

Grokking the Coding Interview: Patterns for Coding Questions

2% 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

No-repeat Substring (hard)

We'll cover the following

- Problem Statement
- Try it yourself
- Solution
- Code
 - Time Complexity
 - Space Complexity

Problem Statement

Given a string, find the length of the longest substring which has **no repeating characters**.

Example 1:

```
Input: String="aabccbb"
Output: 3
Explanation: The longest substring without any repeating characters is "abc".
```

Example 2:

```
Input: String="abbbb"
Output: 2
Explanation: The longest substring without any repeating characters is "ab".
```

Example 3:

```
Input: String="abccde"
Output: 3
Explanation: Longest substrings without any repeating characters are "abc" & "cde".
```

Try it yourself

Try solving this question here:

JavaPython3JS C++

```
1 const non_repeat_substring = function(str) {
2   // TODO: Write your code here
3   return -1;
4 };
5
```

TESTSAVERESET

Solution

This problem follows the **Sliding Window** pattern and we can use a similar dynamic sliding window strategy as discussed in [Longest Substring with K Distinct Characters](#). We can use a **HashMap** to remember the last index of each character we have processed. Whenever we get a repeating character we will shrink our sliding window to ensure that we always have distinct characters in the sliding window.

Code

Here is what our algorithm will look like:

JavaPython3C++JS

```
1 function non_repeat_substring(str) {
2   let windowStart = 0,
3     maxLength = 0,
4     charIndexMap = {};
5
6   // try to extend the range [windowStart, windowEnd]
7   for (let windowEnd = 0; windowEnd < str.length; windowEnd++) {
8     const rightChar = str[windowEnd];
9     // if the map already contains the 'rightChar', shrink the window from the beginning so that
10    // we have only one occurrence of 'rightChar'
11    if (rightChar in charIndexMap) {
12      // this is tricky; in the current window, we will not have any 'rightChar' after its previous index
13      // and if 'windowStart' is already ahead of the last index of 'rightChar', we'll keep 'windowStart'
14      windowStart = Math.max(windowStart, charIndexMap[rightChar] + 1);
15    }
16    // insert the 'rightChar' into the map
17    charIndexMap[rightChar] = windowEnd;
18    // remember the maximum length so far
19    maxLength = Math.max(maxLength, windowEnd - windowStart + 1);
20  }
21  return maxLength;
22 }
23
```

MW

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

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

24

25 console.log('Length of the longest substring: \${non_repeat_substring('aabccbb')}');

26 console.log('Length of the longest substring: \${non_repeat_substring('abbbb')}');

27 console.log('Length of the longest substring: \${non_repeat_substring('abcde')}');

RUN

SAVE

RESET

Time Complexity

The time complexity of the above algorithm will be $O(N)$ where 'N' is the number of characters in the input string.

Space Complexity

The space complexity of the algorithm will be $O(K)$ where K is the number of distinct characters in the input string. This also means $K \leq N$, because in the worst case, the whole string might not have any repeating character so the entire string will be added to the **HashMap**. Having said that, since we can expect a fixed set of characters in the input string (e.g., 26 for English letters), we can say that the algorithm runs in fixed space $O(1)$; in this case, we can use a fixed-size array instead of the **HashMap**.

← Back

MARK AS COMPLETED

Next →

Fruits into Baskets (medium)

Longest Substring with Same Letters ...

Report an Issue