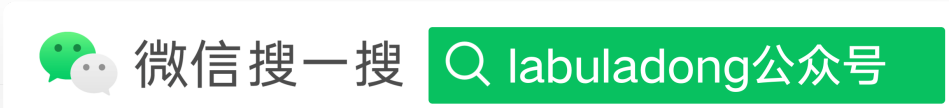
我写了首诗，把滑动窗口算法变成了默写题



通知： **数据结构精品课 V1.6 持续更新中，** **第八期打卡挑战（升级版）** **7/11 截止报名，B 站已更新** **核心算法框架系列视频。**

读完本文，你不仅学会了算法套路，还可以顺便解决如下题目：

牛客	LeetCode	力扣	难度
-	3. Longest Substring Without Repeating Characters	3. 无重复字符的最长子串	🟠
-	438. Find All Anagrams in a String	438. 找到字符串中所有字母异位词	🟠
-	567. Permutation in String	567. 字符串的排列	🟠
-	76. Minimum Window Substring	76. 最小覆盖子串	🔴
-	-	剑指 Offer 48. 最长不含重复字符的子字符串	🟠
-	-	剑指 Offer II 014. 字符串中的变位词	🟠
-	-	剑指 Offer II 015. 字符串中的所有变位词	🟠

-	-	剑指 Offer II 016. 不含重复字符的最长子字符串	
-	-	剑指 Offer II 017. 含有所有字符的最短字符串	

本文有视频版《滑动窗口算法核心模板框架》（由于 B 站限制站外视频的清晰度，建议跳转到 B 站观看）：

鉴于前文 二分搜索框架详解 的那首《二分搜索升天词》很受好评，并在民间广为流传，成为安睡助眠的一剂良方，今天在滑动窗口算法框架中，我再次编写一首小诗来歌颂滑动窗口算法的伟大（手动狗头）：

链表子串数组题，用双指针别犹豫。
双指针家三兄弟，各个都是万人迷。

快慢指针最神奇，链表操作无压力。
归并排序找中点，链表成环搞判定。

左右指针最常见，左右两端相向行。
反转数组要靠它，二分搜索是弟弟。

滑动窗口老猛男，子串问题全靠它。
左右指针滑窗口，一前一后齐头进。

自诩十年老司机，怎料农村道路滑。
一不小心滑到了，鼻青脸肿少颗牙。
算法思想很简单，出了bug想升天。

labuladong稳若狗，一套框架不翻车。
一路漂移带闪电，算法变成默写题。
此等神人何处寻？全靠缘分不可期！
labuladong公众号，开启算法新天地。
关注标星加分享，“下次一定”不可取。

哈哈，我自己快把自己夸上天了，大家乐一乐就好，不要当真：）

关于双指针的快慢指针和左右指针的用法，可以参见前文 双指针技巧汇总，本文就解决一类最难掌握的双指针技巧：滑动窗口技巧。总结出一套框架，可以保你闭着眼睛都能写出正确的解法。

说起滑动窗口算法，很多读者都会头疼。这个算法技巧的思路非常简单，就是维护一个窗口，不断滑动，然后更新答案么。LeetCode 上有起码 10 道运用滑动窗口算法的题目，难度都是中等和困难。该算法的大致逻辑如下：

```
int left = 0, right = 0;

while (right < s.size()) {
    // 增大窗口
    window.add(s[right]);
```

```
        while (window needs shrink) {
            // 缩小窗口
            window.remove(s[left]);
            left++;
        }
    }
```

这个算法技巧的时间复杂度是 O(N)，比字符串暴力算法要高效得多。

其实困扰大家的，不是算法的思路，而是各种细节问题。比如说如何向窗口中添加新元素，如何缩小窗口，在窗口滑动的哪个阶段更新结果。即便你明白了这些细节，也容易出 bug，找 bug 还不知道怎么找，真的挺让人心烦的。

