

题目描述

给定两个字符串s1和s2和正整数K，其中s1长度为n1，s2长度为n2，在s2中选一个子串，满足：

- 该子串长度为n1+k
- 该子串中包含s1中全部字母，
- 该子串每个字母出现次数不小于s1中对应的字母，

我们称s2以长度k冗余覆盖s1，给定s1，s2，k，求最左侧的s2以长度k冗余覆盖s1的子串的首个元素的下标，如果没有返回-1。

输入描述

输入三行，第一行为s1，第二行为s2，第三行为k，s1和s2只包含小写字母

输出描述

最左侧的s2以长度k冗余覆盖s1的子串首个元素下标，如果没有返回-1。

用例

输入	ab aabcd 1
输出	0
说明	无

题目解析

本题的难点在于如何计算s2选中子串是否能够覆盖住s1，即s2选中子串的中的对应字符个数都大于s1中每个字符个数。

本题可以参考最小覆盖子串中统计覆盖子串字符个数的求解思路。

请大家看：

[LeetCode - 76 最小覆盖子串_伏城之外的博客-CSDN博客](#)

[华为机试 - 完美走位_伏城之外的博客-CSDN博客_完美走位华为](#)

JavaScript算法源码

```
1  /* JavaScript Node ACM模式 控制台输入获取 */
2  const readline = require("readline");
3
4  const rl = readline.createInterface({
5    input: process.stdin,
6    output: process.stdout,
7  });
8
9  const lines = [];
10 rl.on("line", (line) => {
11   lines.push(line);
12
13   if (lines.length === 3) {
14     const [s1, s2, k] = lines;
15     console.log(getResult(s1, s2, k - 0));
16     lines.length = 0;
17   }
18 });
19
20 function getResult(s1, s2, k) {
21   // 在s2中选一个子串，满足：该子串长度为 n1+k
22   const n1 = s1.length;
23   const n2 = s2.length;
24   if (n2 < n1 + k) return -1;
25
26   // 统计s1中所有每个字符出现的次数到count中
27   const count = {};
28   for (let c of s1) {
```

题目解析

本题的难点在于如何计算s2选中子串是否能够覆盖住s1，即s2选中子串的中的对应字符个数都大于s1中每个字符个数。

本题可以参考最小覆盖子串中统计覆盖子串字符个数的求解思路。

请大家看：

[LeetCode - 76 最小覆盖子串_伏城之外的博客-CSDN博客](#)


[华为机试 - 完美走位_伏城之外的博客-CSDN博客_完美走位华为](#)

JavaScript算法源码

```
1  /* JavaScript Node ACM模式 控制台输入获取 */
2  const readline = require("readline");
3
4  const rl = readline.createInterface({
5    input: process.stdin,
6    output: process.stdout,
7  });
8
9  const lines = [];
10 rl.on("line", (line) => {
11   lines.push(line);
12
13   if (lines.length === 3) {
14     const [s1, s2, k] = lines;
15     console.log(getResult(s1, s2, k - 0));
16     lines.length = 0;
17   }
18 });
19
20 function getResult(s1, s2, k) {
21   // 在s2中选一个子串，满足：该子串长度为 n1+k
22   const n1 = s1.length;
23   const n2 = s2.length;
24   if (n2 < n1 + k) return -1;
25
26   // 统计s1中所有每个字符出现的次数到count中
27   const count = {};
28   for (let c of s1) {
29     count[c] ? count[c]++ : (count[c] = 1);
30   }
31
32   // s1字符总数
33   let total = n1;
34
35   // 滑动窗口的左边界从0开始，最大maxI；滑动窗口长度len
36   const maxI = n2 - n1 - k;
37   const len = n1 + k;
38
39   // 统计s2的0~len范围内出现的s1中字符的次数
40   for (let j = 0; j < len; j++) {
41     const c = s2[j];
42     // 如果s2的0~len范围内的字符c，在count[c]存在，则说明c是s1内的字符，此时我们需要count[c]--，如果自减之前，count[c]
43     if (count[c] !== undefined && count[c]-- > 0) {
44       total--;
45     }
46
47     // 如果total为0了，则说明在s2的0~len范围内找到了所有s1中字符
48     if (total === 0) {
49       // 此时可以直接返回起始索引0
50       return 0;
51     }
52   }
53
54   // 下面是从左边界1开始的滑动窗口，利用差异思想，避免重复部分求解
55   for (let i = 1; i <= maxI; i++) {
56     // 滑动窗口右移一格后，失去了s2[i - 1]，得到了s2[i - 1 + len]，其余部分不变
57     const remove = s2[i - 1];
58     const add = s2[i - 1 + len];
59
60     if (count[remove] !== undefined && count[remove]++ >= 0) {
61       total++;
62     }
63
64     if (count[add] !== undefined && count[add]-- > 0) {
65       total--;
66     }
67
68     if (total === 0) {
69       return i;
70     }
71   }
72
73   return -1;
74 }
```

Java算法源码

下面统计s1中各字符数量时，容器没有使用HashMap，因为后期获取和处理HashMap中数据时比较麻烦，而是利

 伏城之外 [已关注](#)

