

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

leetcode: 438

Find All Anagrams in a String

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

```
3  def findAnagrams(self, s: str, p: str) -> List[int]:
4      ns, np = len(s), len(p)
5      if ns < np:
6          return []
7
8      p_count = Counter(p)
9      s_count = Counter()
10
11     output = []
12     # sliding window on the string s
13     for i in range(ns):
14         # add one more letter
15         # on the right side of the window
16         s_count[s[i]] += 1
17         # remove one letter
18         # from the left side of the window
19         if i >= np:
20             if s_count[s[i - np]] == 1:
21                 del s_count[s[i - np]]
22             else:
23                 s_count[s[i - np]] -= 1
24         # compare array in the sliding window
25         # with the reference array
26         if p_count == s_count:
27             output.append(i - np + 1)
28
29     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

```
1  class Solution:
2      def findAnagrams(self, s: str, p: str) -> List[int]:
3          ns, np = len(s), len(p)
4          if ns < np:
5              return []
6
7          p_count, s_count = [0] * 26, [0] * 26
8          # build reference array using string p
9          for ch in p:
10             p_count[ord(ch) - ord('a')] += 1
11
12     output = []
13     # sliding window on the string s
14     for i in range(ns):
15         # add one more letter
16         # on the right side of the window
17         s_count[ord(s[i]) - ord('a')] += 1
18         # remove one letter
19         # from the left side of the window
20         if i >= np:
21             s_count[ord(s[i - np]) - ord('a')] -= 1
22         # compare array in the sliding window
23         # with the reference array
24         if p_count == s_count:
25             output.append(i - np + 1)
26
27     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: ★★★★★

Comments: 23

Sort By ▾

Type comment here... (Markdown is supported)

Preview

Post

bshaibu ★175 · February 6, 2020 9:53 PM

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

Read More

56

Share

Reply

SHOW 4 REPLIES

harsh014 ★61 · February 23, 2020 4:21 AM

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

26

Share

Reply

SHOW 4 REPLIES

monhoshum ★15 · February 27, 2020 10:45 AM

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)?

14

Share

Reply

SHOW 4 REPLIES

asperas ★183 · April 21, 2020 7:14 PM

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())

6

Share

Reply

ntkow ★58 · April 14, 2020 8:54 PM

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

3

Share

Reply

asish_cse ★4 · April 6, 2020 9:12 PM

Time complexity should be O(S + P) or O(S) as S will always be larger than P?

2

Share

Reply

SHOW 1 REPLY

CoderJoe ★702 · February 5, 2020 7:13 PM

public List findAnagrams(String s, String p) {
 List res = new ArrayList<>();
 if (s == null || s.length() == 0) return res;
 int[] pCount = new int[26];
 int[] sCount = new int[26];
 for (int i = 0; i < p.length(); i++)
 pCount[p.charAt(i) - 'a']++;
 for (int i = 0; i < s.length(); i++)
 sCount[s.charAt(i) - 'a']++;
 for (int i = 0; i < s.length() - p.length() + 1; i++)
 if (Arrays.equals(pCount, sCount))
 res.add(i);
 return res;
}

2

Share

Reply

SHOW 1 REPLY

user4717V ★1 · July 1, 2020 7:16 PM

instead of calculating both p_count and s_count we can keep track of the difference between s and p and add the index to the output everytime the difference array is all 0. This way the comparison of p_count == s_count is faster since we have to check the array is empty in O(1) instead of comparing all of the character counts worst case in O(p).

1

Share

Reply

wengkeat575 ★0 · May 20, 2020 5:37 AM

Can someone explained: Why is the space complexity is O(1) in sliding window with array? When you create new array and using it to store the count, won't it be using the space?

1

Share

Reply

SHOW 1 REPLY

venk07 ★18 · July 13, 2020 6:47 PM

Hi, not able to wrap my head around why 2nd code snippet has better performance than the 1st. Pasting the entire code below

1

Share

Reply

Read More

0

Share

Reply

1

2

3