Longest Substring with K Distinct Characters (medium)

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 characters is "araa".
```

Example 2:

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

Example 3:

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

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

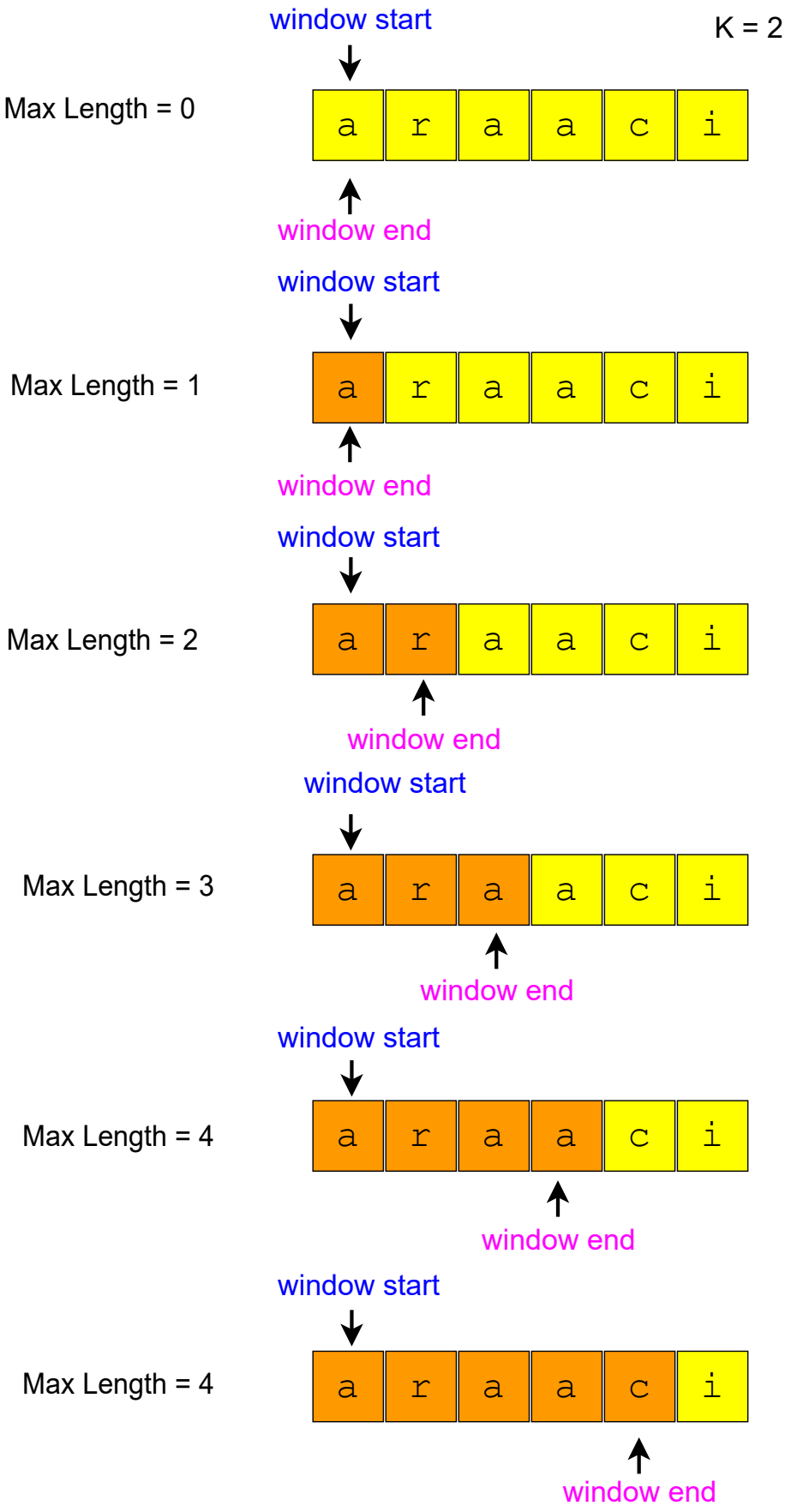
```
1  using namespace std;
2
3  #include <iostream>
4  #include <string>
5  #include <unordered_map>
6
7  class LongestSubstringKDistinct {
8   public:
9    static int findLength(const string& str, int k) {
10     int maxLength = 0;
11     // TODO: Write your code here
12     return maxLength;
13   }
14 };
15
```

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 given sum. We can use a **HashMap** to remember the frequency of each character we have processed. Here is how we will solve this problem:

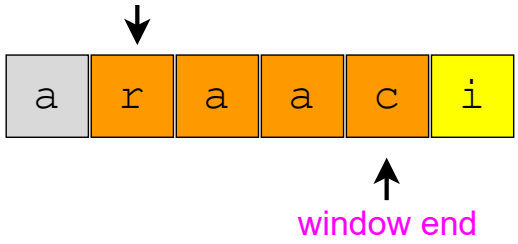
1. First, we will insert characters from the beginning of the string until we have 'K' distinct characters in the **HashMap**.
2. These characters will constitute our sliding window. We are asked to find the longest such window having no more than 'K' distinct characters. We will remember the length 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 frequency of the character 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:



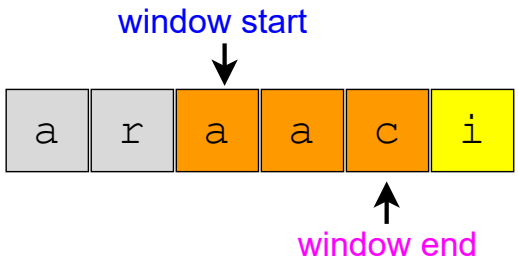
Number of distinct characters > 2, let's shrink the sliding window

window start

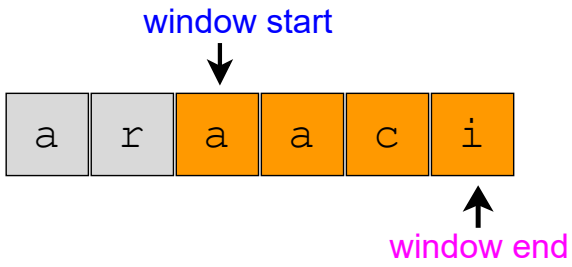


Number of distinct characters are still > 2, let's shrink the sliding window

Max Length = 4



Max Length = 4



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

Max Length = 4