# 3. Longest Substring Without Repeating Characters

Medium Hash Table String Sliding Window

## Problem Description

The task is to find the length of the longest substring within a given string `s` that does not contain any repeating characters. A substring is defined as a contiguous sequence of characters within a string. The goal is to seek out the longest possible substring where each character is unique; no character appears more than once.

Imagine you have a unique piece of string and you want to make the longest possible necklace with beads where each bead must have a different shape or color. How would you pick beads so that repeats are avoided? Similar to this, the problem requires us to find such a unique sequence of characters in the given string `s`.

## Intuition

To solve this challenge, the approach revolves around using a sliding window to scan through the string `s` while keeping track of unique characters we've seen so far. This technique involves expanding and shrinking a window on different portions of the string to cover the non-repeating sequence of characters.

We use two pointers `i` (the start of the window) and `j` (the end of the window) to represent the current segment of the string we're looking at. If we see a new character that hasn't appeared in the current window, it's safe to add this character to our current sequence and move the end pointer `j` ahead.

However, if we find a character that's already in our current window, it means we've found a repeat and must therefore move the start pointer `i` forward. We do this by removing characters from the start of our window until the repeating character is eliminated from it.

During this process, we always keep track of the maximum length of the substring found that meets the condition. The length of the current unique character substring can be calculated by taking the difference of the end and start pointers `j - i` and adding 1 (since our count is zero-indexed).

At the end of this process, when we've checked all characters in the string, the maximum length we tracked gives us the length of the longest substring without repeating characters.

## Solution Approach

The solution implements a **sliding window** algorithm, which is an efficient way to maintain a subset of data in a larger dataset such as an array or string. In this context, we're dealing with a string and aiming to find a length of substring, i.e. a continuous range of characters, with distinct values. Two pointers, often denoted as `i` and `j`, are used to represent the current window, starting from the beginning of the string and ending at the last unique character found.

The algorithm also incorporates a hash set, named `ss` in the code, to efficiently verify if a character has already been seen within the current window. Hash sets offer constant time complexity for add, remove, and check operations, which makes them an ideal choice for this algorithm.

Here is the step-by-step breakdown of the implementation:

1. Initialize a hash set `ss` to record the characters in the current window. This will allow us to quickly check if a character is part of the current substring without having to scan all of it.
2. Initialize two pointers, `i` and `j`. `i` will point to the start of the current window, and `j` will iterate over the string to examine each character.
3. Initialize a variable `ans` to keep track of the length of the longest substring found.
4. Iterate over the string with `j`. For each character `c` located at the `j`th position:
   - While `c` is already in the set `ss` (implying a repeat and therefore a violation of the unique-substring rule), remove characters starting from the `i`th position and move `i` forward; this effectively shrinks the window from the left side until `c` can be added without creating a duplicate.
   - Once `c` is not in `ss`, add `c` to `ss` to include it in the current window.
   - Update `ans` if the current window size (`j - i + 1`) is larger than the maximum found so far.
5. After iterating over all characters, return the value of `ans`.

The solution functions within $O(n)$ time complexity—where $n$ is the length of the string—since each character in the string is visited once by `j` and at most once by `i` as the window is moved forward.

To demonstrate the behavior of the sliding window algorithm, consider the Java template provided in the reference solution approach. This generic algorithm pattern consists of two pointers moving over a dataset and a condition checked in a while loop that modifies one of the pointers (`j`) based on some condition (`check(j, i)`) applied to the current range (or window) that the pointers define.

```
for (int i = 0, j = 0; j < n; ++j) {
  while (i < j && check(j, i)) {
    ++i;
  }
  // logic of specific problem
}
```

In our solution, the "check" is finding if `c` is already in the set `ss`, and the logic after the while loop is the add-to-set and max-value-update operations.

### Example Walkthrough

Let's walk through the solution approach by using a simple example. Consider the input string `s = "abcabcbb"`.

1. **Initialize variables**:
   - A hash set `ss` to store characters of the current substring without repeating ones.
   - Two pointers `i` (start of the window) set to `0`, and `j` (end of the window) also set to `0`.
   - A variable `ans` to store the length of the longest substring found, set to `0`.
2. **Traverse the string with `j`**:
   - The outer loop moves `j` from the left to the right across the string.
3. **Examining each character**:
   - When `j = 0`, the character is `'a'`. It's not in `ss`, so add it to the set and `ans` is updated to `1`.
   - Move `j` to `1`, now the character is `'b'`. It's not in `ss`, so add it, and `ans` is updated to `2`.
   - Move `j` to `2`, the character is `'c'`. It's not in `ss`, so add it, and `ans` is updated to `3`.
   - Move `j` to `3`, we find `'a'` again. It's in `ss`, so we enter the while loop to start removing from the left side:
     - Remove `'a'` from `ss`, and move `i` to `1`.
   - Add `'a'` back as its repeat was removed, and move `j` to `4`.
   - Now `j = 4` and the character is `'b'`. Since `'b'` is in `ss`, remove characters with the while loop:
     - Remove `'b'` from `ss`, and move `i` to `2`.
   - Because the repeat `'b'` has been removed from `ss`, add the new `'b'` and `j` moves to `5`.
   - Continuing this pattern, `j` keeps moving to the right, adding unique characters to `ss`, and updating `ans` if we find a longer unique-substring along the way.
4. **Completing the traverse**:
   - As `j` progresses to the end of the string (`j == 8`), we keep removing duplicates from the left and adding new characters to the right until our window contains unique characters only.
   - The length of each window is calculated and compared with `ans`, and `ans` is updated if a longer length is found.
5. **Return the result**:
   - After the above process, we find that the longest substring without repeating characters is `'abc'` with a length of `3`, which is the final value of `ans`.