所以今天我就写一套滑动窗口算法的代码框架，我连再哪里做输出 debug 都给你写好了，以后遇到相关的问题，你就默写出来如下框架然后改三个地方就行，还不会出 bug：

```
/* 滑动窗口算法框架 */
void slidingWindow(string s) {
    unordered_map<char, int> window;

    int left = 0, right = 0;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 增大窗口
        right++;
        // 进行窗口内数据的一系列更新
        ...

        /*** debug 输出的位置 ***/
        printf("window: [%d, %d)\n", left, right);
        /********************/

        // 判断左侧窗口是否要收缩
        while (window needs shrink) {
            // d 是将移出窗口的字符
            char d = s[left];
            // 缩小窗口
            left++;
            // 进行窗口内数据的一系列更新
            ...
        }
```

其中两处 `...` 表示的更新窗口数据的地方，到时候你直接往里面填就行了。

而且，这两个 `...` 处的操作分别是扩大和缩小窗口的更新操作，等会你会发现它们操作是完全对称的。

另外，虽然滑动窗口代码框架中有一个嵌套的 while 循环，但算法的时间复杂度依然是 `O(N)`，其中 `N` 是输入字符串/数组的长度。

为什么呢？简单说，字符串/数组中的每个元素都只会进入窗口一次，然后被移出窗口一次，不会说有某些元素多次进入和离开窗口，所以算法的时间复杂度就和字符串/数组的长度成正比。后文算法时空复杂度分析实用指南 有具体讲时间复杂度的估算，这里就不展开了。

说句题外话，我发现很多人喜欢执着于表象，不喜欢探求问题的本质。比如说有很多人评论我这个框架，说什么散列表速度慢，不如用数组代替散列表；还有很多人喜欢把代码写得特别短小，说我这样代码太多余，影响编译速度，LeetCode 上速度不够快。

我的意见是，算法主要看时间复杂度，你能确保自己的时间复杂度最优就行了。至于 LeetCode 所谓的运行速度，那个都是玄学，只要不是慢的离谱就没啥问题，根本不值得你从编译层面优化，不要舍本逐末……

我的公众号重点在于算法思想，你把框架思维了然于心，然后随你魔改代码好吧，你高兴就好。

言归正传，下面就直接上**四道**力扣原题来套这个框架，其中第一道题会详细说明其原理，后面四道就直接闭眼睛秒杀了。

因为滑动窗口很多时候都是在处理字符串相关的问题，而 Java 处理字符串不方便，所以本文代码为 C++ 实现。不会用到什么编程语言层面的奇技淫巧，但是还是简单介绍一下一些用到的数据结构，以免有的读者因为语言的细节问题阻碍对算法思想的理解：

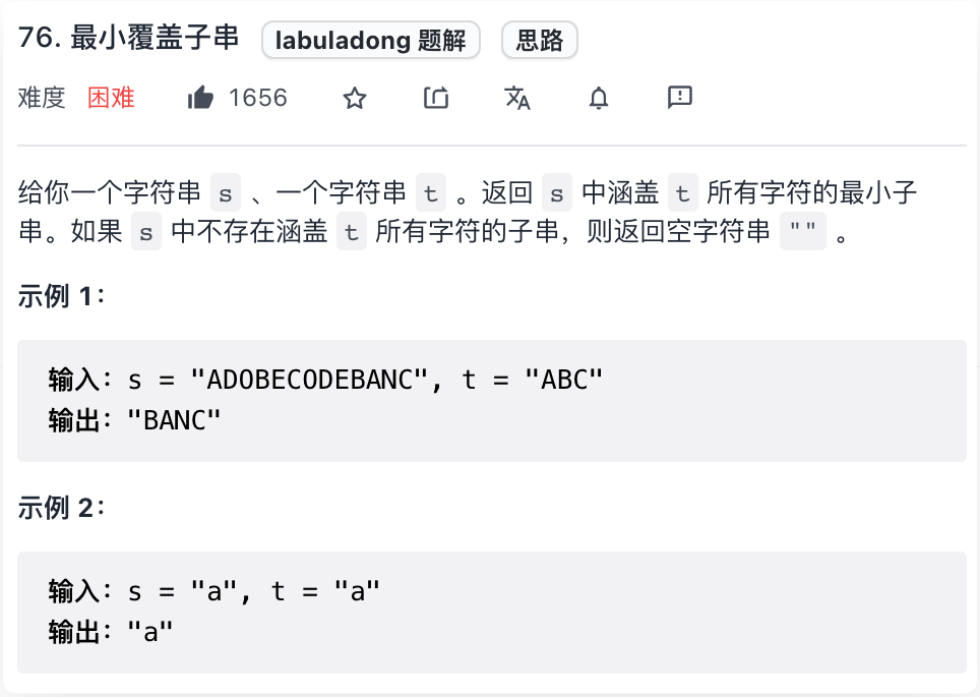
`unordered_map` 就是哈希表（字典），相当于 Java 的 `HashMap`，它的一个方法 `count(key)` 相当于 Java 的 `containsKey(key)` 可以判断键 key 是否存在。

