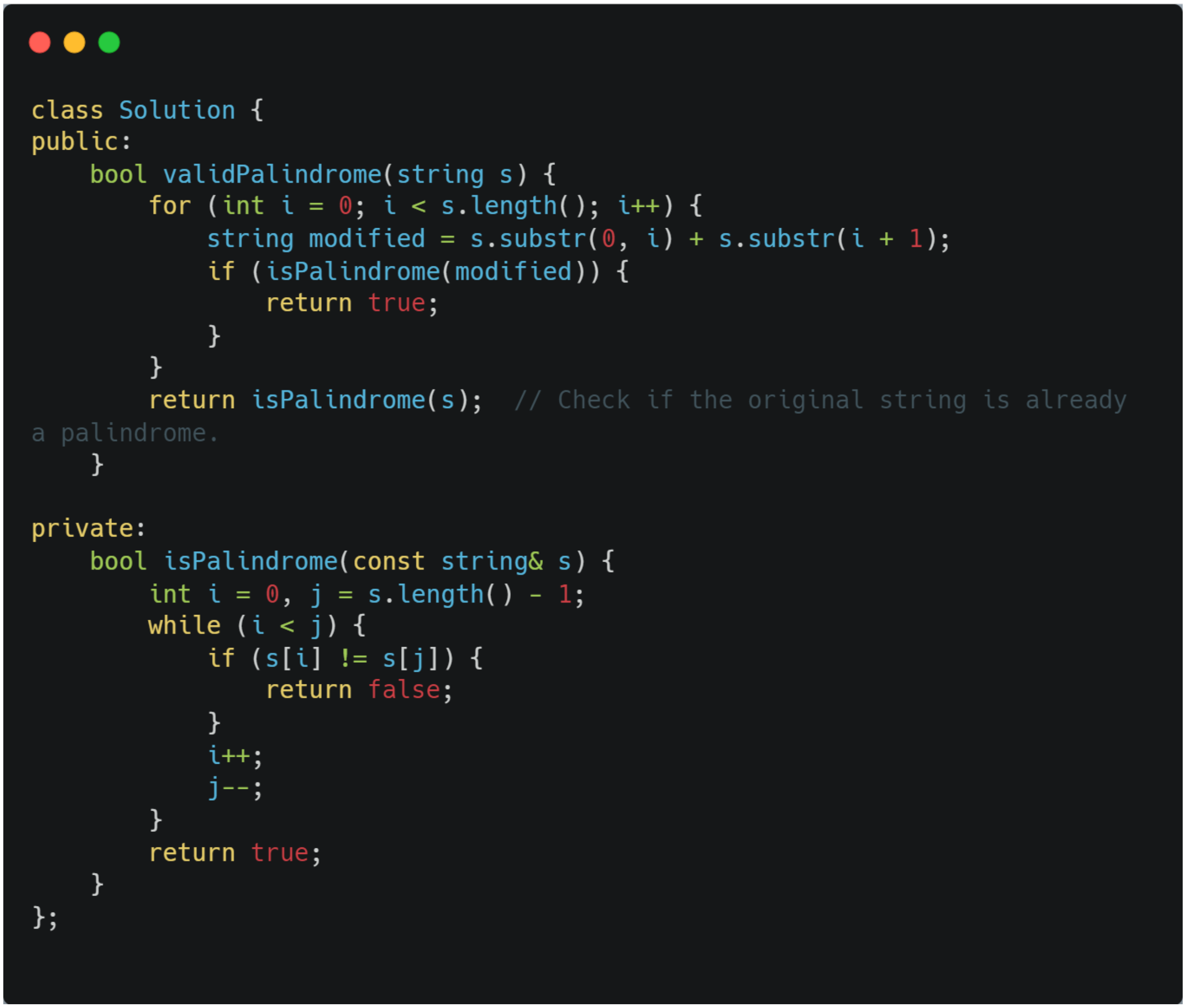
1. Brute Force Approach:

- **Approach:**
  - Iterate over each character in the string.
  - For each character, remove it and check if the resulting string is a palindrome.
  - If any of the resulting strings is a palindrome, return true; otherwise, return false.



