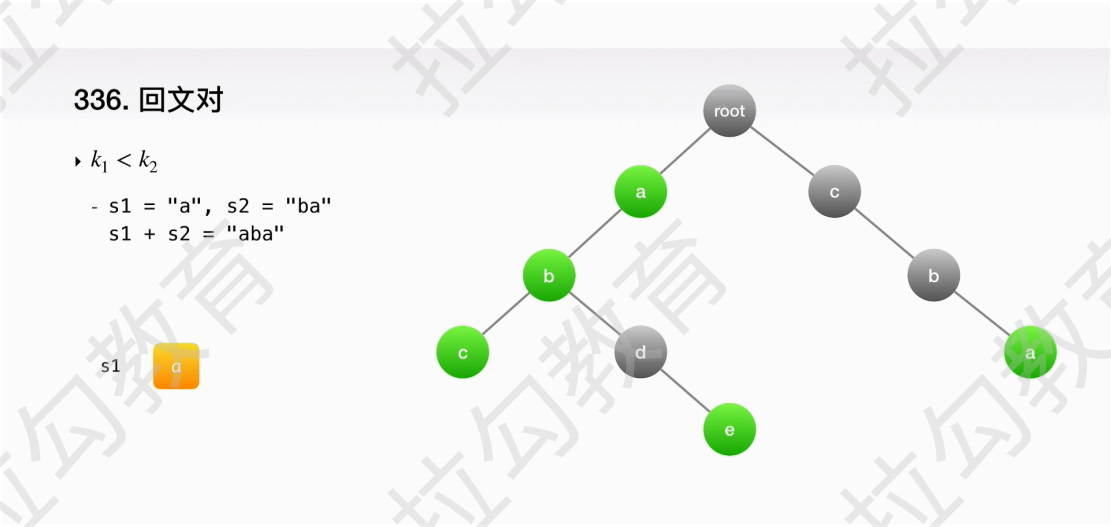
- k1 > k2



举例：s1 = “abc”，s2 = “ba”，s1+s2 = “abcba”。

1. 从 s1 的第一个字符 a 开始，能从 Trie 里找到 a，于是继续。
2. 字符 b，也能找到，并且 b 指向的节点是一个绿色节点，即从 Trie 里找到了字符”ba“。
3. 要能使 s1 + s2 构成回文，条件就是 s1 里剩下的部分也是回文，此时 s1 剩下的是字符 c，而字符 c 是回文，因此，” abc” 和 “ba”能构成回文串。

- k1 < k2



举例：s1 = “a”，s2 = “ba”，s1+s2 =”aba”。

当 s1 遍历完毕后，Trie 来到了 b 节点，由于 b 也是回文，因此它们两个也能构成回文串。

对于情况一、三：

1. s1 字符串一定会被遍历完毕
2. 遍历完毕后，在 Trie 里所对应的节点

- 是 s2 中的最后一个字符；
- 是 s2 的剩余字符
- 只要该剩余字符本身是回文，就可以给这个节点添加一个数组，用来记录从该节点向后所有剩余能构成回文的字符串的下标即可。

对于情况二：

1. 在 Trie 里，当遇到某个绿色节点，而它表示了某个字符串的结束，只要 s1 剩下的字符能构成回文即可。
2. 修改 isEnd，用 index 替代，来得到 Trie 里 s2 的下标。

a. 当 index 为 -1 时，表示不是字符串的结束位置。

b. 当是字符串的结束时，用 index 来记录输入字符串的下标即可。

代码实现

```
// 修改 TrieNode 结构，用 index 替换 isEnd
class TrieNode {
    int index;
    List<Integer> palindromes;
    HashMap<Character, TrieNode> children;

        // 添加一个 palindromes 列表，用来记录从该节点往下的能构成回文的所有输入字符串的下标
        TrieNode() {
        index = -1;
        children = new HashMap<>();
        palindromes = new ArrayList<>();
    }
}
```