可以使用方括号访问键对应的值 `map[key]`。需要注意的是，如果该 `key` 不存在，C++ 会自动创建这个 key，并把 `map[key]` 赋值为 0。所以代码中多次出现的 `map[key]++` 相当于 Java 的 `map.put(key, map.getOrDefault(key, 0) + 1)`。

另外，Java 中的 Integer 和 String 这种包装类不能直接用 `==` 进行相等判断，而应该使用类的

一、最小覆盖子串

先来看看力扣第 76 题「最小覆盖子串」难度 Hard：



76. 最小覆盖子串 labuladong 题解 思路

难度 困难 1656

给你一个字符串 `s` 、一个字符串 `t` 。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""` 。

示例 1：

```
输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"
```

示例 2：

```
输入：s = "a", t = "a"
输出："a"
```

就是说要在 `S`(source) 中找到包含 `T`(target) 中全部字母的一个子串，且这个子串一定是所有可能子串中最短的。

如果我们使用暴力解法，代码大概是这样的：

```
for (int i = 0; i < s.size(); i++)
    for (int j = i + 1; j < s.size(); j++)
        if s[i:j] 包含 t 的所有字母:
            更新答案
```

思路很直接，但是显然，这个算法的复杂度肯定大于 O(N^2) 了，不好。

滑动窗口算法的思路是这样：

开区间 `[left, right)` 称为一个「窗口」。

> PS：理论上你可以设计两端都开或者两端都闭的区间，但设计为左闭右开区间是最方便处理的。因为这样初始化 `left = right = 0` 时区间 `[0, 0)` 中没有元素，但只要让 `right` 向右移动（扩大）一位，区间 `[0, 1)` 就包含一个元素 `0` 了。如果你设置为两端都开的区间，那么让 `right` 向右移动一位后开区间 `(0, 1)` 仍然没有元素；如果你设置为两端都闭的区间，那么初始区间 `[0, 0]` 就包含了一个元素。这两种情况都会给边界处理带来不必要的麻烦。

2、我们先不断地增加 `right` 指针扩大窗口 `[left, right)`，直到窗口中的字符串符合要求（包含了 `T` 中的所有字符）。

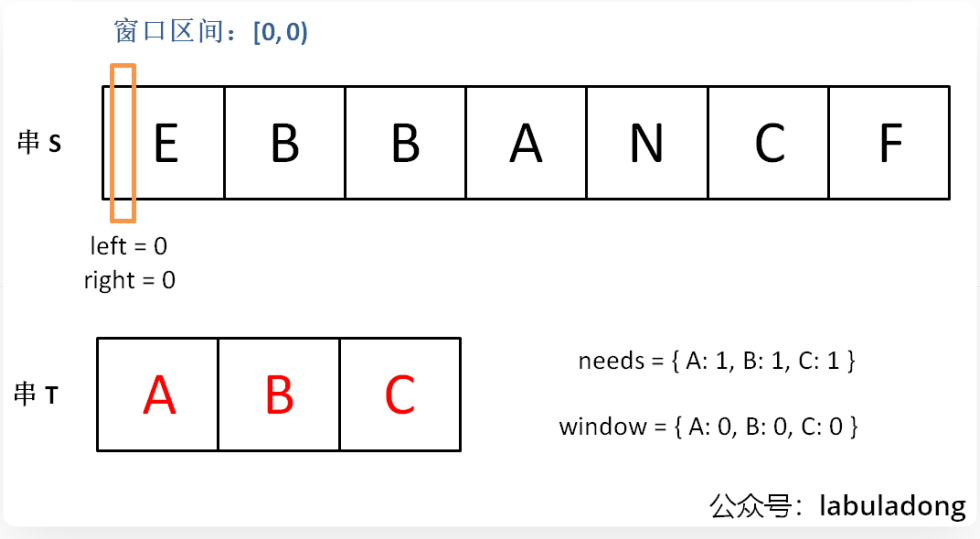
3、此时，我们停止增加 `right`，转而不断增加 `left` 指针缩小窗口 `[left, right)`，直到窗口中的字符串不再符合要求（不包含 `T` 中的所有字符了）。同时，每次增加 `left`，我们都要更新一轮结果。

