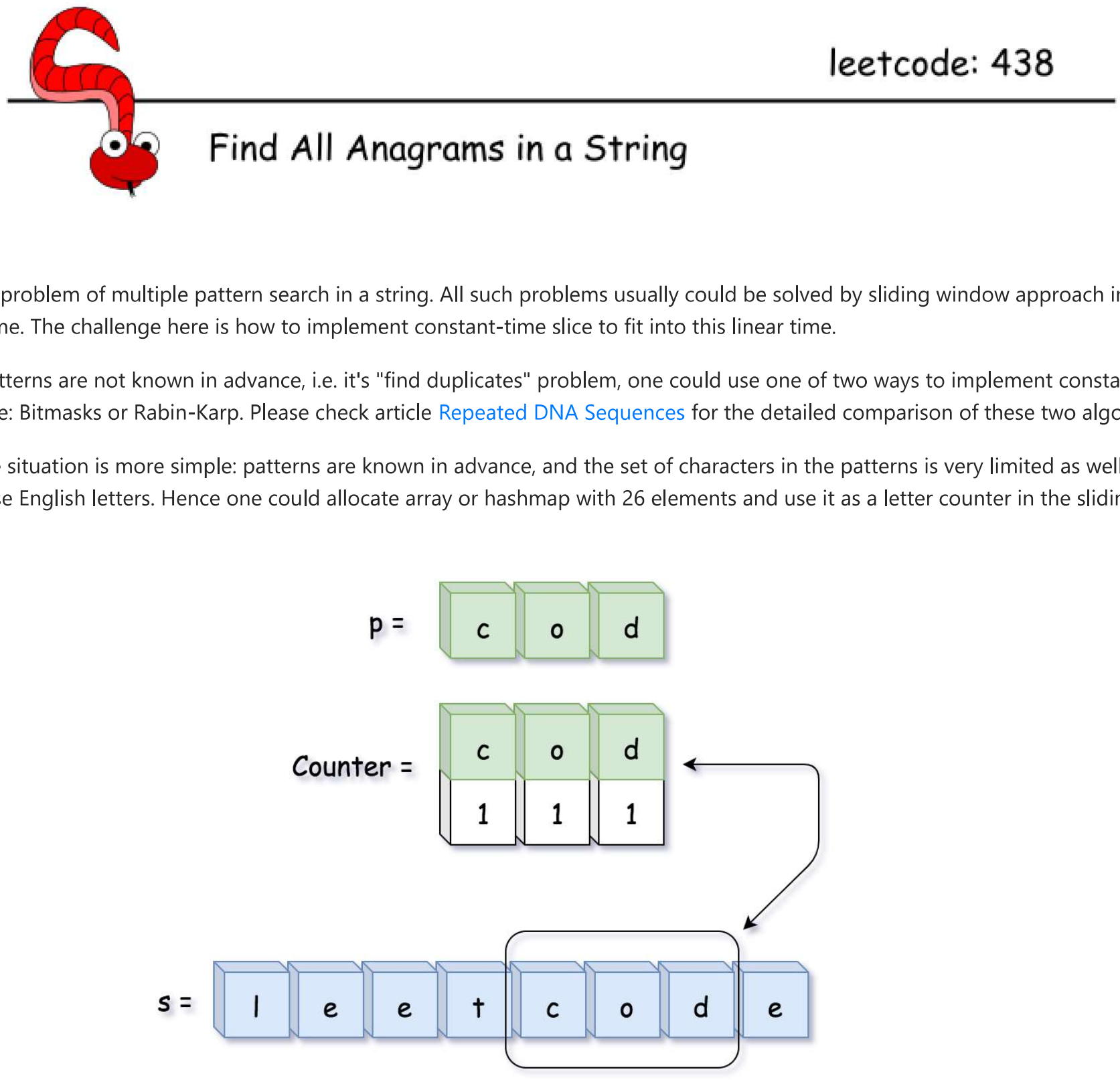
Find All Anagrams in a String

LeetCode Admin Feb 04, 2020

Solution

Solution Template



This is a problem of multiple pattern search in a string. All such problems usually could be solved by sliding window approach in a linear time. The challenge here is how to implement constant-time slice to fit into this linear time.

If the patterns are not known in advance, i.e. it's "find duplicates" problem, one could use one of two ways to implement constant-time slice: Bitmasks or Rabin-Karp. Please check article Repeated DNA Sequences for the detailed comparison of these two algorithms.

Here the situation is more simple: patterns are known in advance, and the set of characters in the patterns is very limited as well: 26 lowercase English letters. Hence one could allocate array or hashmap with 26 elements and use it as a letter counter in the sliding window.

