Longest Substring with maximum K Distinct Characters (medium)

We'll cover the following

Problem Statement#

Given a string, find the length of the **longest substring** in it **with no more than** `K` **distinct characters**.

Example 1:

```
Input: String="araaci", K=2
Output: 4
Explanation: The longest substring with no more than '2' distinct charac
ters is "araa".
```

Example 2:

```
Input: String="araaci", K=1
Output: 2
Explanation: The longest substring with no more than '1' distinct charac
ters is "aa".
```

Example 3:

```
Input: String="cbbebi", K=3
Output: 5
Explanation: The longest substrings with no more than '3' distinct chara
cters are "cbbeb" & "bbebi".
```

Example 4:

```
Input: String="cbbebi", K=10
Output: 6
Explanation: The longest substring with no more than '10' distinct chara
cters is "cbbebi".
```

Try it yourself#

Try solving this question here:

C++

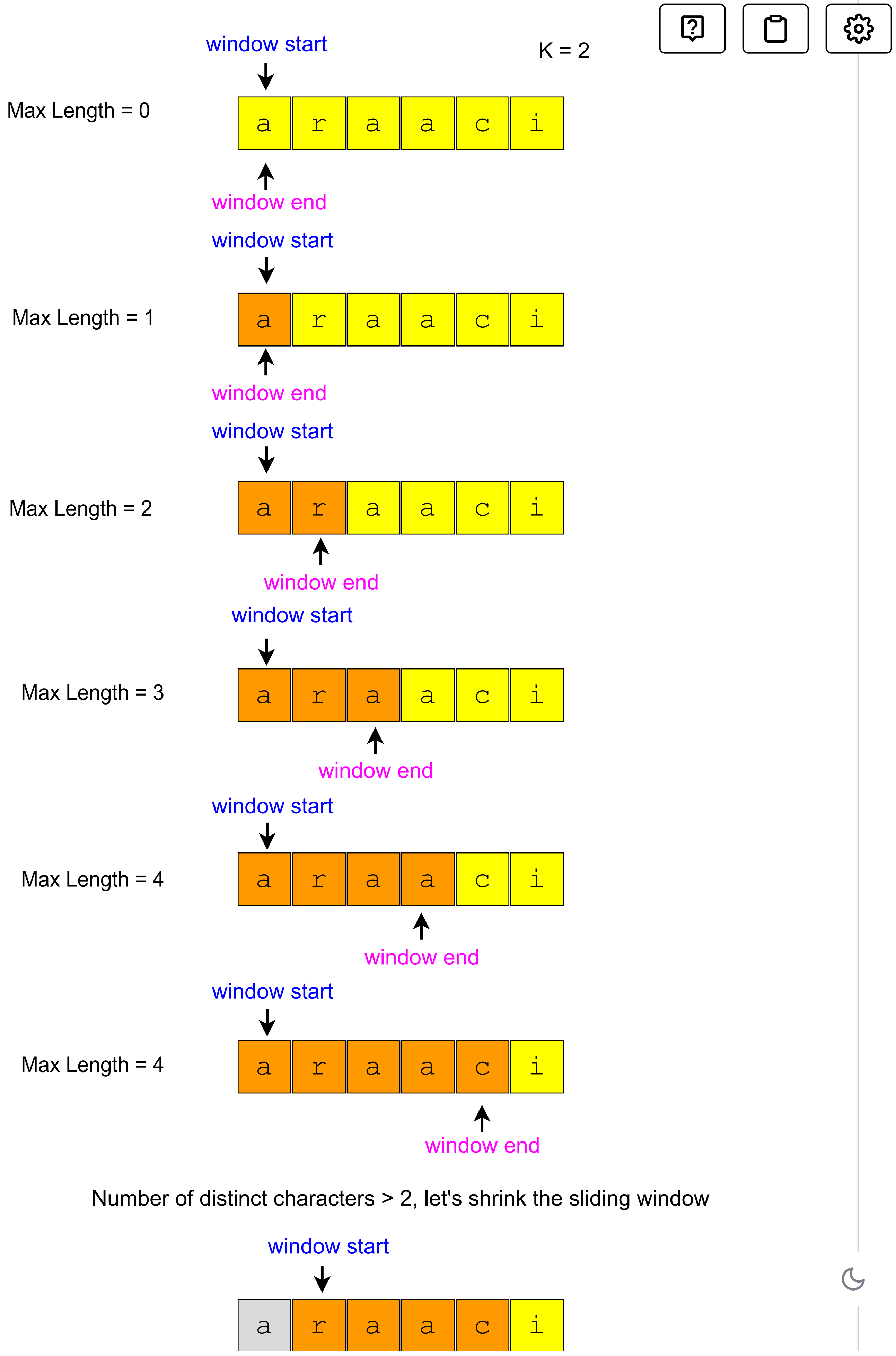
```cpp
using namespace std;

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

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

