438. Find All Anagrams in a String

Feb. 3, 2020 | 413K views

Average Rating: 4.43 (21 votes)

Given a string **s** and a **non-empty** string **p**, find all the start indices of **p**'s anagrams in **s**.

Strings consists of lowercase English letters only and the length of both strings **s** and **p** will not be larger than 20,100.

The order of output does not matter.

Example 1:

```
Input:
s: "cbaebabacd" p: "abc"

Output:
[0, 6]

Explanation:
The substring with start index = 0 is "cba", which is an anagram of "abc".
The substring with start index = 6 is "bac", which is an anagram of "abc".
```

Example 2:

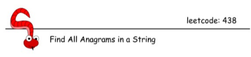
```
Input:
s: "abab" p: "ab"

Output:
[0, 1, 2]

Explanation:
The substring with start index = 0 is "ab", which is an anagram of "ab".
The substring with start index = 1 is "ba", which is an anagram of "ab".
The substring with start index = 2 is "ab", which is an anagram of "ab".
```

Solution

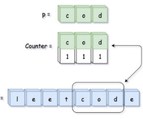
Solution Template



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

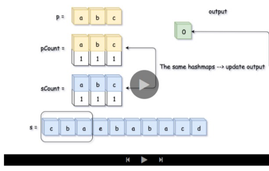
Let's start from the simplest approach: sliding window + two counter hashmaps `letter -> its count`. The first hashmap is a reference counter `pCount` for string `p`, and the second one is a counter `sCount` for string in the sliding window.

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 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

```
def findAnagrams(self, s: str, p: str) -> List[int]:
    ns, np = len(s), len(p)
    if ns < np:
        return []

    p_count = Counter(p)
    s_count = Counter()

    output = []
    # sliding window on the string s
    for i in range(ns):
        # add one more letter
        # on the right side of the window
        s_count[s[i]] += 1
        # remove one letter
        # from the left side of the window
        if i >= np:
            if s_count[s[i - np]] == 1:
                del s_count[s[i - np]]
            else:
                s_count[s[i - np]] -= 1
        # compare array in the sliding window
        # with the reference array
        if p_count == s_count:
            output.append(i - np + 1)

    return output
```

Complexity Analysis

- Time complexity: $O(N_s + N_p)$ since it's one pass along both strings.
- Space complexity: $O(1)$, because `pCount` and `sCount` contain not more than 26 elements.

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 26 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

```
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        ns, np = len(s), len(p)
        if ns < np:
            return []

        p_count, s_count = [0] * 26, [0] * 26
        # build reference array using string p
        for ch in p:
            p_count[ord(ch) - ord('a')] += 1

        output = []
        # sliding window on the string s
        for i in range(ns):
            # add one more letter
            # on the right side of the window
            s_count[ord(s[i]) - ord('a')] += 1
            # remove one letter
            # from the left side of the window
            if i >= np:
                s_count[ord(s[i - np]) - ord('a')] -= 1
            # compare array in the sliding window
            # with the reference array
            if p_count == s_count:
                output.append(i - np + 1)

        return output
```

Complexity Analysis

- Time complexity: $O(N_s + N_p)$ since it's one pass along both strings.
- Space complexity: $O(1)$, because `pCount` and `sCount` contain 26 elements each.

Rate this article:

Previous Next