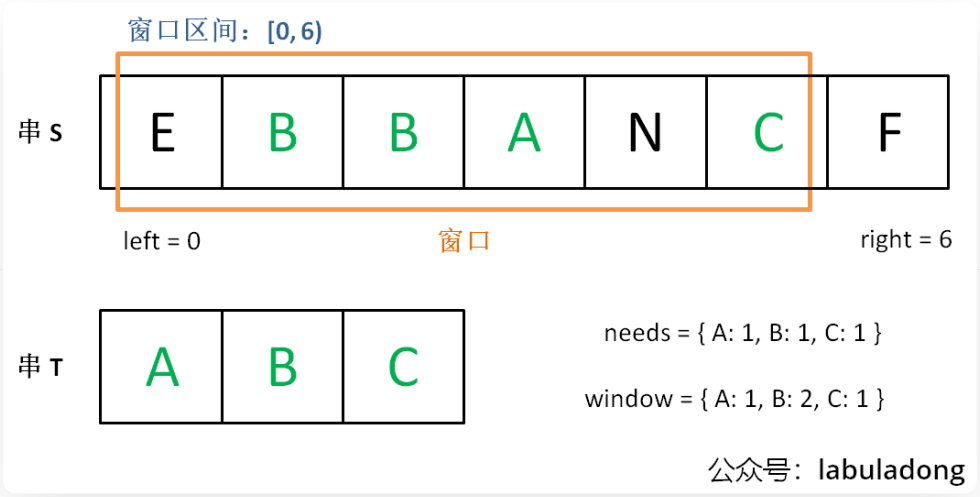
4、重复第 2 和第 3 步，直到 `right` 到达字符串 `S` 的尽头。

这个思路其实也不难，**第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解**，也就是最短的覆盖子串。左右指针轮流前进，窗口大小增增减减，窗口不断向右滑动，这就是「滑动窗口」这个名字的来历。

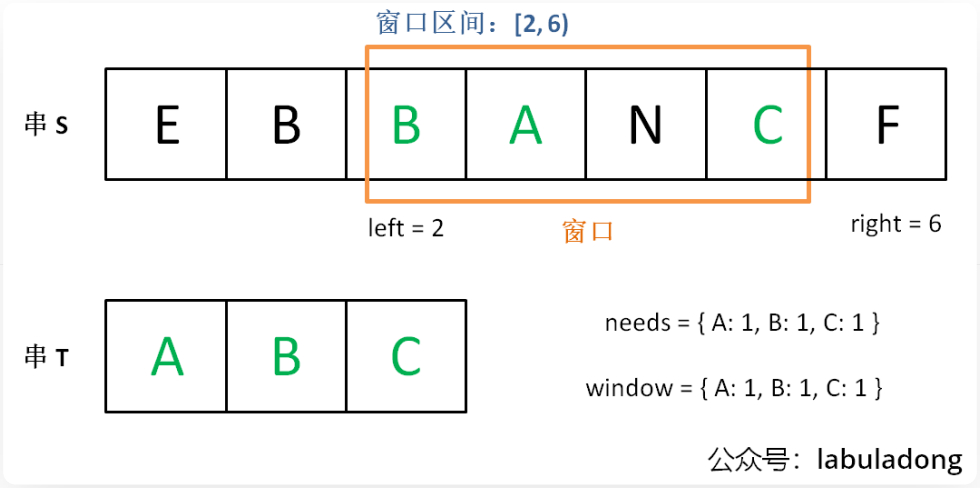
下面画图理解一下，`needs` 和 `window` 相当于计数器，分别记录 `T` 中字符出现次数和「窗口」中的相应字符的出现次数。