```

Solution#

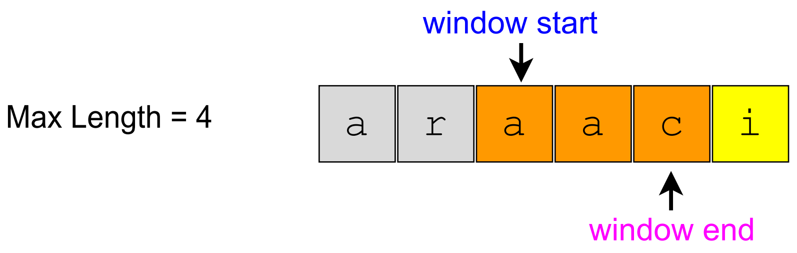
This problem follows the **Sliding Window** pattern, and we can use a similar dynamic sliding window strategy as discussed in Smallest Subarray With a Greater Sum. We can use a **HashMap** to remember the frequency of each character we have processed. Here is how we will solve this problem:

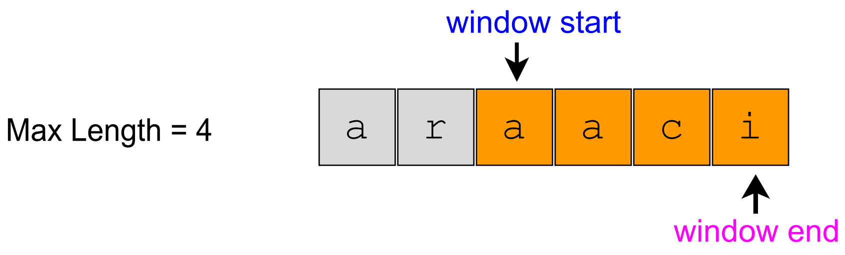
1. First, we will insert characters from the beginning of the string until we have `K` distinct characters in the **HashMap**.
2. These characters will constitute our sliding window. We are asked to find the longest such window having no more than `K` distinct characters. We will remember the length of this window as the longest window so far.
3. After this, we will keep adding one character in the sliding window (i.e., slide the window ahead) in a stepwise fashion.
4. In each step, we will try to shrink the window from the beginning if the count of distinct characters in the **HashMap** is larger than `K`. We will shrink the window until we have no more than `K` distinct characters in the **HashMap**. This is needed as we intend to find the longest window.
5. While shrinking, we'll decrement the character's frequency going out of the window and remove it from the **HashMap** if its frequency becomes zero.
6. At the end of each step, we'll check if the current window length is the longest so far, and if so, remember its length.

Here is the visual representation of this algorithm for the Example-1:

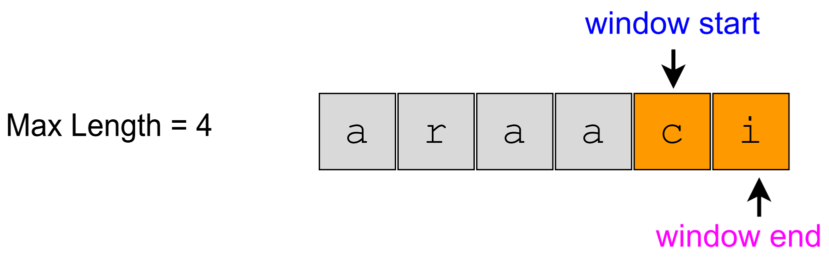

window start
K = 2
Max Length = 0
a r a a c i
window end
window start
Max Length = 1
a r a a c i
window end
window start
Max Length = 2
a r a a c i
window end
window start
Max Length = 3
a r a a c i
window end
window start
Max Length = 4
a r a a c i
window end
window start
Max Length = 4
a r a a c i
window end
Number of distinct characters > 2, let's shrink the sliding window
window start
a r a a c i

Number of distinct characters are still > 2, let's shrink the sliding window





Number of distinct character > 2, let's shrink the sliding window



Code#

Here is how our algorithm will look like:

C++

```
using namespace std;

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

class LongestSubstringKDistinct {
public:
  static int findLength(const string &str, int k) {
    int windowStart = 0, maxLength = 0;
    unordered_map<char, int> charFrequencyMap;
    // in the following loop we'll try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); window
      char rightChar = str[windowEnd];
      charFrequencyMap[rightChar]++;
      // shrink the sliding window, until we are left with 'k' distinct characters i
      // map
      while ((int)charFrequencyMap.size() > k) {
        char leftChar = str[windowStart];
        charFrequencyMap[leftChar]--;
        if (charFrequencyMap[leftChar] == 0) {
          charFrequencyMap.erase(leftChar);
        }
        windowStart++; // shrink the window
      }
      maxLength = max(maxLength, windowEnd - windowStart + 1); // remember the maxim
    }

    return maxLength;
  }
};
```

Time Complexity#

The above algorithm's time complexity will be $O(N)$, where N is the number of characters in the input string. The outer `for` loop runs for all characters, and the inner `while` loop processes each character only once; therefore, the time complexity of the algorithm will be $O(N + N)$, which is asymptotically equivalent to $O(N)$.

Space Complexity#

The algorithm's space complexity is $O(K)$, as we will be storing a maximum of $K + 1$ characters in the HashMap.

← Back

Next -

Smallest Subarray With a Greater Su...

k a ole

Report an Issue