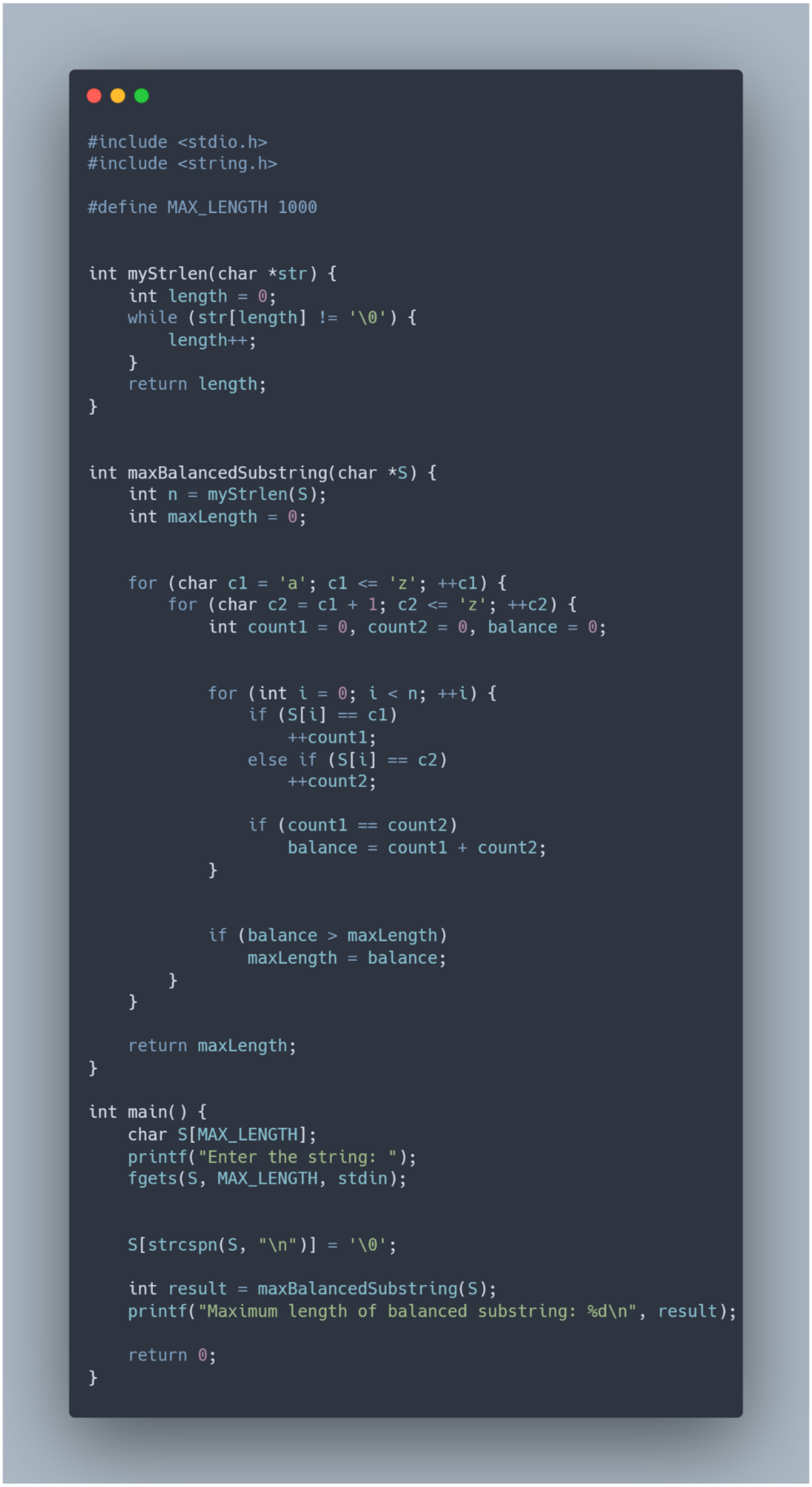
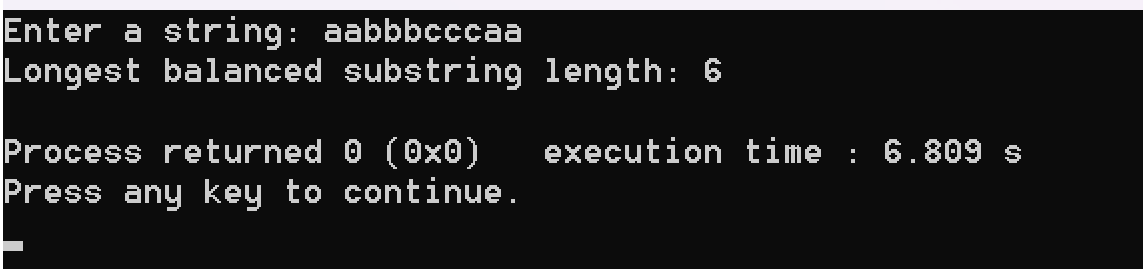
Implementation of non_recursive:

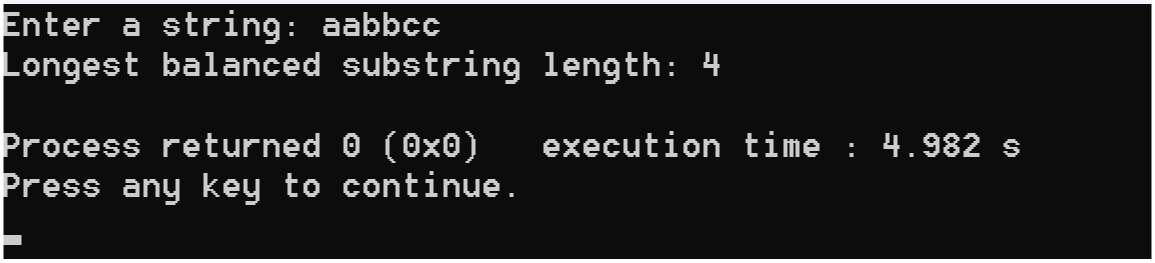


```c
#include <stdio.h>
#include <string.h>

#define MAX_LENGTH 1000


int myStrlen(char *str) {
    int length = 0;
    while (str[length] != '\0') {
        length++;
    }
    return length;
}


int maxBalancedSubstring(char *S) {
    int n = myStrlen(S);
    int maxLength = 0;


    for (char c1 = 'a'; c1 <= 'z'; ++c1) {
        for (char c2 = c1 + 1; c2 <= 'z'; ++c2) {
            int count1 = 0, count2 = 0, balance = 0;


            for (int i = 0; i < n; ++i) {
                if (S[i] == c1)
                    ++count1;
                else if (S[i] == c2)
                    ++count2;

                if (count1 == count2)
                    balance = count1 + count2;
            }


            if (balance > maxLength)
                maxLength = balance;
        }
    }

    return maxLength;
}

int main() {
    char S[MAX_LENGTH];
    printf("Enter the string: ");
    fgets(S, MAX_LENGTH, stdin);


    S[strcspn(S, "\n")] = '\0';

    int result = maxBalancedSubstring(S);
    printf("Maximum length of balanced substring: %d\n", result);

    return 0;
}
```

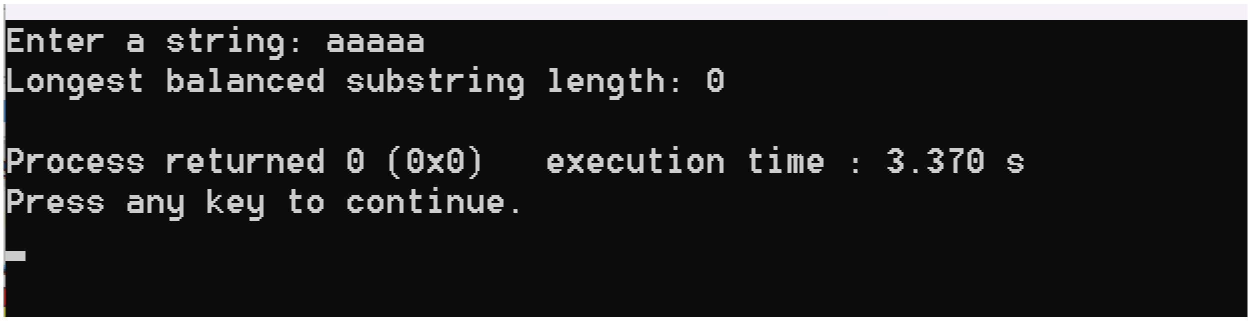
Output:



```
Enter a string: aabbbcccaa
Longest balanced substring length: 6

Process returned 0 (0x0)   execution time : 6.809 s
Press any key to continue.
```