初始状态：

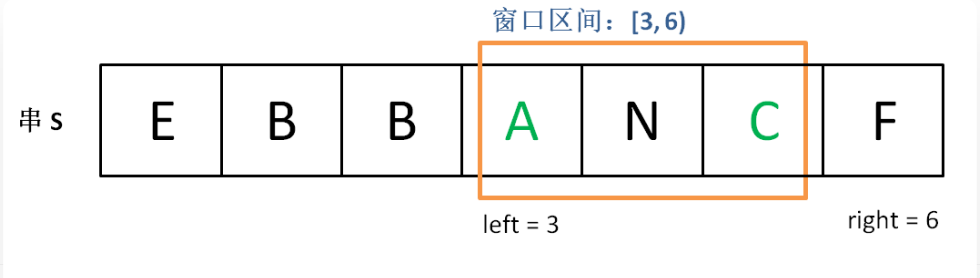




现在开始增加 `left`，缩小窗口 `[left, right)`：



直到窗口中的字符串不再符合要求，`left` 不再继续移动：



之后重复上述过程，先移动 `right`，再移动 `left`..... 直到 `right` 指针到达字符串 `S` 的末端，算法结束。

如果你能够理解上述过程，恭喜，你已经完全掌握了滑动窗口算法思想。**现在我们来看看这个滑动窗口代码框架怎么用**：

首先，初始化 `window` 和 `need` 两个哈希表，记录窗口中的字符和需要凑齐的字符：

```
unordered_map<char, int> need, window;
for (char c : t) need[c]++;
```

然后，使用 `left` 和 `right` 变量初始化窗口的两端，不要忘了，区间 `[left, right)` 是左闭右开的，所以初始情况下窗口没有包含任何元素：

```
int left = 0, right = 0;
int valid = 0;
while (right < s.size()) {
    // 开始滑动
}
```

其中 `valid` 变量表示窗口中满足 `need` 条件的字符个数，如果 `valid` 和 `need.size` 的大小相同，则说明窗口已满足条件，已经完全覆盖了串 `T`。

现在开始套模板，只需要思考以下几个问题：

1、什么时候应该移动 `right` 扩大窗口？窗口加入字符时，应该更新哪些数据？

2、什么时候窗口应该暂停扩大，开始移动 `left` 缩小窗口？从窗口移出字符时，应该更新哪些数据？

如果一个字符进入窗口，应该增加 `window` 计数器；如果一个字符将移出窗口的时候，应该减少 `window` 计数器；当 `valid` 满足 `need` 时应该收缩窗口；应该在收缩窗口的时候更新最终结果。

下面是完整代码：

```
string minWindow(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    // 记录最小覆盖子串的起始索引及长度
    int start = 0, len = INT_MAX;
    while (right < s.size()) {
        // c 是将移入窗口的字符
        char c = s[right];
        // 扩大窗口
        right++;
        // 进行窗口内数据的一系列更新
        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c])
                valid++;
        }

        // 判断左侧窗口是否要收缩
        while (valid == need.size()) {
            // 在这里更新最小覆盖子串
            if (right - left < len) {
                start = left;
                len = right - left;
            }
            // d 是将移出窗口的字符
            char d = s[left];
            // 缩小窗口
            left++;
            // 进行窗口内数据的一系列更新
            if (need.count(d)) {
                if (window[d] == need[d])
                    valid--;
                window[d]--;
            }
        }
```

```
        return len == INT_MAX ?
            "" : s.substr(start, len);
    }
```

> PS：使用 Java 的读者要尤其警惕语言特性的陷阱。Java 的 Integer，String 等类型判定相等应该用 `equals` 方法而不能直接用等号 `==`，这是 Java 包装类的一个隐晦细节。所以在缩小窗口更新数据的时候，不能直接改写为 `window.get(d) == need.get(d)`，而要用 `window.get(d).equals(need.get(d))`，之后的题目代码同理。