主函数代码如下。

```
List<List<Integer>> palindromePairs(String[] words) {
    List<List<Integer>> res = new ArrayList<>(); // 定义一个空的列表，用来记录找到的配对

        TrieNode root = new TrieNode(); // 定义一个 Trie 的根节点 root

    for (int i = 0; i < words.length; i++) {
        addWord(root, words[i], i);
```

```
        } // 创建 Trie

    for (int i = 0; i < words.length; i++) {
        search(words, i, root, res);
    }// 利用 Trie，找出所有的配对

        return res;
}
```

创建 Tire 如下。

```
// 创建 Trie 的时候，从每个字符串的末尾开始遍历
void addWord(TrieNode root, String word, int index) {
    for (int i = word.length() - 1; i >= 0; i--) {
        char ch = word.charAt(i);

                // 对于每个当前字符，如果它还没有被添加
到 children 哈希表里，就创建一个新的节点
                if (!root.children.containsKey(ch)) {
            root.children.put(ch, new TrieNode());
        }
        // 若该字符串从头开始到当前位置能成为回文的话，把这个字符串
的下标添加到这个 Trie 节点的回文列表里
                if (isPalindrome(word, 0, i)) {
            root.palindromes.add(index);
        }

        root = root.children.get(ch);
    }

    // 当对该字符串创建完 Trie 之后，将字符串的下标添加到回文列表
里，并且将它赋给 index
    root.palindromes.add(index);
    root.index = index;

}
```

若该字符串从头开始到当前位置能成为回文的话，把这个字符串的下标添加到这个 Trie 节点的回文列表里。例如，如果字符串是“aaaba”，由于我们从后面

往前面遍历，当遍历到字符 b 的时候，发现 aaa 是回文，于是更新 b 所指向的那个节点，说该节点往下有一个字符串能构成回文。

处理查找如下。

```
//  处理查找，从头遍历每个字符串，然后从  Trie  里寻找匹配的字符串
void  search(String[]  words,  int  i,  TrieNode  root,  List<List<In
teger>>  res)  {

    //  k1  >  k2，且  s1  剩下的字符能构成回文，就把这对组合添加到结
果中
        //  k1=k2  或  k1<k2，只需要把回文列表里的字符串都和  s1  组
合即可
        for  (int  j  =  0;  j  <  words[i].length();  j++)  {
        if  (root.index  >=  0  &&  root.index  !=  i  &&
                        isPalindrome(words[i],  j,  words[i].length()
  -  1))
                {
            res.add(Arrays.asList(i,  root.index));
                }

        root  =  root.children.get(words[i].charAt(j));
            if  (root  ==  null)  return;
    }

    for  (int  j  :  root.palindromes)  {
        if  (i  ==  j)  continue;
        res.add(Arrays.asList(i,  j));
    }
}
```

复杂度分析

利用 Trie，在创建和查找的过程中，最多会遇到 n×k 个节点，而且会进行回文检查，所以整体的时间复杂度是：O(n×k×k)。

如果字符串的字符个数是在一定范围之内的，那么这个问题就可以优化成一个近乎于线性问题了。

例题分析二

LeetCode 第 340 题：给定一个字符串 s ，找出至多包含 k 个不同字符的最长子串 T。

示例 1

输入: s = "eceba", k = 2

输出: 3

解释: 则 T 为 "ece"，所以长度为 3。

示例 2

输入: s = "aa", k = 1

输出: 2

解释: 则 T 为 "aa"，所以长度为 2。

解题思路：暴力法

思路：找出所有的子串，然后逐一检查是否最多包含 k 个不同的字符。

实现：用一个哈希表或者哈希集合去统计。

复杂度：O(n^2)。

第 8 课讲解了一道 LeetCode 的题目，给定一个字符串，找出无重复字符的最长子串。当时提出了一种比较聪明的办法，能够在 O(n) 的时间里找到答案。上述例题其实是它的另外一种扩展，可以运用相似的策略来进行。

