No-repeat Substring (hard)

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:

Java | Python3 | JS | C++

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

```

TEST | SAVE | RESET

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:

Java | Python3 | C++ | JS

```
function non_repeat_substring(str) {
  let windowStart = 0,
    maxLength = 0,
    charIndexMap = {};

  // try to extend the range [windowStart, windowEnd]
  for (let windowEnd = 0; windowEnd < str.length; windowEnd++) {
    const rightChar = str[windowEnd];
    // if the map already contains the 'rightChar', shrink the window from the beginning so that
    // we have only one occurrence of 'rightChar'
    if (rightChar in charIndexMap) {
      // this is tricky; in the current window, we will not have any 'rightChar' after its previous index
      // and if 'windowStart' is already ahead of the last index of 'rightChar', we'll keep 'windowStart'
      windowStart = Math.max(windowStart, charIndexMap[rightChar] + 1);
    }
    // insert the 'rightChar' into the map
    charIndexMap[rightChar] = windowEnd;
    // remember the maximum length so far
    maxLength = Math.max(maxLength, windowEnd - windowStart + 1);
  }
  return maxLength;
}

```

Mark Course as Completed

```javascript

console.log(`Length of the longest substring: ${non_repeat_substring('aabccbb')}`);
console.log(`Length of the longest substring: ${non_repeat_substring('abbbb')}`);
console.log(`Length of the longest substring: ${non_repeat_substring('abccde')}`);
```

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 $K <= N$, 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**.

✓ MARK AS COMPLETED

← Back
Fruits into Baskets (medium)

Next →
Longest Substring with Same Letters ...

Report an Issue