需要注意的是，当我们发现某个字符在 `window` 的数量满足了 `need` 的需要，就要更新 `valid`，表示有一个字符已经满足要求。而且，你能发现，两次对窗口内数据的更新操作是完全对称的。

当 `valid == need.size()` 时，说明 `T` 中所有字符已经被覆盖，已经得到一个可行的覆盖子串，现在应该开始收缩窗口了，以便得到「最小覆盖子串」。

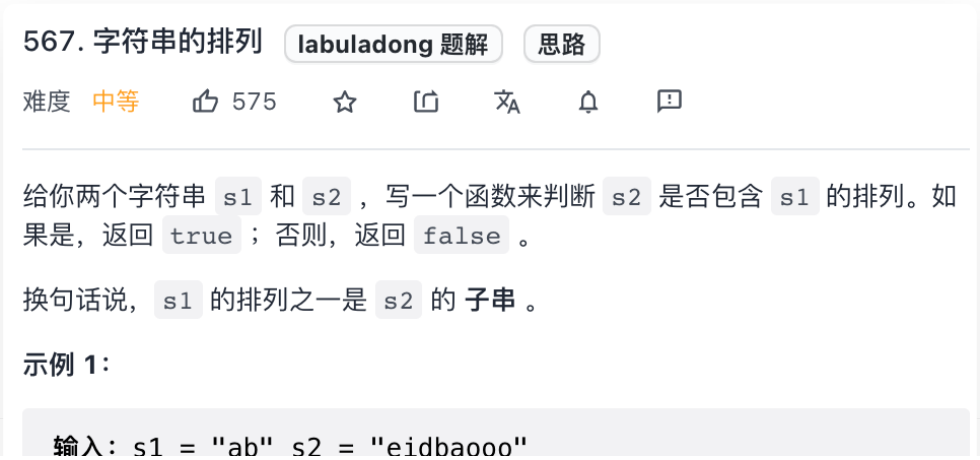
移动 `left` 收缩窗口时，窗口内的字符都是可行解，所以应该在收缩窗口的阶段进行最小覆盖子串的更新，以便从可行解中找到长度最短的最终结果。

至此，应该可以完全理解这套框架了，滑动窗口算法又不难，就是细节问题让人烦得很。**以后遇到滑动窗口算法，你就按照这框架写代码，保准没有 bug，还省事儿**。

下面就直接利用这套框架秒杀几道题吧，你基本上一眼就能看出思路了。

二、字符串排列

这是力扣第 567 题「字符串的排列」，难度中等：



示例 2：

```
输入: s1= "ab" s2 = "eidboaoo"
输出: false
```

注意哦，输入的 `s1` 是可以包含重复字符的，所以这个题难度不小。

这种题目，是明显的滑动窗口算法，**相当给你一个 `S` 和一个 `T`，请问你 `S` 中是否存在一个子串，包含 `T` 中所有字符且不包含其他字符**？

首先，先复制粘贴之前的算法框架代码，然后明确刚才提出的几个问题，即可写出这道题的答案：

```
// 判断 s 中是否存在 t 的排列
bool checkInclusion(string t, string s) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    while (right < s.size()) {
        char c = s[right];
        right++;
        // 进行窗口内数据的一系列更新
        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c])
                valid++;
        }

        // 判断左侧窗口是否要收缩
        while (right - left >= t.size()) {
            // 在这里判断是否找到了合法的子串
            if (valid == need.size())
                return true;
            char d = s[left];
            left++;
            // 进行窗口内数据的一系列更新
            if (need.count(d)) {
                if (window[d] == need[d])
                    valid--;
```

```
            }
        }
        // 未找到符合条件的子串
        return false;
    }
```

