Longest Substring with 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 c
haracters is "araa".
```

Example 2:

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

Example 3:

```
Input: String="cbbebi", K=3
Output: 5
Explanation: The longest substrings with no more than '3' distinc
t characters are "cbbeb" & "bbebi".
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```java
import java.util.*;

class LongestSubstringKDistinct {
  public static int findLength(String str, int k) {
    // TODO: Write your code here
    return -1;
  }
}
```

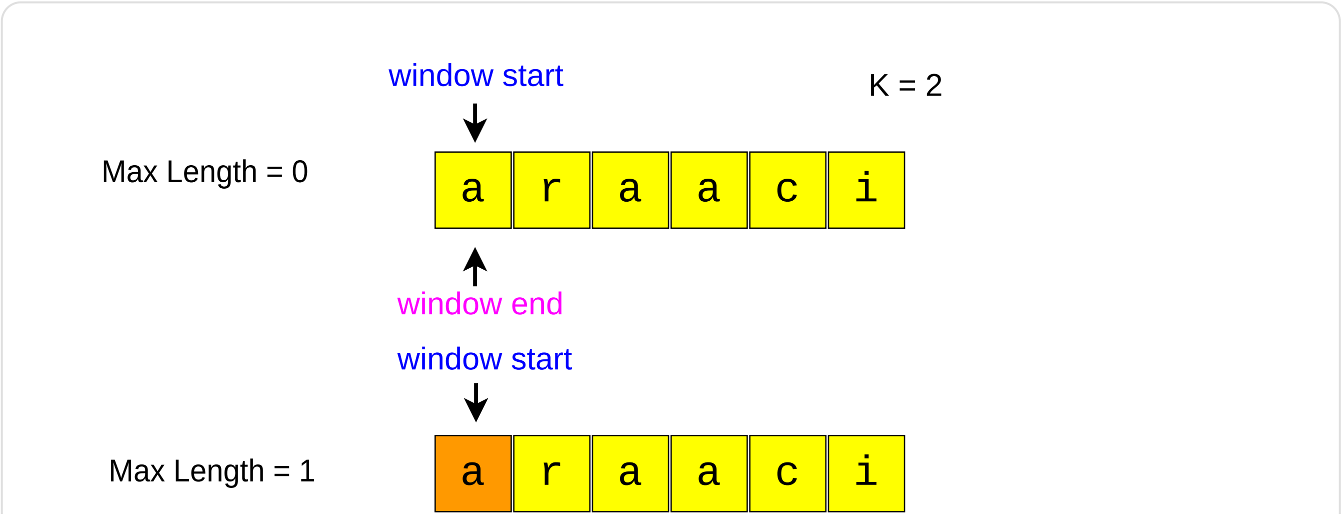
Solution

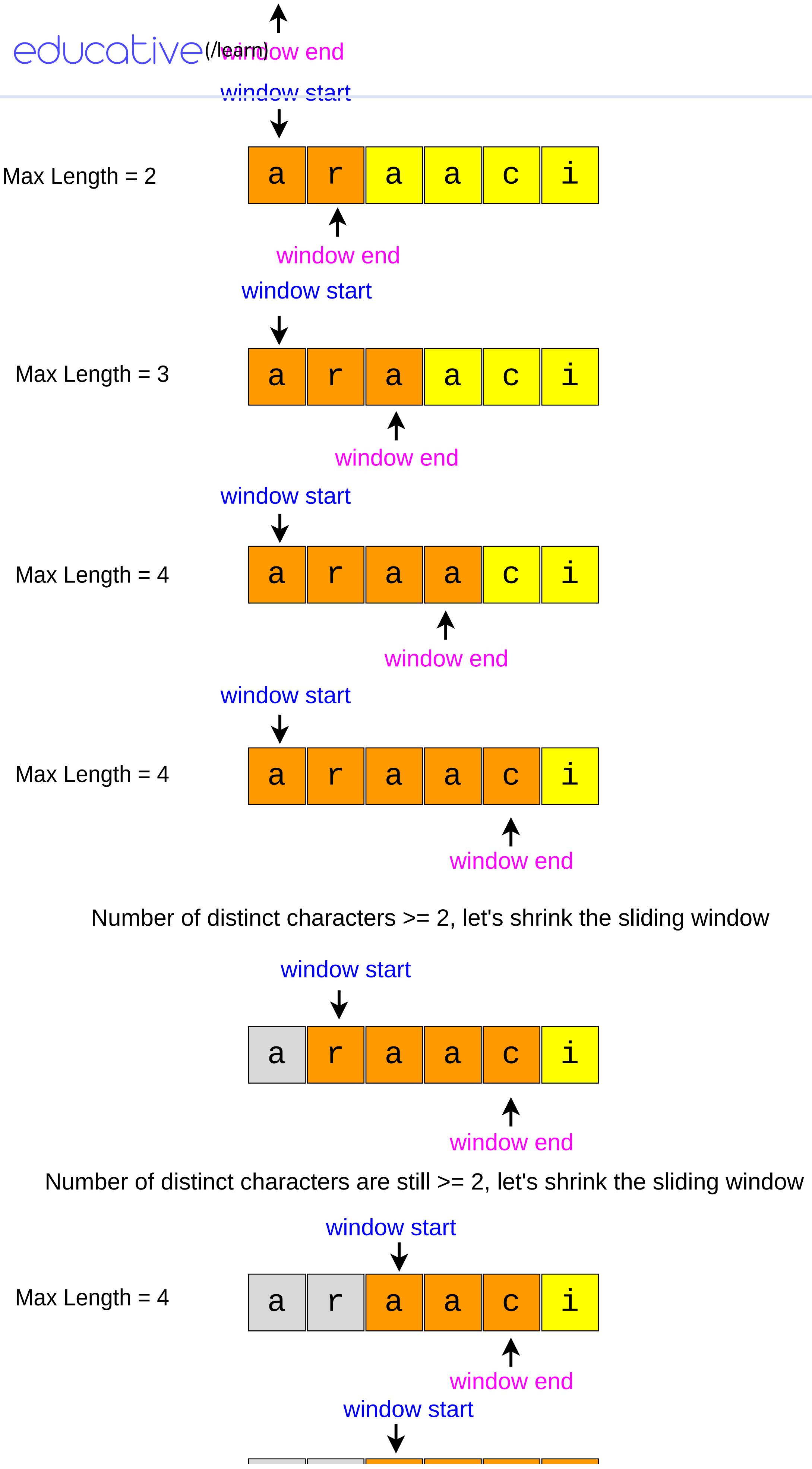
This problem follows the **Sliding Window** pattern and we can use a similar dynamic sliding window strategy as discussed in Smallest Subarray with a given sum

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

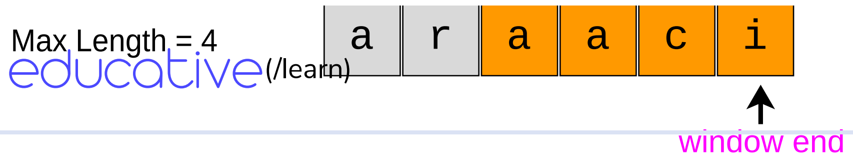
1. First, we will insert characters from the beginning of the string until we have ‘K’ distinct characters in the **HashMap**.
2. These characters will constitute our sliding window. We are asked to find the longest such window having no more than ‘K’ distinct characters. We will remember the length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

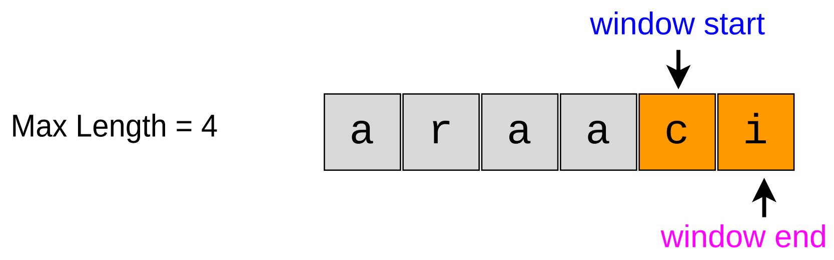



window end
window start
Max Length = 2
a r a a c i
window end
window start
Max Length = 3
a r a a c i
window end
window start
Max Length = 4
a r a a c i
window end
window start
Max Length = 4
a r a a c i
window end
Number of distinct characters >= 2, let's shrink the sliding window
window start
a r a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

(/learn)



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



Code

Here is how our algorithm will look:

Java | Python3 | C++ | JS

