Longest Substring with Distinct Characters (hard)

We'll cover the following

Problem Statement#

Given a string, find the **length of the longest substring**, which has all **distinct characters**.

Example 1:

```
Input: String="aabccbb"
Output: 3
Explanation: The longest substring with distinct characters is "abc".
```

Example 2:

```
Input: String="abbbb"
Output: 2
```

```
Explanation: The longest substring with distinct character
```

Example 3:

```
Input: String="abccde"
Output: 3
Explanation: Longest substrings with distinct characters are "abc" & "cd
e".
```

Try it yourself#

Try solving this question here:

C++

```
using namespace std;

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

class NoRepeatSubstring {
 public:
  static int findLength(const string& str) {
    int maxLength = 0;
    // TODO: Write your code here
    return maxLength;
  }
};
```

Solution#

This problem follows the **Sliding Window** pattern, and we c e lla 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 duplicate 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:

C++

```
using namespace std;

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

class NoRepeatSubstring {
 public:
  static int findLength(const string& str) {
    int windowStart = 0, maxLength = 0;
    unordered_map<char, int> charIndexMap;
    // try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str[windowEnd];
      // if the map already contains the 'rightChar', shrink the window from
      // we have only one occurrence of 'rightChar'
      if (charIndexMap.find(rightChar) != charIndexMap.end()) {
        // this is tricky; in the current window, we will not have any 'righ
        // previous index and if 'windowStart' is already ahead of the last
        // we'll keep 'windowStart'
        windowStart = max(windowStart, charIndexMap[rightChar] + 1);
      }
      charIndexMap[rightChar] = windowEnd;  // insert the 'rightChar' into t
      maxLength =
          max(maxLength, windowEnd - windowStart + 1);  // remember the maxi
    }
```

Time Complexity#

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

Space Complexity#

The algorithm's space complexity will be $O(K)$, where K is the number of distinct characters in the input string. This also means $K <= N$, because in the worst case, the whole string might not have any duplicate 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

Fruits into Baskets (medium)

Next →

Longest Substring with Same Letters ...

☑ Mark as Completed

Report an Issue