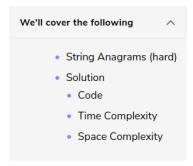




# Solution Review: Problem Challenge 2



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

## Solution

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

```
Python3
                  ⓒ C++
   find string_anagrams(str1, pattern):
window start, matched = 0, 0
char_frequency = {}
for chr in pattern:
  if chr not in char_frequency:
```

```
char frequency[chr] += 1
      result_indices = []
      for window end in range(len(strl)):
        right char = str1[window end]
        if right_char in char_frequency:
          char_frequency[right_char] -= 1
          if char_frequency[right_char] == 0:
            matched += 1
        if matched == len(char_frequency): # Have we found an anagram?
          result_indices.append(window_start)
        if window end >= len(pattern) - 1:
          left_char = str1[window_start]
          window start += 1
          if left_char in char_frequency:
            if char_frequency[left_char] == 0:
    matched -= 1 # Before putting the character back, decrement the matched count
            char_frequency[left_char] += 1 # Put the character back
33
34
36 def main():
37 print(fine
      print(find_string_anagrams("ppqp", "pq"))
      print(find_string_anagrams("abbcabc", "abc"))
    main()
Run
                                                                                                     Reset []
```

## 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.

## Space Complexity

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.

