

# Grokking the Coding Interview: Patterns for Coding Questions



## Pattern: Two Pointers

Problem Challenge 4

Solution Review: Problem Challenge 4

Challenge 3

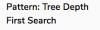


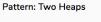
## Pattern: Merge Intervals









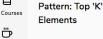




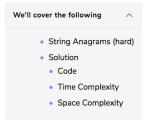








# Solution Review: Problem Challenge 2



### 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. abo
- 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

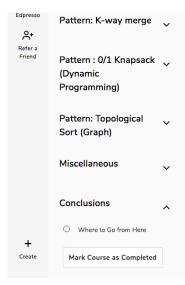
This problem follows the **Sliding Window** pattern and is very similar to Permutation in a String. In this problem, we need to find every occurrence of any permutation of the pattern in the string. We will use a list to store the starting indices of the anagrams of the pattern in the string.

### Code

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

```
👙 Java
            Python3
                         G C++
        ction find_string_anagrams(str, pattern)
       let windowStart = 0,
         matched = 0,
         charFrequency = {};
       for (i = 0; i < pattern.length; i++) {
         if (!(chr in charFrequency)) {
  charFrequency[chr] = 0;
         charFrequency[chr] += 1;
      const resultIndices = [];
                    is to match all the characters from the 'charFrequency' with the current window
       for (windowEnd = 0; windowEnd < str.length; windowEnd++) {</pre>
         const rightChar = str[windowEnd];
         if (rightChar in charFrequency) {
           charFrequency[rightChar] -= 1;
           if (charFrequency[rightChar] === 0) {
             matched += 1;
         if (matched === Object.keys(charFrequency).length) { // have we found an anagram?
           resultIndices.push(windowStart):
                                                                                         SAVE
                                                                                                    RESET
                                                                                                              0
```

Time Complexity



The time complexity of the above agorithm with be O(1) + 10, where is and in the manuscrot 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.