Applying this approach to our example string `s = "abcabcbb"`, we successfully find that the longest substring without repeating characters is `3` characters long.

## Solution Implementation

### Python

```python
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # Initialize a set to store unique characters of the substring
        unique_chars = set()

        # Initialize pointers for the sliding window
        left_pointer = 0
        max_length = 0

        # Iterate over the string using the right_pointer
        for right_pointer, char in enumerate(s):

            # If the current char is already in the set,
            # remove characters from the left until the current char
            # is no longer in the set to assure all unique substring
            while char in unique_chars:
                unique_chars.remove(s[left_pointer])
                left_pointer += 1  # Shrink the window from the left side

            # Add the current char to the set as it's unique in current window
            unique_chars.add(char)

            # Update the max length if the current window size is greater
            max_length = max(max_length, right_pointer - left_pointer + 1)

        # Return the length of the longest substring without repeating characters
        return max_length
```

### Java

```java
class Solution {
    // Method to calculate the length of the longest substring without repeating characters
    public int lengthOfLongestSubstring(String s) {
        // Use a HashSet to store characters in the current window without duplicates
        Set<Character> charSet = new HashSet<>();
        int leftPointer = 0; // Initialize the left pointer for the sliding window
        int maxLength = 0; // Variable to keep track of the longest substring length

        // Iterate through the string with the right pointer
        for (int rightPointer = 0; rightPointer < s.length(); ++rightPointer) {
            char currentChar = s.charAt(rightPointer); // Current character at the right pointer

            // If currentChar is already in the set, it means we have found a repeating character
            // We slide the left pointer of the window to the right until the duplicate is removed
            while (charSet.contains(currentChar)) {
                charSet.remove(s.charAt(leftPointer++));
            }

            // Add the current character to the set as it is now unique in the current window
            charSet.add(currentChar);

            // Calculate the length of the current window (rightPointer - leftPointer + 1)
            // Update the maxLength if the current window is larger
            maxLength = Math.max(maxLength, rightPointer - leftPointer + 1);
        }

        // Return the length of the longest substring without repeating characters
        return maxLength;
    }
}
```

### C++

```cpp
#include <string>
#include <unordered_set>
#include <algorithm>

class Solution {
public:
    int lengthOfLongestSubstring(std::string s) {
        // This unordered set is used to store the characters that are currently in the
        // longest substring without repeating characters.
        std::unordered_set<char> charSet;

        // The starting index of the substring.
        int start = 0;
        // The length of the longest substring without repeating characters.
        int maxLength = 0;

        // Iterate over the string.
        for (int end = 0; end < s.size(); ++end) {
            // If the character at the current ending index of the substring already exists
            // in the character set, continue to remove characters from the start of the
            // substring until the character is no longer in the set.
            while (charSet.count(s[end])) {
                charSet.erase(s[start]);
                start += 1;
            }

            // Insert the current character into the set.
            charSet.insert(s[end]);

            // Calculate the length of the current substring and update maxLength
            // if this length is the largest we've found so far.
            maxLength = std::max(maxLength, end - start + 1);
        }

        // Return the length of the longest substring found.
        return maxLength;
    }
};
```

### TypeScript

```typescript
function lengthOfLongestSubstring(s: string): number {
    // Initialize the length of the longest substring without repeating characters
    let maxLength = 0;

    // Create a Set to store the unique characters of the current substring
    const seenCharacters = new Set<string>();

    // Use two pointers i and j to denote the start and end of the substring
    let i = 0;
    let j = 0;

    while (j < s.length) {
        // If the current character is already in the set, remove characters from the set starting from the beginning
        // until the current character is no longer in the set
        while (seenCharacters.has(s[j])) {
            seenCharacters.delete(s[i]);
            i++;
        }

        // Add the current character to the set
        seenCharacters.add(s[j]);

        // Calculate the length of the current substring and update the maxLength if needed
        maxLength = Math.max(maxLength, j - i + 1);

        // Move to the next character
        j++;
    }

    // Return the length of the longest substring without repeating characters
    return maxLength;
}
```

```python
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        # Initialize a set to store unique characters of the substring
        unique_chars = set()

        # Initialize pointers for the sliding window
        left_pointer = 0
        max_length = 0

        # Iterate over the string using the right_pointer
        for right_pointer, char in enumerate(s):

            # If the current char is already in the set,
            # remove characters from the left until the current char
            # is no longer in the set to assure all unique substring
            while char in unique_chars:
                unique_chars.remove(s[left_pointer])
                left_pointer += 1  # Shrink the window from the left side

            # Add the current char to the set as it's unique in current window
            unique_chars.add(char)

            # Update the max length if the current window size is greater
            max_length = max(max_length, right_pointer - left_pointer + 1)

        # Return the length of the longest substring without repeating characters
        return max_length
```

## Time and Space Complexity

### Time Complexity

The time complexity of the code is $O(2n)$ which simplifies to $O(n)$, where $n$ is the length of the string `s`. This is because in the worst case, each character will be visited twice by the two pointers `i` and `j` - once when `j` encounters the character and once when `i` moves past the character after it has been found in the set `ss`. However, each character is processed only a constant number of times, hence we consider the overall time complexity to be linear.

### Space Complexity

The space complexity of the code is $O(min(n, m))$, where $n$ is the size of the string `s` and $m$ is the size of the character set (the number of unique characters in `s`). In the worst case, the set `ss` can store all unique characters of the string if all characters in the string are distinct. However, $m$ is the limiting factor since it represents the size of the character set that can be stored in `ss`. Therefore, if $n$ is larger than $m$, the space complexity is limited by $m$ rather than $n$.