# 题目

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。

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。

示例 2:

输入: s = "abab", p = "ab"

输出: [0,1,2]

解释:

起始索引等于 0 的子串是 "ab", 它是 "ab" 的异位词。

起始索引等于 1 的子串是 "ba", 它是 "ab" 的异位词。

起始索引等于 2 的子串是 "ab", 它是 "ab" 的异位词。

提示:

1 <= s.length, p.length <= 3 \* 104

s 和 p 仅包含小写字母

# 分析

## 方法一：滑动窗口

思路：

根据题目要求，我们需要在字符串 s 寻找字符串 p 的异位词。因为字符串 p 的异位词的长度一定与字符串 p 的长度相同，所以我们可以在字符串 s 中构造一个长度为与字符串 p 的长度相同的滑动窗口，并在滑动中维护窗口中每种字母的数量；当窗口中每种字母的数量与字符串 p 中每种字母的数量相同时，则说明当前窗口为字符串 p 的异位词。

算法：

在算法的实现中，我们可以使用数组来存储字符串 p 和滑动窗口中每种字母的数量。

细节：

当字符串 s 的长度小于字符串 p 的长度时，字符串 s 中一定不存在字符串 p 的异位词。但是因为字符串 s 中无法构造长度与字符串 p 的长度相同的窗口，所以这种情况需要单独处理。

代码：

class Solution {

public:

vector<int> findAnagrams(string s, string p) {

int sLen = s.size(), pLen = p.size();

if (sLen < pLen) {

return vector<int>();

}

vector<int> ans;

vector<int> sCount(26);

vector<int> pCount(26);

for (int i = 0; i < pLen; ++i) {

++sCount[s[i] - 'a'];

++pCount[p[i] - 'a'];

}

if (sCount == pCount) {

ans.emplace\_back(0);

}

for (int i = 0; i < sLen - pLen; ++i) {

--sCount[s[i] - 'a'];

++sCount[s[i + pLen] - 'a'];

if (sCount == pCount) {

ans.emplace\_back(i + 1);

}

}

return ans;

}

};

复杂度分析

时间复杂度：O(m+(n−m)×Σ)，其中 n 为字符串 s 的长度，m 为字符串 p 的长度，Σ 为所有可能的字符数。我们需要 O(m) 来统计字符串 p 中每种字母的数量；需要 O(m) 来初始化滑动窗口；需要判断 n−m+1 个滑动窗口中每种字母的数量是否与字符串 p 中每种字母的数量相同，每次判断需要 O(Σ) 。因为 s 和 p 仅包含小写字母，所以 Σ=26。

空间复杂度：O(Σ)。用于存储字符串 p 和滑动窗口中每种字母的数量。

## 方法二：优化滑动窗口

思路：

在方法一的基础上，我们不再分别统计滑动窗口和字符串 p 中每种字母的数量，而是统计滑动窗口和字符串 p 中每种字母数量的差；并引入变量 differ 来记录当前窗口与字符串 p 中数量不同的字母的个数，并在滑动窗口的过程中维护它。

在判断滑动窗口中每种字母的数量与字符串 p 中每种字母的数量是否相同时，只需要判断 differ 是否为零即可。

代码：

class Solution {

public:

vector<int> findAnagrams(string s, string p) {

int sLen = s.size(), pLen = p.size();

if (sLen < pLen) {

return vector<int>();

}

vector<int> ans;

vector<int> count(26);

for (int i = 0; i < pLen; ++i) {

++count[s[i] - 'a'];

--count[p[i] - 'a'];

}

int differ = 0;

for (int j = 0; j < 26; ++j) {

if (count[j] != 0) {

++differ;

}

}

if (differ == 0) {

ans.emplace\_back(0);

}

for (int i = 0; i < sLen - pLen; ++i) {

if (count[s[i] - 'a'] == 1) { // 窗口中字母 s[i] 的数量与字符串 p 中的数量从不同变得相同

--differ;

} else if (count[s[i] - 'a'] == 0) { // 窗口中字母 s[i] 的数量与字符串 p 中的数量从相同变得不同

++differ;

}

--count[s[i] - 'a'];

if (count[s[i + pLen] - 'a'] == -1) { // 窗口中字母 s[i+pLen] 的数量与字符串 p 中的数量从不同变得相同

--differ;

} else if (count[s[i + pLen] - 'a'] == 0) { // 窗口中字母 s[i+pLen] 的数量与字符串 p 中的数量从相同变得不同

++differ;

}

++count[s[i + pLen] - 'a'];

if (differ == 0) {

ans.emplace\_back(i + 1);

}

}

return ans;

}

};

复杂度分析：

时间复杂度：O(n+m+Σ)，其中 n 为字符串 s 的长度，m 为字符串 p 的长度，其中Σ 为所有可能的字符数。我们需要 O(m) 来统计字符串 p 中每种字母的数量；需要 O(m) 来初始化滑动窗口；需要 O(Σ) 来初始化 differ；需要 O(n−m) 来滑动窗口并判断窗口内每种字母的数量是否与字符串 p 中每种字母的数量相同，每次判断需要 O(1) 。因为 s 和 p 仅包含小写字母，所以 Σ=26。

空间复杂度：O(Σ)。用于存储滑动窗口和字符串 p 中每种字母数量的差。