pseudocode for the non-recursive algorithm:

```
 // Function to calculate the length of a string
function length(s)
{
    cnt ← 0;
    i ← 0;
    while s[i] is not NULL do
    {
        cnt ← cnt + 1;
        i ← i + 1;
    }
 return cnt;
}

// Function to find the length of the longest balanced substring
function longestBalancedSubstring(s)
{
    n ← length(s); // Get the length of the string
    maxLength ← 0;
    if n < 2 then
        return 0;
    for i ← 0 to n - 1 do
```

```
    {
        count1 ← 0;
        count2 ← 0;
        j ← i;
        while s[j] is equal to s[i] or s[j] is equal to s[i + 1] do
        {
            if s[j] = s[i] then
                count1 ← count1 + 1;
            else
                count2 ← count2 + 1;
            if count1 = count2 and (count1 + count2) > maxLength then
                maxLength ← count1 + count2;
            j ← j + 1;
        }
    }
    return maxLength;
}
```

Analysis :

1. for loop in length() start from 0 to n-1 and the basic operation is increase cnt so using $\sum_{i=0}^{n-1} 1$. Give us O(n) or (n+3 for the hole function)

2. The if statement checks whether n is larger than 1 or not. This step takes O(1) and quit with 0 in return if it's 1 or 0.

3. the outer loop in second function starts from I = 0 to n-2, and the inner for start from j = I to n-1 and the basic operation is checking if the counter 1 = counter 2 so using $\sum_{i=0}^{n-2}\sum_{j=i}^{n-1} 1$ give us $O(n^2)$ or (n^2 - $\frac{n^2}{2}$ for the hole Algorithm)

4. because of they are in sequence, so we take the max that is $O(n^2)$

Therefore :

1. Best-case time complexity of the algorithm is O(n) because of when the input is less than 2 the quit before the nested loops.

2. Average-case time complexity is $O(n^2)$.

3. Worst-case time complexity is $O(n^2)$, it's occurring when the input is equal to 2 or more.

screenshots for the outputs :

```
int length(char* s) {
    return cnt;
}

int longestBalancedSubstring(char* s) {
    int n = length(s);
    if (n < 2) return 0;
    int maxLength = 0;
    for (int i = 0; i < n - 1; i++) {
    int count1 = 0, count2 = 0;
    for (int j = i; s[j] == s[i] || s[j] == s[i + 1]; j++) {
        s[j] == s[i] ? count1++ : count2++;
        count1 == count2 && (count1 + count2) > maxLength && (maxLength = count1 + count2);
    }
    }
    return maxLength;
}

int main() {
    char string[100];
    int exit = 1;
    while (exit != 0) {
        printf("Enter a string: ");
        scanf("%s", string);
        printf("%d\n", longestBalancedSubstring(string));
        printf("Do you want to continue? (0/1): ");
        scanf("%d", &exit);
    }
}/.h
```

```
ects\programing-projects\c\algo\" ; if ($?) { gcc 1.c
-o 1 } ; if ($?) { .\1 }
Enter a string: cabbacc
4
Do you want to continue? (0/1): 1
Enter a string: abababa
6
Do you want to continue? (0/1): 1
Enter a string: aaaaaaa
0
Do you want to continue? (0/1):
0
PS E:\Projects\programing-projects\c\algo>
```

pseudocode for the recursive algo:

```
// Function to find the maximum of two integers
Algorithm maxInt(a, b)
{
    return a > b ? a : b;
}
// Function to check if a substring is balanced
function isBalanced(subStr)
{
    count1 ← 0;
    count2 ← 0;
    for i ← 0 to end of subStr do
    {
        if subStr[i] equals subStr[0] then
            count1 ← count1 + 1;
        else if subStr[i] equals subStr[1] then
            count2 ← count2 + 1;
        else
            return 0; // Not balanced if any character different from first two.
    }
    return count1 equals count2;
}
```

```
// Function to find the length of the longest balanced substring from a given index

Algorithm lBSFromIndex(s, index)

{

    if not s[index] then

        return 0;

    maxLength ← 0;

    for i ← index + 1 to end of s do

    {

        strLen ← i - index + 1;

        char subStr[strLen];

        for j ← 0 to i - index do

        subStr[j] ← s[index + j];

        subStr[strLen] ← '\0';

        if isBalanced(subStr) then

              (maxLength ← maxInt(maxLength, strLen));

    }

    return maxInt(maxLength, lBSFromIndex(s, index + 1));

}
```

// Function to find the length of the longest balanced substring in the given string

Algorithm longestBalancedSubstring(s)

{

return lBSFromIndex(s, 0);

}

Analysis :

1. for the maxInt() it takes exactly two steps.

2. For the loop in isBalanced() it's start from 0 to n-1 with three more steps so $\sum_{i=0}^{n-1} 1$ so it will be O(n) or (n+3 for the hole function).