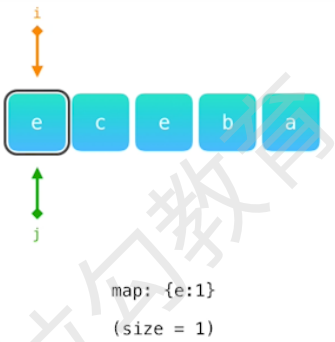
举例：s = “eceba”，k = 2。

实现过程如下。

340. 至多包含 K 个不同字符的最长子串

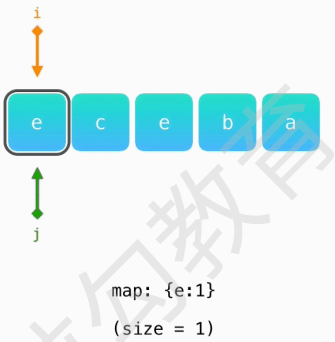
用两个快慢指针：i 和 j，i 是慢指针，j 是快指针。一开始，两个指针都指向字符串的开头。另外，还需要一个哈希表来统计每个字符出现的个数，同时用来统计不同字符的个数。

340. 至多包含 K 个不同字符的最长子串

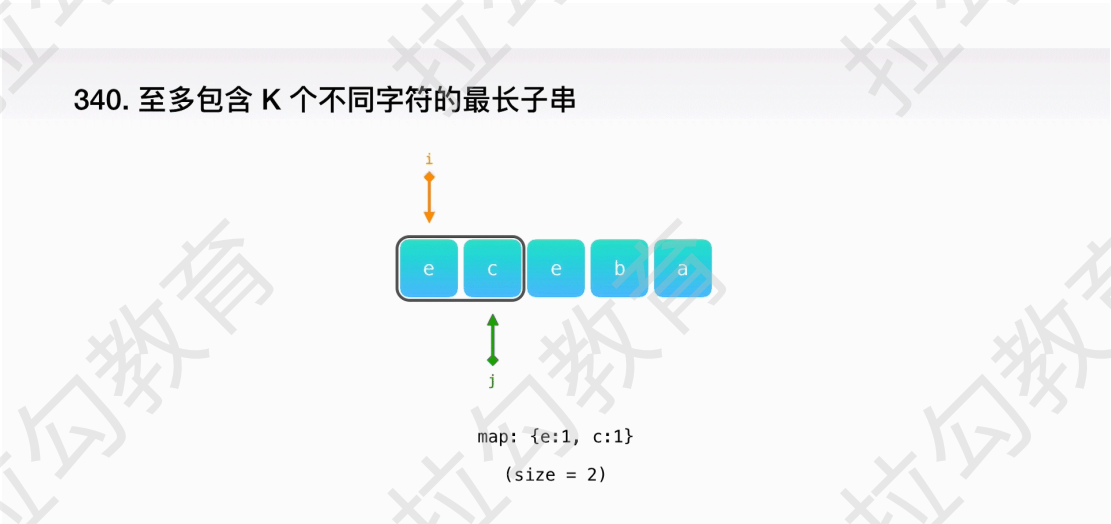


