Question 6: Longest Substring Without Repeating Characters

Problem

Find the length of the longest substring without repeating characters.

Algorithm

1. Use sliding window technique with two pointers: left and right.
2. Use a set to keep track of characters in current window.
3. Expand window to right (add character) when possible.
4. Contract window from left (remove character) when a duplicate is found.
5. Track maximum length of window during this process.

Program Implementation

cpp

```cpp
#include <iostream>
#include <string>
#include <unordered_set>
#include <algorithm>
using namespace std;

int lengthOfLongestSubstring(string s) {
    int n = s.length();
    int maxLength = 0;
    unordered_set<char> charSet;
    int left = 0;

    for (int right = 0; right < n; right++) {
        if (charSet.find(s[right]) == charSet.end()) {
            charSet.insert(s[right]);
            maxLength = max(maxLength, right - left + 1);
        } else {
            while (charSet.find(s[right]) != charSet.end()) {
                charSet.erase(s[left]);
                left++;
            }
            charSet.insert(s[right]);
        }
    }

    return maxLength;
}

int main() {
    string s1 = "abcabcbb";
    cout << "Longest substring without repeating characters in \"" << s1 << "\": "
         << lengthOfLongestSubstring(s1) << endl;

    string s2 = "bbbbb";
    cout << "Longest substring without repeating characters in \"" << s2 << "\": "
         << lengthOfLongestSubstring(s2) << endl;

    string s3 = "pwwkew";
    cout << "Longest substring without repeating characters in \"" << s3 << "\": "
         << lengthOfLongestSubstring(s3) << endl;

    return 0;
}
```

Optimized Implementation

cpp

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
#include <algorithm>
using namespace std;

int lengthOfLongestSubstring(string s) {
    int n = s.length();
    int maxLength = 0;
    unordered_map<char, int> charMap;  // Char -> last position
    int left = 0;

    for (int right = 0; right < n; right++) {
        if (charMap.find(s[right]) != charMap.end() && charMap[s[right]] >= left) {
            left = charMap[s[right]] + 1;
        }
        charMap[s[right]] = right;
        maxLength = max(maxLength, right - left + 1);
    }

    return maxLength;
}
```

Time Complexity

- O(n) where n is the length of the string
- Each character is processed at most twice (once added to the set, once removed)

Space Complexity

- O(min(m, n)) where n is the length of the string and m is the size of the character set
- In practice, m is limited by the size of the alphabet (e.g., 26 for lowercase English letters)

Example Explanation

Let's trace through the example "abcabcbb":

1. Start with empty window: left = 0, right = 0, charSet = {}
2. Process 'a': charSet = {'a'}, maxLength = 1
3. Process 'b': charSet = {'a', 'b'}, maxLength = 2
4. Process 'c': charSet = {'a', 'b', 'c'}, maxLength = 3

5. Process 'a' (duplicate): remove 'a' from set, increment left to 1
    - charSet = {'b', 'c', 'a'}, maxLength = 3
6. Process 'b' (duplicate): remove 'b' from set, increment left to 2
    - charSet = {'c', 'a', 'b'}, maxLength = 3
7. Process 'c' (duplicate): remove 'c' from set, increment left to 3
    - charSet = {'a', 'b', 'c'}, maxLength = 3
8. Process 'b' (duplicate): remove 'a' from set, increment left to 4
    - charSet = {'b', 'c', 'b'}, maxLength = 3
9. Process 'b' (duplicate): remove 'b', 'c' from set, increment left to 6
    - charSet = {'b'}, maxLength = 3

The maximum length is 3, corresponding to substrings "abc" or "cab".