

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:

Java

Python3

JS JS

C++

```
1 import java.util.*;
2
3 class NoRepeatSubstring {
4     public static int findLength(String str) {
5         // TODO: Write your code here
6         return -1;
7     }
8 }
9
```

TEST

SAVE

RESET

Show Results

Show Console

0 of 3 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✗	findLength(aabccbb)	3	-1	Incorrect Output
✗	findLength(abbbb)	2	-1	Incorrect Output
✗	findLength(abccde)	3	-1	Incorrect Output

3.497s

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 JS

```
1 import java.util.*;
2
3 class NoRepeatSubstring {
4     public static int findLength(String str) {
5         int windowStart = 0, maxLength = 0;
6         Map<Character, Integer> charIndexMap = new HashMap<>();
7         // try to extend the range [windowStart, windowEnd]
8         for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
9             char rightChar = str.charAt(windowEnd);
10            // if the map already contains the 'rightChar', shrink the window from the beginning so that
11            // we have only one occurrence of 'rightChar'
12            if (charIndexMap.containsKey(rightChar)) {
13                // this is tricky; in the current window, we will not have any 'rightChar' after its previous
14                // and if 'windowStart' is already ahead of the last index of 'rightChar', we'll keep 'windowStart'
15                windowStart = Math.max(windowStart, charIndexMap.get(rightChar) + 1);
16            }
17            charIndexMap.put(rightChar, windowEnd); // insert the 'rightChar' into the map
18            maxLength = Math.max(maxLength, windowEnd - windowStart + 1); // remember the maximum length so far
19        }
20
21        return maxLength;
22    }
23
24    public static void main(String[] args) {
25        System.out.println("Length of the longest substring: " + NoRepeatSubstring.findLength("aabccbb"));
26        System.out.println("Length of the longest substring: " + NoRepeatSubstring.findLength("abbbb"));
27        System.out.println("Length of the longest substring: " + NoRepeatSubstring.findLength("abccde"));
28    }
29 }
30
```

RUN

SAVE

RESET

Output

1.809s

Length of the longest substring: 3

Length of the longest substring: 2

Length of the longest substring: 3

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**.