1. 每次将快指针指向的字符添加到哈希表中，统计它出现的次数。第一个字符是 e，加入到 map 中。

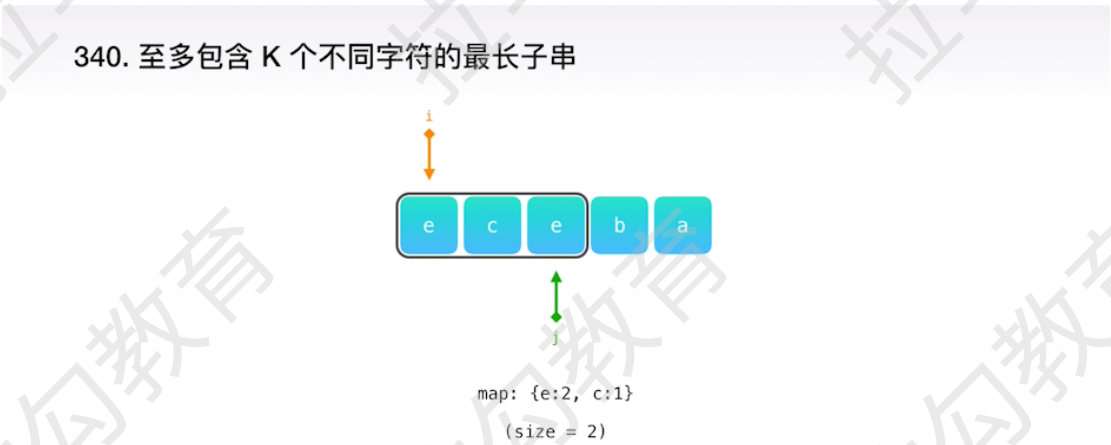
340. 至多包含 K 个不同字符的最长子串



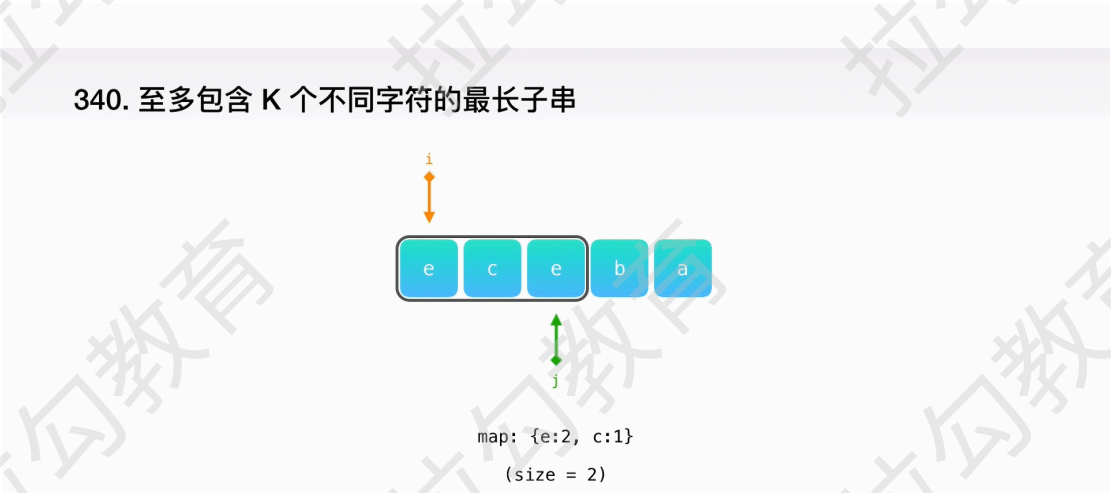
2. map 的大小为 1，表明到目前为止出现了一个字符。由于 map 的大小还没有超过 k，快指针向前移动一步。



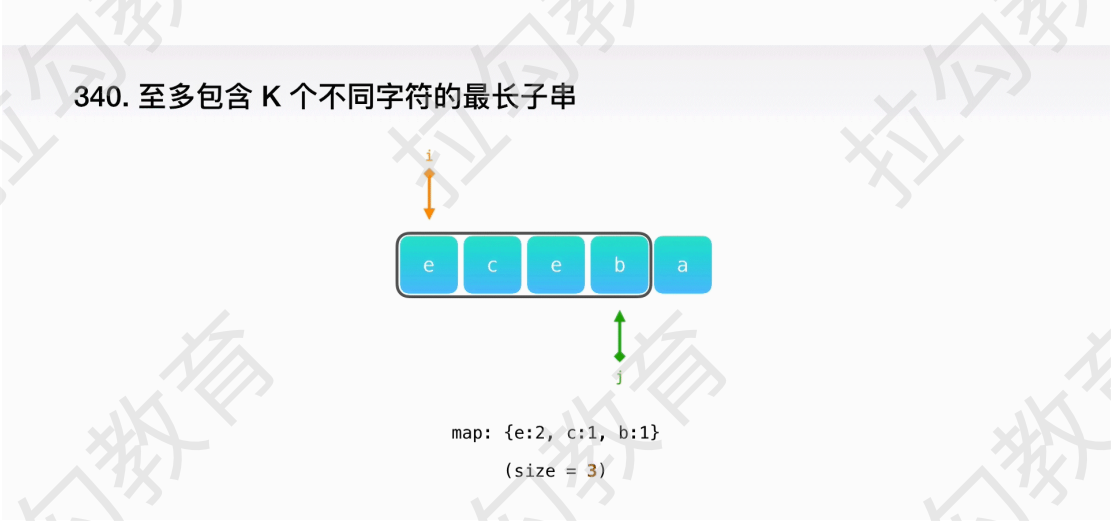
3. j 指向的字符是 c，同样统计到 map 中，此时 map 的大小为 2，也没有超过 k，快指针继续移动。



