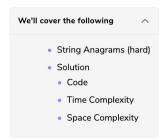




# Solution Review: Problem Challenge 2



# 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"
Output: [2, 3, 4]
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:

```
Pvthon3
                    @ C++
  mport java.util.*;
class StringAnagrams {
  public static List<Integer> findStringAnagrams(String str, String pattern) {
    int windowStart = 0, matched = 0;
    Map<Character, Integer> charFrequencyMap = new HashMap<>();
    for (char chr : pattern.toCharArray())
      charFrequencyMap.put(chr, charFrequencyMap.getOrDefault(chr, 0) + 1);
    List<Integer> resultIndices = new ArrayList<Integer>();
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {</pre>
      char rightChar = str.charAt(windowEnd);
      if (charFrequencyMap.containsKey(rightChar)) {
        charFrequencyMap.put(rightChar, charFrequencyMap.get(rightChar) - 1);
        if (charFrequencyMap.get(rightChar) == 0)
          matched++;
```

## **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.