对于这道题的解法代码，基本上和最小覆盖子串一模一样，只需要改变几个地方：

1、本题移动 `left` 缩小窗口的时机是窗口大小大于 `t.size()` 时，应为排列嘛，显然长度应该是一样的。

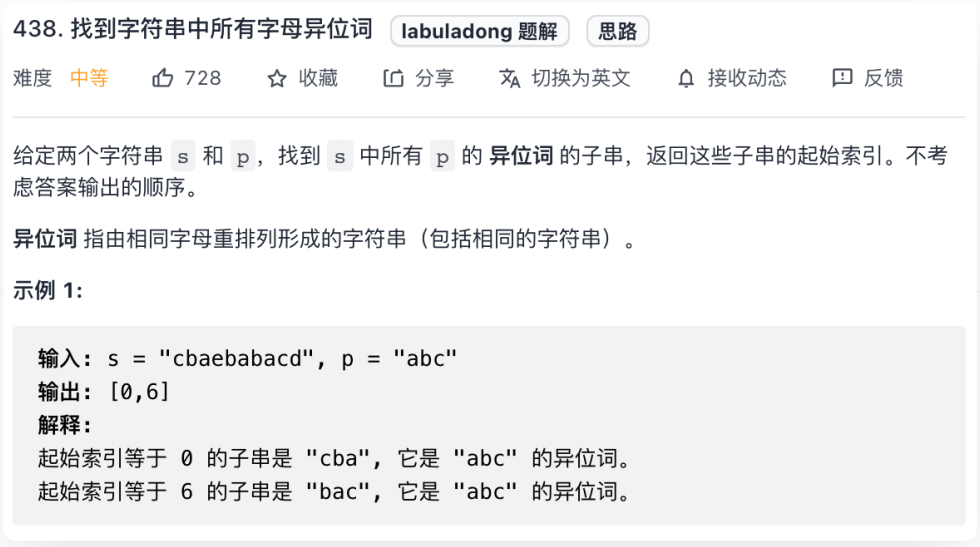
2、当发现 `valid == need.size()` 时，就说明窗口中就是一个合法的排列，所以立即返回 `true`。

至于如何处理窗口的扩大和缩小，和最小覆盖子串完全相同。

> PS：由于这道题中 `[left, right)` 其实维护的是一个**定长**的窗口，窗口大小为 `t.size()`。因为定长窗口每次向前滑动时只会移出一个字符，所以可以把内层的 while 改成 if，结果是一样的。

三、找所有字母异位词

这是力扣第 438 题「找到字符串中所有字母异位词」，难度中等：



438. 找到字符串中所有字母异位词 labuladong 题解 思路

难度 中等 728 收藏 分享 切换为英文 接收动态 反馈

给定两个字符串 s 和 p，找到 s 中所有 p 的 **异位词** 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。

示例 1:

```
输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

个串 `s`，一个串 `t`，找到 `s` 中所有 `t` 的排列，返回它们的起始索引。

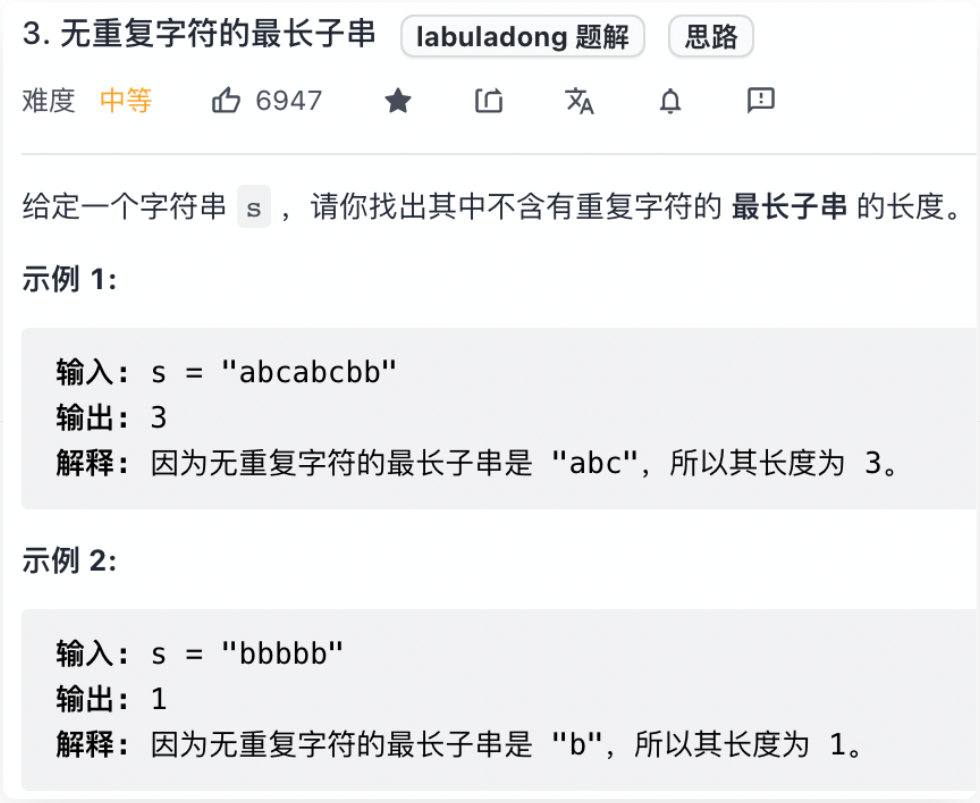
直接默写一下框架，明确刚才讲的 4 个问题，即可秒杀这道题：