Approach 1: Sliding Window with HashMap

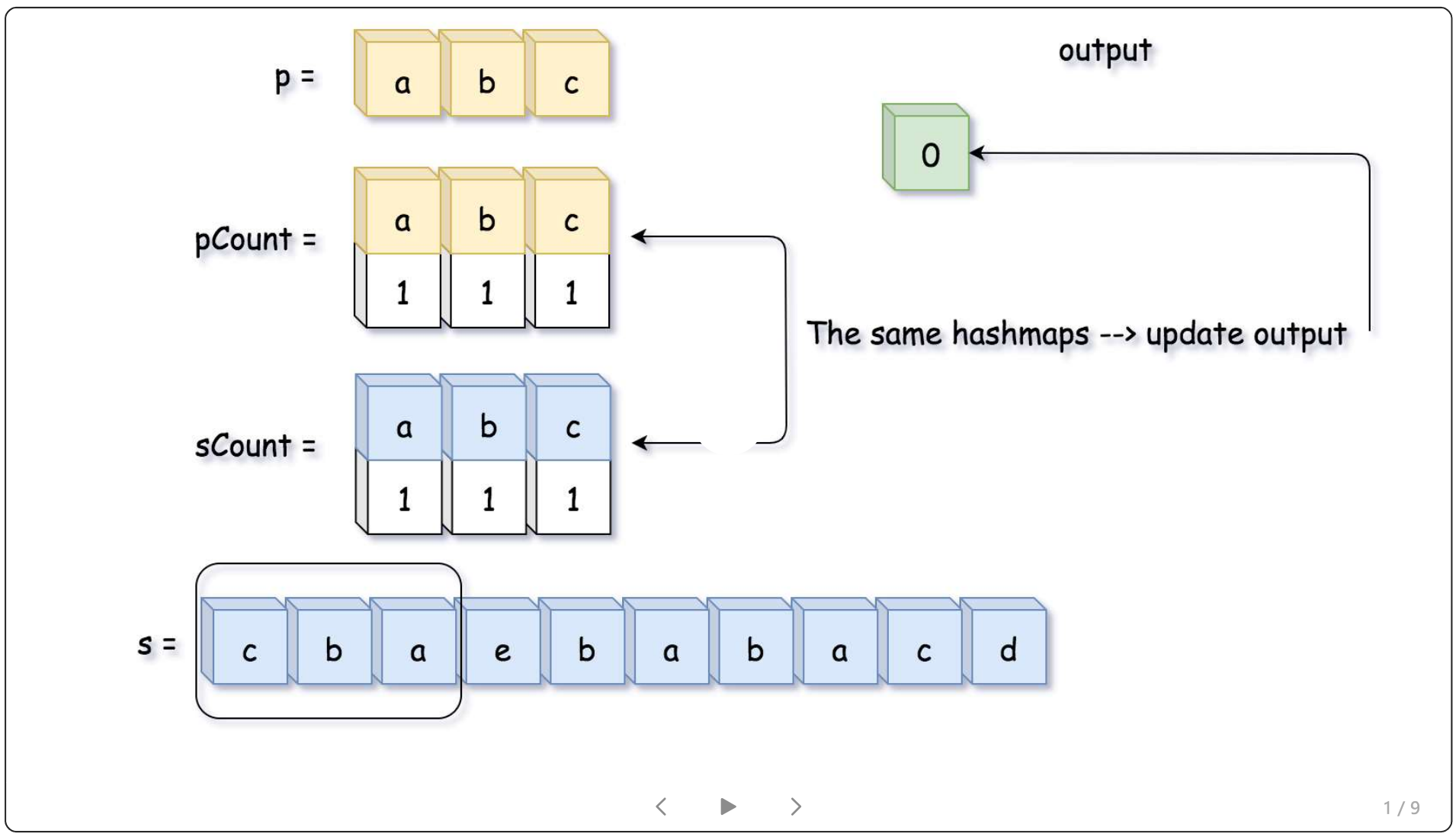
Let's start from the simplest approach: sliding window + two counter hashmaps `letter -> its count`. The first hashmap is a reference counter `pCount` for string `p`, and the second one is a counter `sCount` for string in the sliding window.

The idea is to move sliding window along the string `s`, recompute the second hashmap `sCount` in a constant time and compare it with the first hashmap `pCount`. If `sCount == pCount`, then the string in the sliding window is a permutation of string `p`, and one could add its start position in the output list.

Algorithm

- Build reference counter `pCount` for string `p` .
- Move sliding window along the string `s` :
  - Recompute sliding window counter `sCount` at each step by adding one letter on the right and removing one letter on the left.
  - If `sCount == pCount` , update the output list.
- Return output list.

Implementation



Java Python Copy

```
10      if (pCount.containsKey(ch)) {
11        pCount.put(ch, pCount.get(ch) + 1);
12      }
13      else {
14        pCount.put(ch, 1);
15      }
16    }
17
18    List<Integer> output = new ArrayList();
19    // sliding window on the string s
20    for (int i = 0; i < ns; ++i) {
21      // add one more letter
22      // on the right side of the window
23      char ch = s.charAt(i);
24      if (sCount.containsKey(ch)) {
25        sCount.put(ch, sCount.get(ch) + 1);
26      }
27      else {
28        sCount.put(ch, 1);
29      }
30      // remove one letter
31      // from the left side of the window
32      if (i >= np) {
33        ch = s.charAt(i - np);
34        if (sCount.get(ch) == 1) {
35          sCount.remove(ch);
36        }
```

Complexity Analysis

Let N_s and N_p be the length of `s` and `p` respectively. Let K be the maximum possible number of distinct characters. In this problem, K equals 26 because `s` and `p` consist of lowercase English letters.