3. The longestBalancedSubstring() function will take just two steps also.

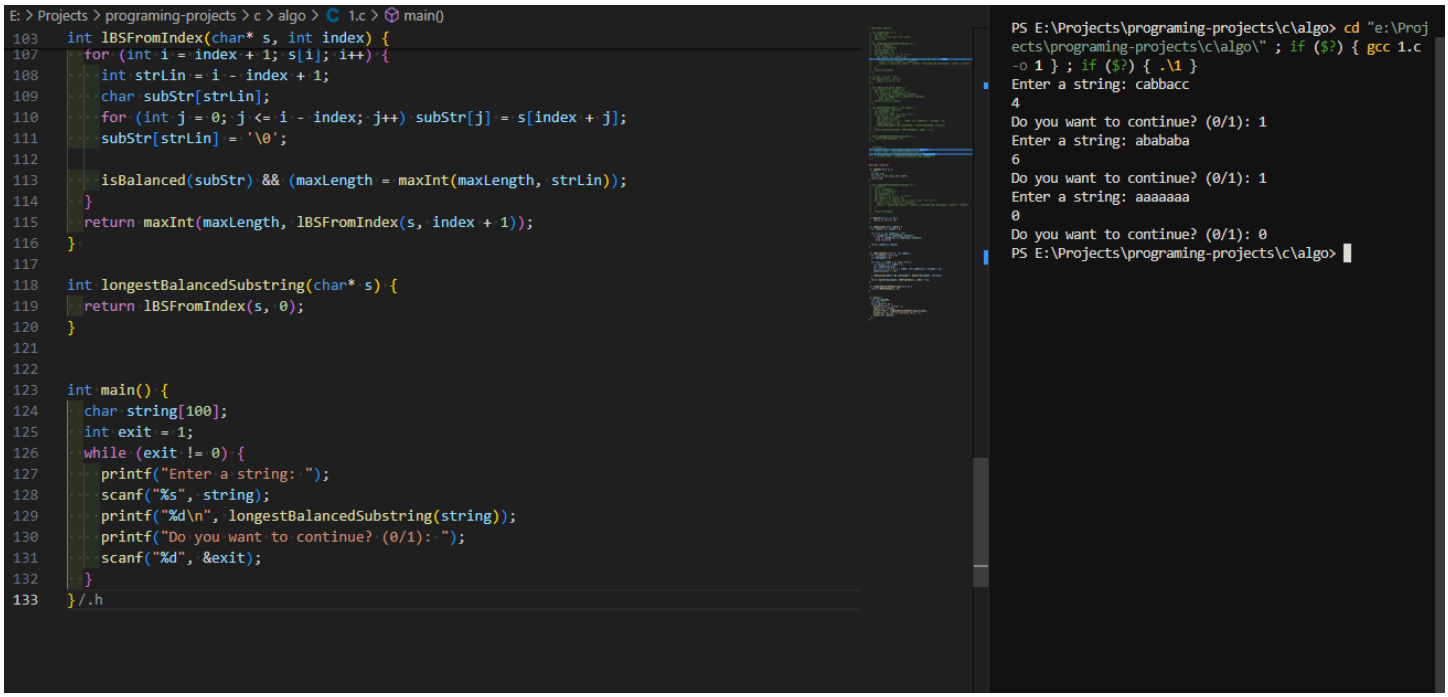
4. lBSFromIndex() has an outer loop that has a kind of three loops in it(the for loop, the maxInt() function invocation and the array initialization) so it will take $2n^2 + 5n$ so it's $O(n^2)$

5. all of the above is not the recursion part the form of the recursion will be T(n) = T(n -1) + n^2 and with T(1) = 1 (for simplicity but in real life it will be 4) whit the Iteration Method, it gave us the general form T(n) = T(n – k) + $\sum_{i=1}^{k}(\mathrm{n}-\mathrm{i}+1)^2$ with using T(1) = 1 the n – k =1 so the k will be equal to n-1 so the summation will be $\sum_{i=1}^{n-1}(\mathrm{n}-\mathrm{i}+1)^2$ and by substitute the $(\mathrm{n}-\mathrm{i}+1)^2$ with just I^2 for simplicity it will give us $\frac{n.(n-1).(2n-1)}{6} \cong \frac{n^3}{6}$, so it will be around $O(n^3)$

Therefore:

the time complexity is $O(n^3)$ in the best, average, and worst cases.

screenshots for output:



comparison

code	Best-case	Average-case	Worst-case
non-recursive	Θ(n)	Ʊ(n^2)	O(n^2)
recursive	Θ(n^3)	Ʊ(n^3)	O(n^3)

- in the three cases the non-recursive is better than recursive because of the recursive not quit until it reaches the end of the string'\0'.
- The first code depends on two for loops that loop around the string, which has a worst-case time complexity of O(n)if it less than 2 chars, otherwise the loop will reach the end any way and it will take an $O(n^2)$.
- The second code uses recursion and for loops, which make the time complexity in the three cases $O(n^3)$.
- In general, between these two codes it is recommended to use the non-recursive one anyway.