Question 8: Longest Palindromic Substring

Problem

Find the longest palindromic substring in a given string.

Algorithm 1: Expand Around Center

1. Iterate through each character in the string.
2. For each character, expand outward checking for palindromes: a. Check for odd-length palindromes centered at the current character. b. Check for even-length palindromes centered between the current and next character.
3. Track the longest palindrome found.

Program Implementation

cpp

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

string longestPalindrome(string s) {
    if (s.length() <= 1) return s;

    int start = 0;
    int maxLength = 1;

    auto expandAroundCenter = [&](int left, int right) {
        while (left >= 0 && right < s.length() && s[left] == s[right]) {
            left--;
            right++;
        }

        int length = right - left - 1;
        if (length > maxLength) {
            maxLength = length;
            start = left + 1;
        }
    };

    for (int i = 0; i < s.length(); i++) {
        // Odd length palindromes
        expandAroundCenter(i, i);

        // Even length palindromes
        expandAroundCenter(i, i + 1);
    }

    return s.substr(start, maxLength);
}

int main() {
    string s1 = "babad";
    cout << "Longest palindromic substring in \"" << s1 << "\": "
         << longestPalindrome(s1) << endl;

    string s2 = "cbbd";
    cout << "Longest palindromic substring in \"" << s2 << "\": "
         << longestPalindrome(s2) << endl;

    string s3 = "racecar";
    cout << "Longest palindromic substring in \"" << s3 << "\": "
         << longestPalindrome(s3) << endl;
```

```
    return 0;
}
```

Algorithm 2: Dynamic Programming

1. Create a boolean DP table where dp[i][j] indicates whether substring from i to j is a palindrome.
2. Initialize all single characters as palindromes.
3. Fill the table for substrings of length 2, then 3, and so on.
4. Track the longest palindrome found.

cpp

```cpp
string longestPalindromeDP(string s) {
    int n = s.length();
    if (n <= 1) return s;

    bool dp[n][n] = {false};
    int start = 0;
    int maxLength = 1;

    // All substrings of length 1 are palindromes
    for (int i = 0; i < n; i++) {
        dp[i][i] = true;
    }

    // Check for substrings of length 2
    for (int i = 0; i < n - 1; i++) {
        if (s[i] == s[i + 1]) {
            dp[i][i + 1] = true;
            start = i;
            maxLength = 2;
        }
    }

    // Check for lengths greater than 2
    for (int len = 3; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;  // ending index

            if (s[i] == s[j] && dp[i + 1][j - 1]) {
                dp[i][j] = true;

                if (len > maxLength) {
                    maxLength = len;
                    start = i;
                }
            }
        }
    }

    return s.substr(start, maxLength);
}
```

Time Complexity

- Expand Around Center: $O(n^2)$ where n is the length of the string

- Dynamic Programming: $O(n^2)$ where n is the length of the string

Space Complexity

- Expand Around Center: O(1)
- Dynamic Programming: $O(n^2)$ for the DP table

Example Explanation

Let's trace through the example "babad" using the expand around center approach:

1. Start with i = 0 ('b'):
   - Expand around 'b': "b" (palindrome of length 1)
   - Expand between 'b' and 'a': not a palindrome
2. i = 1 ('a'):
   - Expand around 'a': "a" (palindrome of length 1)
   - Expand between 'a' and 'b': not a palindrome
3. i = 2 ('b'):
   - Expand around 'b': "b" (palindrome of length 1)
   - Expand between 'b' and 'a': "ab" is not a palindrome
4. i = 3 ('a'):
   - Expand around 'a': "bab" (palindrome of length 3)
   - Expand between 'a' and 'd': not a palindrome
5. i = 4 ('d'):
   - Expand around 'd': "d" (palindrome of length 1)

The algorithm returns "bab" as the longest palindromic substring with length 3.