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 "a
bc".
```

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:

```
using namespace std;

#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

class StringAnagrams {
 public:
  static vector<int> findStringAnagrams(const string &str, const string &pattern) {
    int windowStart = 0, matched = 0;
    unordered_map<char, int> charFrequencyMap;
    for (auto chr : pattern) {
      charFrequencyMap[chr]++;
    }

    vector<int> resultIndices;
    // our goal is to match all the characters from the map with the current window
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      // decrement the frequency of the matched character
      if (charFrequencyMap.find(rightChar) != charFrequencyMap.end()) {
        charFrequencyMap[rightChar]--;
        if (charFrequencyMap[rightChar] == 0) {
          matched++;
        }
      }

      if (matched == (int)charFrequencyMap.size()) {  // have we found an anagram?
        resultIndices.push_back(windowStart);
      }

      if (windowEnd >= pattern.length() - 1) {  // shrink the window
        char leftChar = str[windowStart++];
        if (charFrequencyMap.find(leftChar) != charFrequencyMap.end()) {
          if (charFrequencyMap[leftChar] == 0) {
            matched--;  // before putting the character back, decrement the matched count
          }
          // put the character back
          charFrequencyMap[leftChar]++;
        }
      }
    }

    return resultIndices;
  }
};

int main(int argc, char *argv[]) {
  auto result = StringAnagrams::findStringAnagrams("ppqp", "pq");
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = StringAnagrams::findStringAnagrams("abbcabc", "abc");
  for (auto num : result) {
    cout << num << " ";
```

```
62
```

Output 1.327s

```
1 2
2 3 4
```

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.