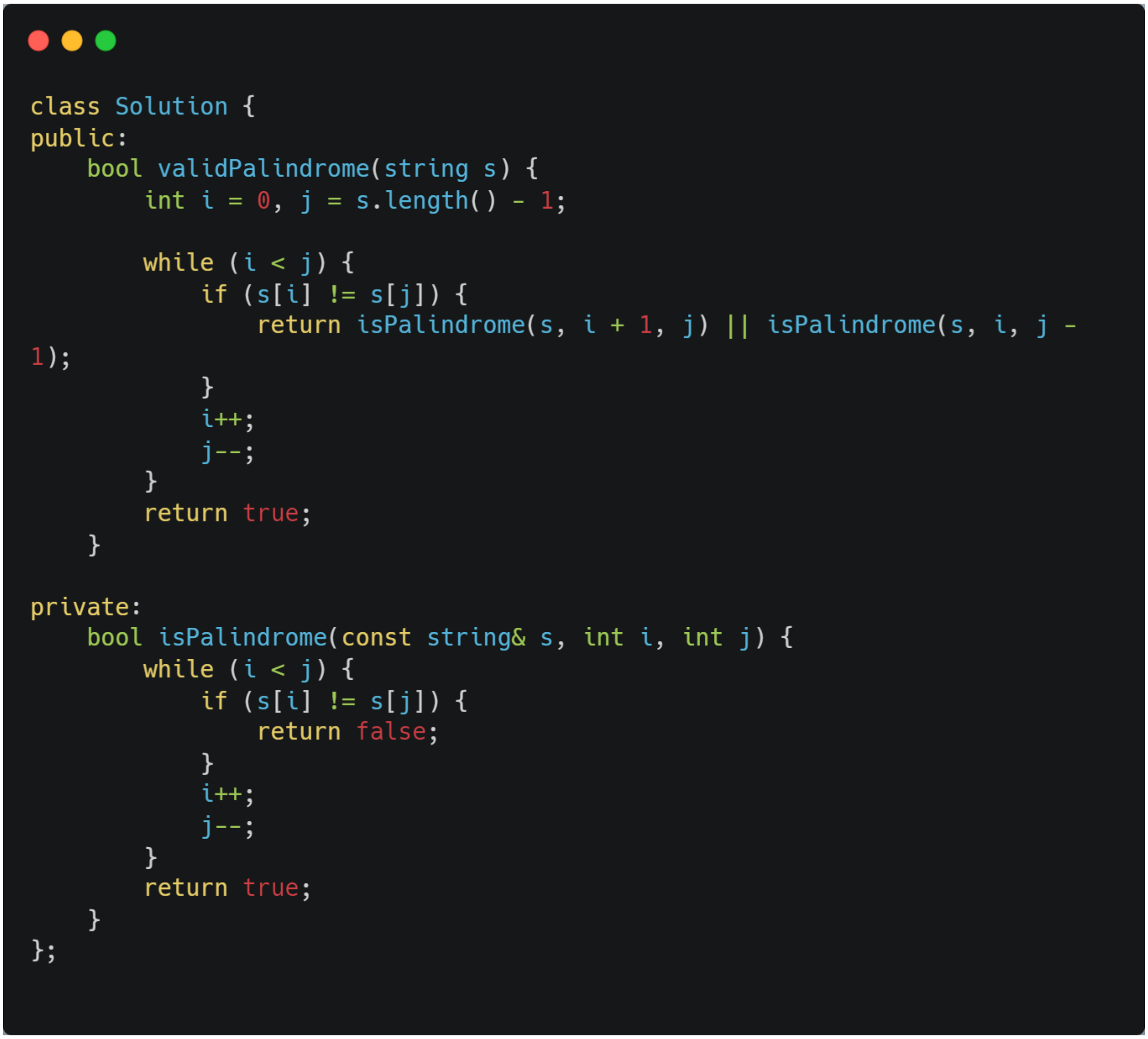
```
class Solution {
public:
    bool validPalindrome(string s) {
        for (int i = 0; i < s.length(); i++) {
            string modified = s.substr(0, i) + s.substr(i + 1);
            if (isPalindrome(modified)) {
                return true;
            }
        }
        return isPalindrome(s);  // Check if the original string is already
a palindrome.
    }

private:
    bool isPalindrome(const string& s) {
        int i = 0, j = s.length() - 1;
        while (i < j) {
            if (s[i] != s[j]) {
                return false;
            }
            i++;
            j--;
        }
        return true;
    }
};
```

- **Complexity:**
  - **Time Complexity:** O(n^2) due to checking for palindrome after removing each character.
  - **Space Complexity:** O(n) for creating substrings.

2. Better Approach:

- **Approach:**
  - Use two pointers starting from the beginning and end of the string.

- If a mismatch is found, check if either the substring obtained by removing the left character or the one by removing the right character is a palindrome.