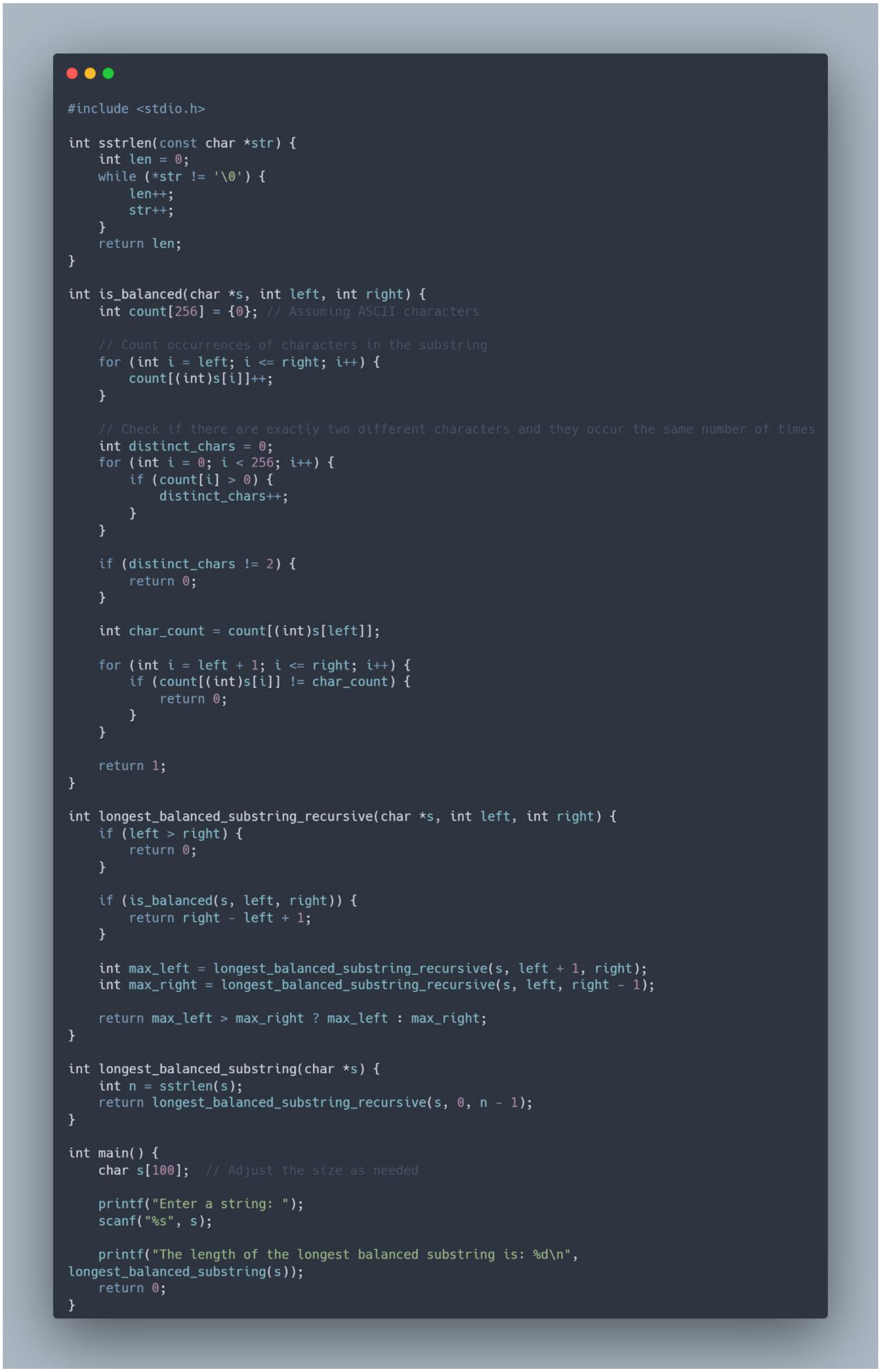


```
Enter a string: aabbcc
Longest balanced substring length: 4

Process returned 0 (0x0)   execution time : 4.982 s
Press any key to continue.
```



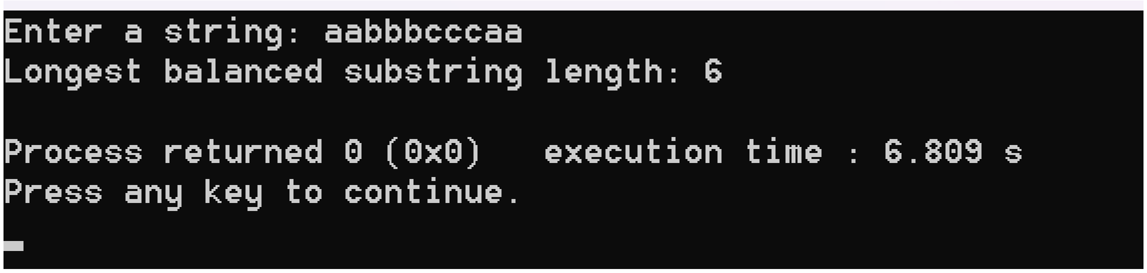
```
Enter a string: aaaaa
Longest balanced substring length: 0

Process returned 0 (0x0)   execution time : 3.370 s
Press any key to continue.
```