4. 当前 j 指向的字符是 e，现在 e 出现了 2 次，但是 map 的大小还是 2，表明到目前为止只看到两个不同的字符，即 e 和 c。

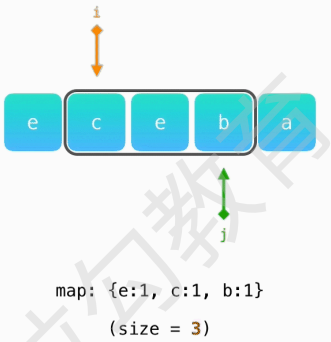


5. 继续移动 j 指针，出现了新的字符 b，加入到 map 中。



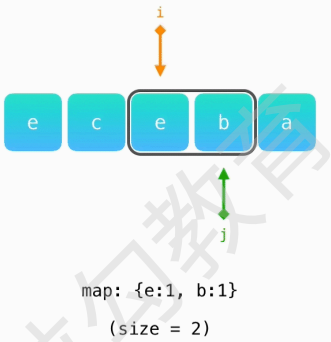
6. 此时 map 的大小为 3，已经超过了 2，于是慢指针开始删除字符，目的是为了控制住 map 的大小不超过 2。

340. 至多包含 K 个不同字符的最长子串



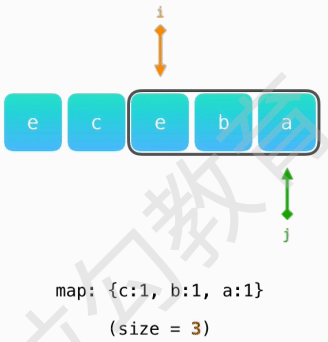
7. 当把第一个字符删除的时候，在 map 里更新 e 字符的计数，但是整个 map 的大小还是等于 3，继续相同的操作。

340. 至多包含 K 个不同字符的最长子串



8. c 的个数只有一个，直接把它从 map 里删除掉。现在 map 的大小恢复正常，继续移动快指针。

340. 至多包含 K 个不同字符的最长子串



9. 当把 a 添加到 map 里后，map 的大小又超过了 2，于是移动慢指针，把它指向的字符从 map 中删除掉。

10. 结束循环。

代码实现

- 初始化一个哈希表 map，用来统计所出现了的不同字符。
- 用 max 变量记录最长的子串，其中子串最多包含 k 个不同的字符。
- 用快慢指针遍历字符串。
- 将快指针指向的字符加入到 map 中，统计字符出现的次数。
- 如果发现 map 的大小超过了 k，那么就得开始不断地将慢指针所指向的字符从 map 里清除掉。
- 首先获取当前慢指针指向的字符。
- 将它在 map 中的计数减一。

- 一旦它的统计次数变成了 0，就可以把它从 map 中删掉了。
- 接下来，慢指针继续往前走。
- 当 map 的大小恢复正常了，统计一下当前子串的长度。
- 最后返回最大的子串长度。

```
int lengthOfLongestSubstringKDistinct(String s, int k) {
    HashMap<Character, Integer> map = new HashMap<>();
    int max = 0;

    for (int i = 0, j = 0; j < s.length(); j++) {
        char cj = s.charAt(j);

        // Step 1. count the character
        map.put(cj, map.getOrDefault(cj, 0) + 1);

        // Step 2. clean up the map if condition doesn't ma
tch
        while (map.size() > k) {
            char ci = s.charAt(i);

            map.put(ci, map.get(ci) - 1);

            if (map.get(ci) == 0) {
                map.remove(ci); // that character count has bec
ome 0
                        }
            i++;
    }

        // Step 3. condition matched, now update the result
    max = Math.max(max, j - i + 1);
    }

    return max;
}
```