 0   2  1  [专栏目录](#) [已订阅](#)

Java算法源码

下面统计s1中各字符数量时，容器没有使用HashMap，因为后期获取和处理HashMap中数据时比较麻烦，而是利用s1中所有字符都是小写字母的特点，使用128长度的int数组，因为小写字母的ASCII码范围是97~122，因此可以对应到0~127的int数组的索引上。

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(System.in);
6
7         String s1 = sc.next();
8         String s2 = sc.next();
9         int k = sc.nextInt();
10
11         System.out.println(getResult(s1, s2, k));
12     }
13
14     public static int getResult(String s1, String s2, int k) {
15         // 在s2中选一个子串，满足：该子串长度为 n1+k
16         int n1 = s1.length();
17         int n2 = s2.length();
18         if (n2 < n1 + k) return -1;
19
20         // 由于字符串s1中都是小写字母，因此每个字母的ASCII码范围是97~122，因此这里初始化128长度数组来作为统计容器
21         int[] count = new int[128];
22
23         // 统计s1中所有每个字符出现的次数到count中
24         for (int i = 0; i < n1; i++) {
25             int c = s1.charAt(i);
26             count[c]++;
27         }
28
29         // s1字符总数
30         int total = n1;
31
32         // 滑动窗口的左边界从0开始，最大maxI：滑动窗口长度len
33         int maxI = n2 - n1 - k;
34         // s2子串长度
35         int len = n1 + k;
36
37         // 统计s2的0~len范围内出现的s1中字符的次数
38         for (int j = 0; j < len; j++) {
39             int c = s2.charAt(j);
40
41             // 如果s2的0~len范围内的字符c，在count[c]存在，则说明c是s1内有的字符，
42             // 此时我们需要count[c]--，如果自减之前，count[c] > 0，则自减时，total也应该--，否则total不--
43             if (count[c]-- > 0) {
44                 total--;
45             }
46
47             // 如果total为0了，则说明在s2的0~len范围内找到了所有s1中字符
48             if (total == 0) {
49                 // 此时可以直接返回起始索引0
50                 return 0;
51             }
52         }
53
54         // 下面是从左边界1开始的滑动窗口，利用差异思想，避免重复部分求解
55         for (int i = 1; i <= maxI; i++) {
56             // 滑动窗口右移一格后，失去了s2[i - 1]，得到了s2[i - 1 + len]，其余部分不变
57             int remove = s2.charAt(i - 1);
58             int add = s2.charAt(i - 1 + len);
59
60             if (count[remove]++ >= 0) {
61                 total++;
62             }
63
64             if (count[add]-- > 0) {
65                 total--;
66             }
67
68             if (total == 0) {
69                 return i;
70             }
71         }
72         return -1;
73     }
74 }
```

Python算法源码

```
1 # 输入获取
2 s1 = input()
3 s2 = input()
4 k = int(input())
5
6
7 # 算法入口
8 def getResult(s1, s2, k):
9     # 在s2中选一个子串，满足：该子串长度为 n1+k
```

Python算法源码

```
1 # 输入获取
2 s1 = input()
3 s2 = input()
4 k = int(input())
5
6
7 # 算法入口
8 def getResult(s1, s2, k):
9     # 在s2中选一个子串, 满足: 该子串长度为 n1+k
10    n1 = len(s1)
11    n2 = len(s2)
12    if n2 < n1 + k:
13        return -1
14
15    # 统计s1中所有每个字符出现的次数到count中
16    count = {}
17    for c in s1:
18        if count.get(c) is None:
19            count[c] = 1
20        else:
21            count[c] += 1
22
23    # s1字符总数
24    total = n1
25
26    # 滑动窗口的左边界从0开始, 最大maxI: 滑动窗口长度len
27    maxI = n2 - n1 - k
28    n = n1 + k
29
30    # 统计s2的0~len范围内出现的s1中字符的次数
31    for j in range(n):
32        c = s2[j]
33        # 如果s2的0~len范围内的字符c, 在count[c]存在, 则说明c是s1内有的字符, 此时我们需要count[c]--, 如果自减之前, count
34        if count.get(c) is not None:
35            if count[c] > 0:
36                total -= 1
37            count[c] -= 1
38
39    # 如果total为0了, 则说明在s2的0~len范围内找到了所有s1中字符
40    if total == 0:
41        # 此时可以直接返回起始索引0
42        return 0
43
44    # 下面是从左边界1开始的滑动窗口, 利用差异思想, 避免重复部分求解
45    for i in range(1, maxI + 1):
46        # 滑动窗口右移一格后, 失去了s2[i - 1], 得到了s2[i - 1 + len], 其余部分不变
47        remove = s2[i - 1]
48        add = s2[i - 1 + n]
49
50        if count.get(remove) is not None:
51            if count[remove] >= 0:
52                total += 1
53            count[remove] -= 1
54
55        if count.get(add) is not None:
56            if count[add] > 0:
57                total -= 1
58            count[add] -= 1
59
60        if total == 0:
61            return i
62
63    return -1
64
65
66 # 调用算法
67 print(getResult(s1, s2, k))
```