```
import java.util.*;

class LongestSubstringKDistinct {
  public static int findLength(String str, int k) {
    if (str == null || str.length() == 0 || str.length() < k)
      throw new IllegalArgumentException();

    int windowStart = 0, maxLength = 0;
    Map<Character, Integer> charFrequencyMap = new HashMap<>();
    // in the following loop we'll try to extend the range [windowStart, windowEnd]
    for (int windowEnd = 0; windowEnd < str.length(); windowEnd++) {
      char rightChar = str.charAt(windowEnd);
      charFrequencyMap.put(rightChar, charFrequencyMap.getOrDefault(rightChar, 0) + 1
      // shrink the sliding window, until we are left with 'k' distinct characters in
      while (charFrequencyMap.size() > k) {
        char leftChar = str.charAt(windowStart);
        charFrequencyMap.put(leftChar, charFrequencyMap.get(leftChar) - 1);
        if (charFrequencyMap.get(leftChar) == 0) {
          charFrequencyMap.remove(leftChar);
        }
        windowStart++; // shrink the window
      }
      maxLength = Math.max(maxLength, windowEnd - windowStart + 1); // remember the r
    }

    return maxLength;
  }

  public static void main(String[] args) {
    System.out.println("Length of the longest substring: " + LongestSubstringKDistinc
    System.out.println("Length of the longest substring: " + LongestSubstringKDistinc
    System.out.println("Length of the longest substring: " + LongestSubstringKDistinc
  }
}
```

Time Complexity

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

← Back (/courses/grokking-the-coding-interview/7XMlMEQPnnQ)
Smallest Subarray with a given sum (e...

Next (/courses/grokking-the-coding-interview/Bn2KLlOR0l
Completed
Fruits into Baskets (medium)