复杂度分析

快慢指针遍历字符串一遍，时间复杂度为 O(n)。

运用了一个 map 来作统计，空间复杂度为 O(n)。

例题分析三

LeetCode 第 407 题：给定一个 m x n 的矩阵，其中的值均为正整数，代表二维高度图每个单元的高度，请计算图中形状最多能接多少体积的雨水。

说明：m 和 n 都是小于 110 的整数。每一个单位的高度都大于 0 且小于 20000。

示例：

给出如下 3x6 的高度图:

```
[
  [1,4,3,1,3,2],
  [3,2,1,3,2,4],
  [2,3,3,2,3,1]
]
```

返回 4。

407. 接雨水 II

给定一个 m × n 的矩阵，其中的值均为正整数，代表二维高度图每个单元的高度，请计算图中形状最多能接多少体积的水。

说明：

m 和 n 都是小于 110 的整数。每一个单位的高度都大于 0 且小于 20000。

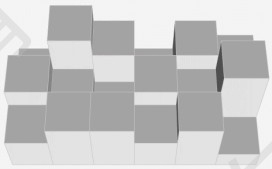
示例：

给出如下 3×6 的高度图：

```
[
    [1, 4, 3, 1, 3, 2],
    [3, 2, 1, 3, 2, 4],
    [2, 3, 3, 2, 3, 1]
]
```

返回：4

这是下雨前的高度图状态。



1. 下雨前的高度图 [[1,4,3,1,3,2],[3,2,1,3,2,4],[2,3,3,2,3,1]]。
2. 下雨后，雨水将会被存储在这些方块中，总的接雨水量是 4。

解题思路一：从内向外

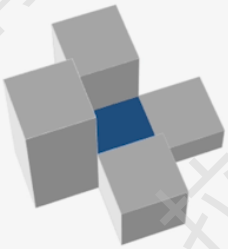
基本情况

举例：假如有一个点高度是 0，而它四周的柱子的高度分别是 1，2，3，4。

解题思路

假设：

1
2 0 3
4



解法：中间的那个位置最多能接高度为 1 的水，因为它的四周最矮的柱子是 1。

扩展情况

举例：假设现在 0 的周围是如下情况，那么 0 那个位置能接水的高度还是 1 吗？

答案应该是 4。

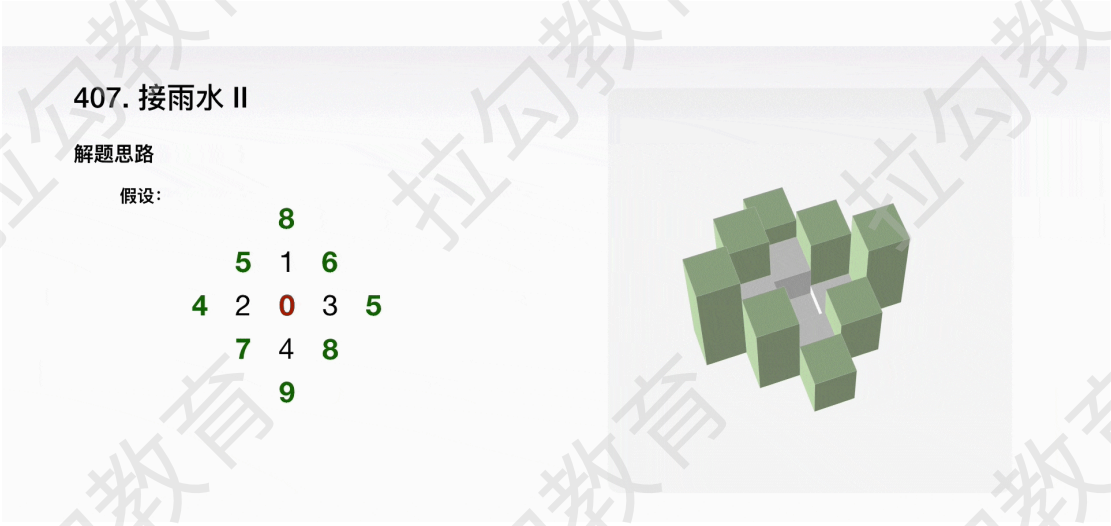
总结思路：对于每个点，都要不断地往外去寻找那个高过自己的最矮的柱子。假设在平面上，一共有 n 个点，按照这样的算法去计算所有的点的接水高度，复杂度是 O(n^3)。

