

Solution Review: Problem Challenge 2

We'll cover the following ^

- String Anagrams (hard)
- Solution
 - Code
 - Time Complexity
 - Space Complexity

String Anagrams (hard)

Given a string and a pattern, find **all anagrams of the pattern in the given string**.

Anagram is actually a **Permutation** of a string. For example, "abc" has the following six anagrams:

1. abc
2. acb
3. bac
4. bca
5. cab
6. cba

Write a function to return a list of starting indices of the anagrams of the pattern in the given string.

Example 1:

Input: String="ppqp", Pattern="pq"

Output: [1, 2]

Explanation: The two anagrams of the pattern in the given string are "pq" and "qp".

Example 2:

Input: String="abbcabc", Pattern="abc"

Output: [2, 3, 4]

Explanation: The three anagrams of the pattern in the given string are "bca", "cab", and "abc".









This problem follows the **Sliding Window** pattern and is very similar to Permutation in a String

(<https://www.educative.io/collection/page/5668639101419520/5671464854355968/5401934796161024/>). In this problem, we need to find every occurrence of any permutation of the pattern in the string. We will use a list to store the starting indices of the anagrams of the pattern in the string.

Code

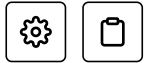
Here is what our algorithm will look like, only the highlighted lines have changed from Permutation in a String

(<https://www.educative.io/collection/page/5668639101419520/5671464854355968/5401934796161024/>):

 Java	 Python3	 C++	 JS
<pre>1 def find_string_anagrams(str1, pattern): 2 window_start, matched = 0, 0 3 char_frequency = {} 4 5 for chr in pattern: 6 if chr not in char_frequency: 7 char_frequency[chr] = 0 8 char_frequency[chr] += 1 9 10 result_indices = [] 11 # Our goal is to match all the characters from the 'char_frequency' with the current win 12 # try to extend the range [window_start, window_end] 13 for window_end in range(len(str1)): 14 right_char = str1[window_end] 15 if right_char in char_frequency: 16 # Decrement the frequency of matched character 17 char_frequency[right_char] -= 1 18 if char_frequency[right_char] == 0: 19 matched += 1 20 21 if matched == len(char_frequency): # Have we found an anagram? 22 result_indices.append(window_start) 23 24 # Shrink the sliding window 25 if window_end >= len(pattern) - 1: 26 left_char = str1[window_start] 27 window_start += 1 28 if left_char in char_frequency:</pre>			
<div></div>			

Time Complexity

The time complexity of the above algorithm will be $O(N + M)$ where 'N' and 'M' are the number of characters in the input string and the pattern respectively.



The space complexity of the algorithm is $O(M)$ since in the worst case, the whole pattern can have distinct characters which will go into the **HashMap**. In the worst case, we also need $O(N)$ space for the result list, this will happen when the pattern has only one character and the string contains only that character.

[← Back](#)[Next →](#)

Problem Challenge 2

Problem Challenge 3

 **Mark as Completed**



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/solution-review-problem-challenge-2__pattern-sliding-window__grokking-the-coding-interview-patterns-for-coding-questions)

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.