```
vector<int> findAnagrams(string s, string t) {
    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    vector<int> res; // 记录结果
    while (right < s.size()) {
        char c = s[right];
        right++;
        // 进行窗口内数据的一系列更新
        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c])
                valid++;
        }
        // 判断左侧窗口是否要收缩
        while (right - left >= t.size()) {
            // 当窗口符合条件时，把起始索引加入 res
            if (valid == need.size())
                res.push_back(left);
            char d = s[left];
            left++;
            // 进行窗口内数据的一系列更新
            if (need.count(d)) {
                if (window[d] == need[d])
                    valid--;
                window[d]--;
            }
        }
    }
    return res;
}
```

跟寻找字符串的排列一样，只是找到一个合法异位词（排列）之后将起始索引加入 `res` 即可。

四、最长无重复子串



3. 无重复字符的最长子串 labuladong 题解 思路

难度 中等 6947

给定一个字符串 `s` ，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

输入: s = "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:

输入: s = "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

这个题终于有了点新意，不是一套框架就出答案，不过反而更简单了，稍微改一改框架就行了：

```
int lengthOfLongestSubstring(string s) {
    unordered_map<char, int> window;

    int left = 0, right = 0;
    int res = 0; // 记录结果
    while (right < s.size()) {
        char c = s[right];
        right++;
        // 进行窗口内数据的一系列更新
        window[c]++;
        // 判断左侧窗口是否要收缩
        while (window[c] > 1) {
            char d = s[left];
            left++;
            // 进行窗口内数据的一系列更新
            window[d]--;
        }
```

```
        }
        return res;
    }
```

这就是变简单了，连 `need` 和 `valid` 都不需要，而且更新窗口内数据也只需要简单的更新计数器 `window` 即可。

当 `window[c]` 值大于 1 时，说明窗口中存在重复字符，不符合条件，就该移动 `left` 缩小窗口了嘛。

唯一需要注意的是，在哪里更新结果 `res` 呢？我们要的是最长无重复子串，哪一个阶段可以保证窗口中的字符串是没有重复的呢？

这里和之前不一样，要在收缩窗口完成后更新 `res`，因为窗口收缩的 while 条件是存在重复元素，换句话说收缩完成后一定保证窗口中没有重复嘛。

好了，滑动窗口算法模板就讲到这里，希望大家能理解其中的思想，记住算法模板并融会贯通。回顾一下，遇到子数组/子串相关的问题，你只要能回答出来以下几个问题，就能运用滑动窗口算法：

1、什么时候应该扩大窗口？

2、什么时候应该缩小窗口？

3、什么时候得到一个合法的答案？

我在 滑动窗口经典习题 中使用这套思维模式列举了更多经典的习题，旨在强化你对算法的理解和记忆，以后就再也不怕子串、子数组问题了。

▸ 引用本文的题目

▸ 引用本文的文章