

## Grokking the Coding Interview: Patterns for Coding Questions

4% completed

### Introduction

- Who should take this course?
- Course Overview

### Pattern: Sliding Window

- Introduction
- Maximum Sum Subarray of Size K (easy)
- Smallest Subarray with a given sum (easy)
- Longest Substring with K Distinct Characters (medium)
- Fruits into Baskets (medium)
- No-repeat Substring (hard)
- Longest Substring with Same Letters after Replacement (hard)
- Longest Subarray with Ones after Replacement (hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1**
- Problem Challenge 2
- Solution Review: Problem Challenge 2
- Problem Challenge 3
- Solution Review: Problem Challenge 3
- Problem Challenge 4
- Solution Review: Problem Challenge 4

### Pattern: Two Pointers

### Pattern: Fast & Slow pointers

### Pattern: Merge Intervals

### Pattern: Cyclic Sort

### Pattern: In-place Reversal of a LinkedList

### Pattern: Tree Breadth First Search

### Pattern: Tree Depth First Search

### Pattern: Two Heaps

### Pattern: Subsets

### Pattern: Modified

## Solution Review: Problem Challenge 1

### We'll cover the following

- Permutation in a String (hard)
- Solution
- Code
  - Time Complexity
  - Space Complexity

### Permutation in a String (hard) #

Given a string and a pattern, find out if the **string contains any permutation of the pattern**.

**Permutation** is defined as the re-arranging of the characters of the string. For example, "abc" has the following six permutations:

1. abc
2. acb
3. bac
4. bca
5. cab
6. cba

If a string has 'n' distinct characters it will have  $n!$  permutations.

#### Example 1:

```
Input: String="oidbcaf", Pattern="abc"
Output: true
Explanation: The string contains "bca" which is a permutation of the given pattern.
```

#### Example 2:

```
Input: String="odicf", Pattern="dc"
Output: false
Explanation: No permutation of the pattern is present in the given string as a substring.
```

#### Example 3:

```
Input: String="bcdxabcdy", Pattern="bcdyabcdx"
Output: true
Explanation: Both the string and the pattern are a permutation of each other.
```

#### Example 4:

```
Input: String="aaacb", Pattern="abc"
Output: true
Explanation: The string contains "acb" which is a permutation of the given pattern.
```

### Solution #

This problem follows the **Sliding Window** pattern and we can use a similar sliding window strategy as discussed in [Longest Substring with K Distinct Characters](#). We can use a **HashMap** to remember the frequencies of all characters in the given pattern. Our goal will be to match all the characters from this **HashMap** with a sliding window in the given string. Here are the steps of our algorithm:

1. Create a **HashMap** to calculate the frequencies of all characters in the pattern.
2. Iterate through the string, adding one character at a time in the sliding window.
3. If the character being added matches a character in the **HashMap**, decrement its frequency in the map. If the character frequency becomes zero, we got a complete match.
4. If at any time, the number of characters matched is equal to the number of distinct characters in the pattern (i.e., total characters in the **HashMap**), we have gotten our required permutation.
5. If the window size is greater than the length of the pattern, shrink the window to make it equal to the size of the pattern. At the same time, if the character going out was part of the pattern, put it back in the frequency **HashMap**.

### Code #

Here is what our algorithm will look like:

```
Java Python3 C++ JS
1 function find_permutation(str, pattern) {
2   let windowStart = 0,
```

Binary Search

Pattern: Bitwise XOR

Pattern: Top 'K' Elements

Pattern: K-way merge

Pattern : 0/1 Knapsack (Dynamic Programming)

Pattern: Topological Sort (Graph)

Miscellaneous

Conclusions

Where to Go from Here

Mark Course as Completed

MW

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

```
3 matched = 0,
4 charFrequency = {};
5
6 for (i = 0; i < pattern.length; i++) {
7   const chr = pattern[i];
8   if (!(chr in charFrequency)) {
9     charFrequency[chr] = 0;
10  }
11  charFrequency[chr] += 1;
12 }
13
14 // Our goal is to match all the characters from the 'charFrequency' with the current window
15 // try to extend the range [windowStart, windowEnd]
16 for (windowEnd = 0; windowEnd < str.length; windowEnd++) {
17   const rightChar = str[windowEnd];
18   if (rightChar in charFrequency) {
19     // Decrement the frequency of matched character
20     charFrequency[rightChar] -= 1;
21     if (charFrequency[rightChar] === 0) {
22       matched += 1;
23     }
24   }
25 }
26 if (matched === Object.keys(charFrequency).length) {
27   return true;
28 }
```

RUN

SAVE

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**.

← Back

Problem Challenge 1

MARK AS COMPLETED

Next →

Problem Challenge 2

[Report an Issue](#)