解题思路二：从外向内

为了提高效率，采用“农村包围城市”的策略，从外面往里面进行计算。

者是因为，每个点都必须找到最外围的高度，否则无法确定它能接多少雨水。既然如此，为什么不从最外面开始呢？即，每一次我们都从外面最矮的开始，慢慢地往里面计算。

以上述例子说明。



1. 最外围开始，而最外围的方块无法承载雨水。
2. 从最外围的高度中选择最矮的柱子，先对它的邻居进行处理。这是因为决定能够接多少雨水并不是由周围最高的柱子决定，而是由最矮的决定。
3. 高度 4 是最矮的，于是对其做 BFS，它的邻居是高度为 2 的方块。

4. 由于 2 小于 4，2 的位置能够接纳高度为 2 的雨水，于是这个位置上的高度就变成了 4。
5. 还是从最矮的点出发，还是 4，它的邻居是 0，于是 0 所能接的雨水高度就是 4。
6. 还是 4 是最矮，可以更新它周围的点在接了雨水后的高度。

那么，如何快速知道接下来哪个高度最矮呢？可以用优先队列来提高速度。

代码实现

代码实现如下，为了配合优先队列的操作，定义一个 Cell 类，用来保存每个方块的坐标以及接了雨水后的高度。

```
class Cell {
    int row;
    int col;
    int height;

    public Cell(int row, int col, int height) {
        this.row = row;
        this.col = col;
        this.height = height;
    }
}
```

首先对输入进行一些基本的判断。用变量 m 和 n 分别表示输入矩阵的行数和列数。定义一个优先队列或者最小堆，按照每个方块接雨水后的高度排列。初始化优先队列的时候，把矩形的外围四个边上的方块都加入到优先队列中。

进入 while 循环，开始进行 BFS。每次，从优先队列中取出高度最矮的方块。从四个方向扩散。该方向上的邻居方块能接多少雨水，取决于它是否低于当前的方块了。同时，将新方块加入到优先队列中。

最后返回承接雨水的总量。

```
public int trapRainWater(int[][] heights) {
    // Sanity check
    if (heights == null || heights.length == 0 || heights[0]
.length == 0) {
        return 0;
    }

    int m = heights.length;
    int n = heights[0].length;

    PriorityQueue<Cell> queue = new PriorityQueue(new Comparator
<Cell>() {
        public int compare(Cell a, Cell b) { return a.height
 - b.height; }
    });

    boolean[][] visited = new boolean[m][n];

    // Initially, add all the Cells which are on borders to
 the queue.
    for (int i = 0; i < m; i++) {
        visited[i][0] = true;
        visited[i][n - 1] = true;
        queue.offer(new Cell(i, 0, heights[i][0]));
        queue.offer(new Cell(i, n - 1, heights[i][n - 1]));
    }

    for (int j = 0; j < n; j++) {
        visited[0][j] = true;
        visited[m - 1][j] = true;
                queue.offer(new Cell(0, j, heights[0][j]));
        queue.offer(new Cell(m - 1, j, heights[m - 1][j]));
    }

    // From the borders, pick the shortest cell visited and
 check its
```

```
        // neighbors:
    // If the neighbor is shorter, collect the water it can
  trap and update
        // its height as its height plus the water trapped.
    // Add all its neighbors to the queue.
    int[][] dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    int total = 0;

        while (!queue.isEmpty()) {
        Cell cell = queue.poll();

                for (int[] dir : dirs) {
            int row = cell.row + dir[0];
            int col = cell.col + dir[1];

                        if (row >= 0 && row < m && col >=
0 && col < n && !visited[row][col])
            {
                visited[row][col] = true;
                total += Math.max(0, cell.height - heights[row][
col]);
                queue.offer(
                    new Cell(row, col, Math.max(heights[row][col],
 cell.height))
                );
            }
        }
    }

        return total;

}
```

复杂度分析

假设一共有 m 行 n 列，那么一共有 m×n 个方块。对于每个方块，都有可能会进行优先队列的操作，而优先队列的大小为 m + n，加上初始化优先队列的操作时间，因此，整体的时间复杂度为 O(m + n) + O(m×n×log(m + n)) = O(m×n×log(m + n))。由上可知，将复杂度下降了一个维度。