Implementation of Recursive Algorithm

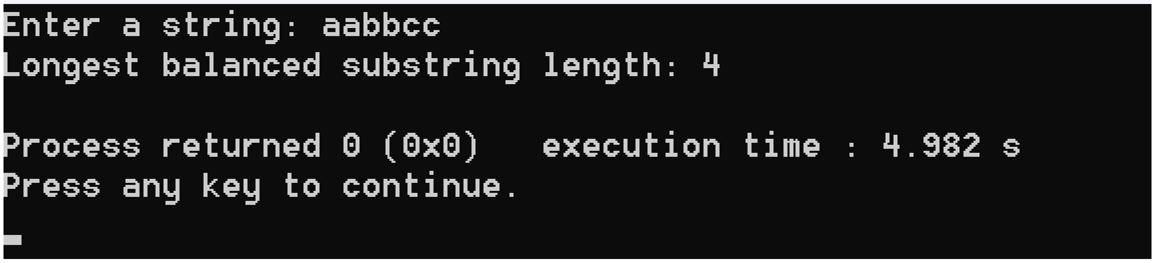


```c
#include <stdio.h>

int sstrlen(const char *str) {
    int len = 0;
    while (*str != '\0') {
        len++;
        str++;
    }
    return len;
}

int is_balanced(char *s, int left, int right) {
    int count[256] = {0}; // Assuming ASCII characters

    // Count occurrences of characters in the substring
    for (int i = left; i <= right; i++) {
        count[(int)s[i]]++;
    }

    // Check if there are exactly two different characters and they occur the same number of times
    int distinct_chars = 0;
    for (int i = 0; i < 256; i++) {
        if (count[i] > 0) {
            distinct_chars++;
        }
    }

    if (distinct_chars != 2) {
        return 0;
    }

    int char_count = count[(int)s[left]];

    for (int i = left + 1; i <= right; i++) {
        if (count[(int)s[i]] != char_count) {
            return 0;
        }
    }

    return 1;
}

int longest_balanced_substring_recursive(char *s, int left, int right) {
    if (left > right) {
        return 0;
    }

    if (is_balanced(s, left, right)) {
        return right - left + 1;
    }

    int max_left = longest_balanced_substring_recursive(s, left + 1, right);
    int max_right = longest_balanced_substring_recursive(s, left, right - 1);

    return max_left > max_right ? max_left : max_right;
}

int longest_balanced_substring(char *s) {
    int n = sstrlen(s);
    return longest_balanced_substring_recursive(s, 0, n - 1);
}

int main() {
    char s[100];  // Adjust the size as needed

    printf("Enter a string: ");
    scanf("%s", s);

    printf("The length of the longest balanced substring is: %d\n",
longest_balanced_substring(s));
    return 0;
}
```

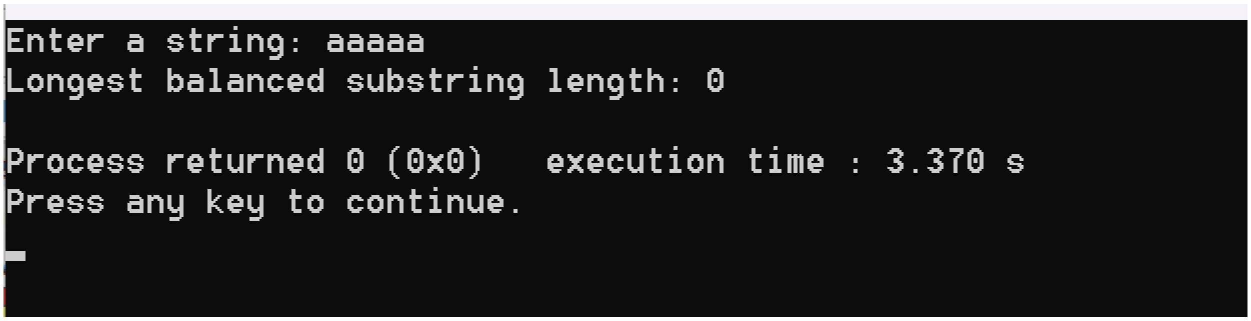
Output:



```
Enter a string: aabbbcccaa
Longest balanced substring length: 6

Process returned 0 (0x0)   execution time : 6.809 s
Press any key to continue.
```



```
Enter a string: aabbcc
Longest balanced substring length: 4

Process returned 0 (0x0)   execution time : 4.982 s
Press any key to continue.
```



```
Enter a string: aaaaa
Longest balanced substring length: 0

Process returned 0 (0x0)   execution time : 3.370 s
Press any key to continue.
```

Comparison Between analysis:

Non__recursive	Recursive
iterates through all possible pairs of distinct characters in the string, then checks each pair's balance. It counts occurrences of each character and updates the maximum balanced substring .length accordingly	uses a recursive approach where it checks all possible substrings to find the longest balanced one. It starts by checking the entire string and then recursively explores smaller substrings. It counts occurrences of characters within substrings to determine balance.
O(n) more efficient than the recursive approach, it's still not the most optimal solution	program has a time complexity of O(2^n) in the worst case due to its recursive nature, where n is the length

	of the string. This can be inefficient for large strings.