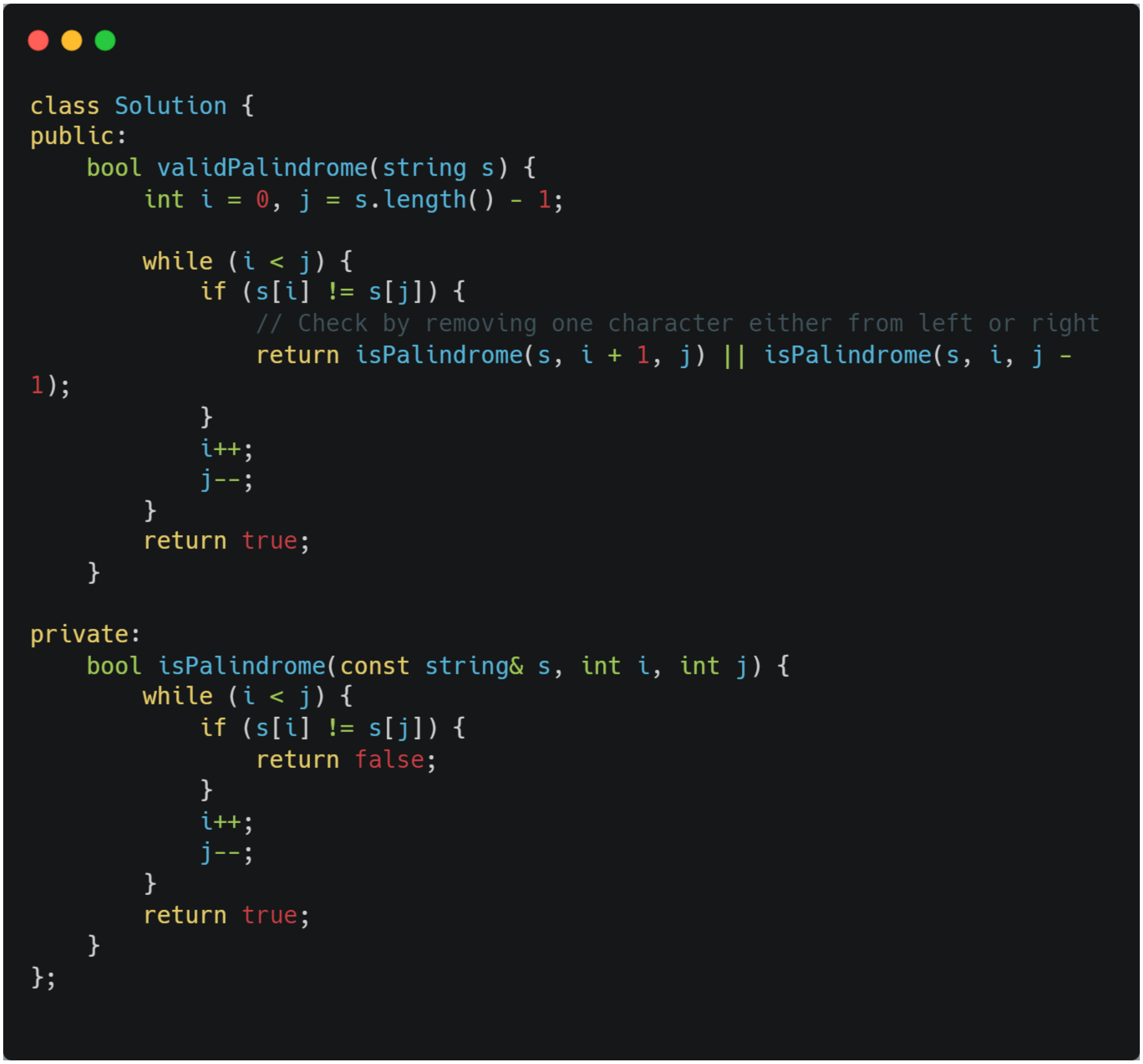


```
class Solution {
public:
    bool validPalindrome(string s) {
        int i = 0, j = s.length() - 1;

        while (i < j) {
            if (s[i] != s[j]) {
                return isPalindrome(s, i + 1, j) || isPalindrome(s, i, j -
1);
            }
            i++;
            j--;
        }
        return true;
    }

private:
    bool isPalindrome(const string& s, int i, int j) {
        while (i < j) {
            if (s[i] != s[j]) {
                return false;
            }
            i++;
            j--;
        }
        return true;
    }
};
```

- **Complexity:**
  - **Time Complexity:** O(n), since we traverse the string at most twice.
  - **Space Complexity:** O(1), as no extra space is required.

3. Optimal Approach:

- **Approach:**
  - The optimal approach is essentially the same as the better approach.
  - The better approach already achieves the most efficient solution, ensuring minimal time complexity by checking both possibilities when a mismatch occurs.



```
class Solution {
public:
    bool validPalindrome(string s) {
        int i = 0, j = s.length() - 1;

        while (i < j) {
            if (s[i] != s[j]) {
                // Check by removing one character either from left or right
                return isPalindrome(s, i + 1, j) || isPalindrome(s, i, j -
1);
            }
            i++;
            j--;
        }
        return true;
    }

private:
    bool isPalindrome(const string& s, int i, int j) {
        while (i < j) {
            if (s[i] != s[j]) {
                return false;
            }
            i++;
            j--;
        }
        return true;
    }
};
```

Complexity:

- **Time Complexity:** O(n).
- **Space Complexity:** O(1).