Solution Review: Problem Challenge 3

Smallest Window containing Substring (hard)

Given a string and a pattern, find the **smallest substring** in the given string which has **all the characters of the given pattern**.

Example 1:

```
Input: String="aabdec", Pattern="abc"
Output: "abdec"
Explanation: The smallest substring having all characters of the pattern is "abdec"
```

Example 2:

```
Input: String="abdabca", Pattern="abc"
Output: "abc"
Explanation: The smallest substring having all characters of the pattern is "abc".
```

Example 3:

```
Input: String="adcad", Pattern="abc"
Output: ""
Explanation: No substring in the given string has all characters of the pattern.
```

Solution

This problem follows the **Sliding Window** pattern and has a lot of similarities with Permutation in a String with one difference. In this problem, we need to find a substring having all characters of the pattern which means that the required substring can have some additional characters and doesn’t need to be a permutation of the pattern. Here is how we will manage these differences:

1. We will keep a running count of every matching instance of a character.
2. Whenever we have matched all the characters, we will try to shrink the window from the beginning, keeping track of the smallest substring that has all the matching characters.
3. We will stop the shrinking process as soon as we remove a matched character from the sliding window. One thing to note here is that we could have redundant matching characters, e.g., we might have two ‘a’ in the sliding window when we only need one ‘a’. In that case, when we encounter the first ‘a’, we will simply shrink the window without decrementing the matched count. We will decrement the matched count when the second ‘a’ goes out of the window.

Code

Here is how our algorithm will look; only the highlighted lines have changed from Permutation in a String:

Java Python3 C++ JS

```cpp
using namespace std;

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

class MinimumWindowSubstring {
 public:
  static string findSubstring(const string &str, const string &pattern) {
    int windowStart = 0, matched = 0, minLength = str.length() + 1, subStrStart = 0;
    unordered_map<char, int> charFrequencyMap;
    for (auto chr : pattern) {
      charFrequencyMap[chr]++;
    }

    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      if (charFrequencyMap.find(rightChar) != charFrequencyMap.end()) {
        charFrequencyMap[rightChar]--;
        if (charFrequencyMap[rightChar] >= 0) {  // count every matching of a character
          matched++;
        }
      }

      // shrink the window if we can, finish as soon as we remove a matched character
      while (matched == pattern.length()) {
        if (minLength > windowEnd - windowStart + 1) {
          minLength = windowEnd - windowStart + 1;
          subStrStart = windowStart;
        }

        char leftChar = str[windowStart++];
        if (charFrequencyMap.find(leftChar) != charFrequencyMap.end()) {
          // note that we could have redundant matching characters, therefore we'll decrement the
          // matched count only when a useful occurrence of a matched character is going out of the
          // window
          if (charFrequencyMap[leftChar] == 0) {
            matched--;
          }
          charFrequencyMap[leftChar]++;
        }
      }
    }

    return minLength > str.length() ? "" : str.substr(subStrStart, minLength);
  }
};

int main(int argc, char *argv[]) {
  cout << MinimumWindowSubstring::findSubstring("aabdec", "abc") << endl;
  cout << MinimumWindowSubstring::findSubstring("abdabca", "abc") << endl;
  cout << MinimumWindowSubstring::findSubstring("adcad", "abc") << endl;
}
```

Output 1.076s

```
abdec
abc
```

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 resulting substring, which will happen when the input string is a permutation of the pattern.