Solution: Longest Substring without Repeating Characters

Let's solve the Longest Substring without Repeating Characters problem using the Sliding Window pattern.

We'll cover the following

Statement

Given a string, `str`, return the length of the longest substring without repeating characters.

Constraints:

- $1 \leq$ `str.length` $\leq 5 \times 10^4$
- `str` consists of English letters, digits, symbols, and spaces.

Solution

So far, you've probably brainstormed some approaches on how to solve this problem. Let's explore some of these approaches and figure out which one to follow while considering time complexity and any implementation constraints.

Naive approach

The naive approach is to explore all possible substrings. For each substring, we check whether any character in it is repeated. After checking all the possible substrings, the substring with the longest length that satisfies the specified condition is returned.

The total time complexity of this solution is $O(n^3)$. To explore all possible substrings, the time complexity is $O(n^2)$, and to check whether all the characters in a substring are unique or not, the time complexity can be approximated to $O(n)$. So, the total time complexity is $O(n^2) * O(n) = O(n^3)$. The space complexity of this naive approach is $O(min(m, n))$, where m is the size of the character set and n is the size of the string.

Optimized approach using sliding window

We use a modified version of the classic sliding window method. Instead of a fixed-size window, we allow our window to grow, looking for the window that corresponds to the longest substring without repeating characters.

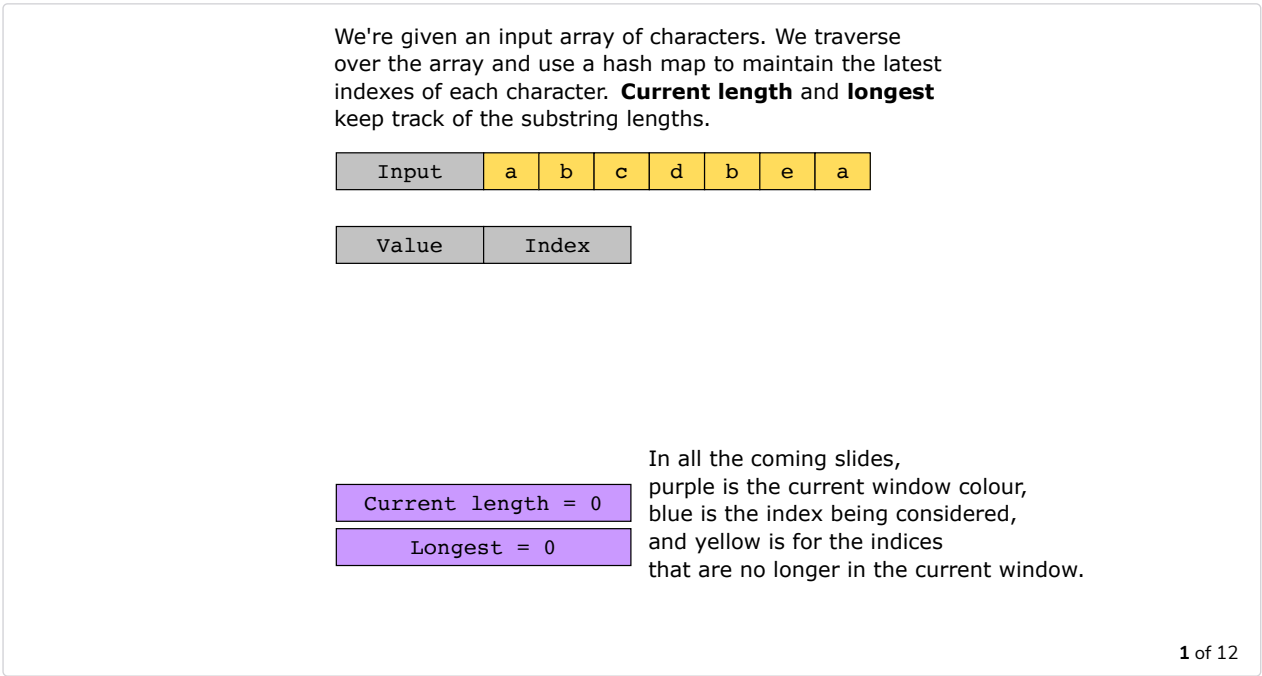
We initialize an empty hash map along with a variable to track character indices and the starting point of the window. Next, we traverse the string character by character. During each iteration, the current character is checked to see if it exists in the hash map. If it does not exist, it is added to the hash map along with its index.

However, if the current character already exists in the hash map and its index falls within the current window, it indicates that a repeating character has been discovered. In this case, the start of the window is updated to the previous location of the current element and incremented, and the length of the current window is calculated. The longest substring seen so far is updated if the length of the current window is greater than its current value. Finally, the length of the longest substring is returned as the result.

Here's how we implement this technique.

1. We initialize the following set of variables to `0` to keep track of the visited elements.
    - `windowStart`: The starting index of the current substring.
    - `windowLength`: The length of the current substring.
    - `longest`: The length of the longest substring.
2. For every element in the string, we check whether or not it's present in the hash map.
    - If it isn't present, we store it in the hash map such that the key is the current element and the value is its index in the string.
    - If it's already present in the hash map, then the element may have already appeared in the current substring. For this, we check if the previous occurrence of the element is before or after the starting index, `windowStart`, of the current substring.
        - If it's after `windowStart`, we calculate the current substring's length, `windowLength`, as the difference between the current index and `windowStart`. If `longest` is less than the new `windowLength`, we set `longest` as `windowLength`.
        - To prevent the repetition of the current element in the current window, the next candidate substring will be set to start from just after the last occurrence of the current element.
    - We then update the value of the corresponding key in the hash map, setting it to the index of the current element.
3. After traversing the entire sequence of elements, `longest` holds the length of the longest substring without repeating characters.

The following illustration shows these steps in detail:



Let's look at the code for the solution:

Java

```java
class Solution {
    public static int findLongestSubstring(String str) {

        // Check the length of input str
        if (str.length() == 0) {
            return 0;
        }

        int n = str.length();
        int windowStart = 0, longest = 0, windowLength = 0, i = 0;

        Hashtable <Character, Integer> lastSeenAt = new Hashtable <Character, Integer> ();

        // Traverse input str to find the longest substring
        // without repeating characters.
        for (i = 0; i < n; i++) {
            // If the current element is not present in the hash map,
            // then store it in the hash map with the current index as the value.
            if (!lastSeenAt.containsKey(str.charAt(i))) {
                lastSeenAt.put(str.charAt(i), i);
            } else {

                // If the current element is present in the hash map,
                // it means that this element may have appeared before.
                // Check if the current element occurs before or after `windowStart`.
                if (lastSeenAt.get(str.charAt(i)) >= windowStart) {
                    windowLength = i - windowStart;
                    if (longest < windowLength) {
```

Longest Substring without Repeating Characters

Solution summary

To recap, the solution to this problem can be divided into the following parts:

2. Use a hash map to store elements along with their respective indexes.
    - If the current element is present in the hash map, check whether it's already present in the current window. If it is, we have found the end of the current window and the start of the next. We check if it's longer than the longest window seen so far and update it accordingly.
    - Store the current element in the hash map with the key as the element and the value as the current index.
3. At the end of the traversal, we have the length of the longest substring with all distinct characters.

Time complexity

We have to iterate over all the n elements in the string. Therefore, the time complexity is $O(n)$.

Space complexity

We need extra space to store the last occurrence of each element. In the worst-case scenario, all of the elements can be unique and we need to store all n elements. Therefore, the space complexity will be $O(n)$.