- Time complexity: $O(N_s)$

  We perform one pass along each string when $N_s \geq N_p$ which costs $O(N_s + N_p)$ time. Since we only perform this step when $N_s \geq N_p$ the time complexity simplifies to $O(N_s)$.

- Space complexity: $O(K)$

  `pCount` and `sCount` will contain at most K elements each. Since K is fixed at 26 for this problem, this can be considered as $O(1)$ space.

Approach 2: Sliding Window with Array

Algorithm

Hashmap is quite complex structure, with known performance issues in Java. Let's implement approach 1 using 26-elements array instead of hashmap:

- Element number 0 contains count of letter `a`.
- Element number 1 contains count of letter `b`.
- ...
- Element number 25 contains count of letter `z`.

Algorithm

- Build reference array `pCount` for string `p`.
- Move sliding window along the string `s`:
  - Recompute sliding window array `sCount` at each step by adding one letter on the right and removing one letter on the left.
  - If `sCount == pCount`, update the output list.
- Return output list.

Implementation

Java | Python

Copy

```java
class Solution {
  public List<Integer> findAnagrams(String s, String p) {
    int ns = s.length(), np = p.length();
    if (ns < np) return new ArrayList();

    int [] pCount = new int[26];
    int [] sCount = new int[26];
    // build reference array using string p
    for (char ch : p.toCharArray()) {
      pCount[(int)(ch - 'a')]++;
    }

    List<Integer> output = new ArrayList();
    // sliding window on the string s
    for (int i = 0; i < ns; ++i) {
      // add one more letter
      // on the right side of the window
      sCount[(int)(s.charAt(i) - 'a')]++;
      // remove one letter
      // from the left side of the window
      if (i >= np) {
        sCount[(int)(s.charAt(i - np) - 'a')]--;
      }
      // compare array in the sliding window
      // with the reference array
      if (Arrays.equals(pCount, sCount)) {
        output.add(i - np + 1);
```

Complexity Analysis

Let N_s and N_p be the length of `s` and `p` respectively. Let K be the maximum possible number of distinct characters. In this problem, K equals 26 because `s` and `p` consist of lowercase English letters.

- Time complexity: $O(N_s)$

  We perform one pass along each string when $N_s \geq N_p$ which costs $O(N_s + N_p)$ time. Since we only perform this step when $N_s \geq N_p$ the time complexity simplifies to $O(N_s)$.

- Space complexity: $O(K)$

  `pCount` and `sCount` contain K elements each. Since K is fixed at 26 for this problem, this can be considered as $O(1)$ space.

Share

Comments (81)

Sort by: **Best**

Type comment here... (Markdown supported)

Preview Comment

Article Commenting Rules

1. This comment section is for questions and comments regarding this **LeetCode article**. All posts must respect our LeetCode Community Rules.

2. Concerns about errors or bugs in the article, problem description, or test cases should be posted on LeetCode Feedback, so that our team can address them.

bshaibu Feb 07, 2020

It's probably worth noting: we can compare p_count and s_count in constant time because they are both at most size 26 (as they only contain the 26 lowercase characters). This makes comparing an O(26) operation i.e. O(1).

If we didn't have a bound on the number of input characters we might want to consider a more complex comparison scheme (a count of satisfied characters or a set of satisfied characters to accompany the maps, etc.)

254 Show 10 Replies Reply

harsh014 Feb 23, 2020

Why is this solution O(S + P)? Shouldn't it just be O(S) since S will always be larger than P?

50 Show 6 Replies Reply

asperas Apr 21, 2020

Should time complexity be O(string.size() * alphabet.size()) ? because a comparison of two arrays/hashmaps is not const time operation . Sure, if we assume that our alphabet is always English alphabet then it's O(string.size() * 26) == O(string.size())

38 Show 1 Replies Reply

Fizzy19 Feb 02, 2022

24 Show 2 Replies Reply

awsmankitakalra May 05, 2020

The test case has one case where length of string is 32764. My code is passing is for all testcases apart from that. Additionally, the question says that the length does not exceed by 20100. I think the testcases should be revised. @liaison and @andvary

16 Reply

monhoshum Feb 27, 2020

Why is the runtime O(S + P)? In the for loop, it compares the p_count and s_count, which has a runtime of O(P). should the runtime be O(P + SP) = O(SP)?

30 Show 5 Replies Reply

ntkw Apr 15, 2020

I'm not convinced about the reported time complexity. isn't time complexity NxP where P is the number of unique characters in p String? once hashmap equals is time complexity O(n)? Or Am I mis interpretating? Thank you

7 Show 1 Replies Reply

MichiganHackers Jul 22, 2020

Though solvable, this problem should probably be marked as hard.

25 Show 1 Replies Reply

doct0rX Sep 25, 2021

Read more

4 Reply

vemantio Feb 02, 2022

Read more

2 Reply