Longest Palindromic substring In C++

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

int LongestPalindromicSubstring(string str) {
    int n = str.length();
    bool dp[n][n];
    int len = 0;

    // Initialize dp array
    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 (str[i] == str[i + 1]) {
            dp[i][i + 1] = true;
            len = 2; // Update length of longest
palindromic substring
        } else {
            dp[i][i + 1] = false;
        }
    }

    // Check for substrings of length > 2
    for (int g = 2; g < n; g++) {
        for (int i = 0, j = g; j < n; i++, j++) {
            if (str[i] == str[j] && dp[i + 1][j - 1]) {
                dp[i][j] = true;
                len = g + 1; // Update length of longest
palindromic substring
            } else {
                dp[i][j] = false;
            }
        }
    }

    return len;
}

int main() {
    string str = "abccbc";
    int longestPalSubstrLen =
LongestPalindromicSubstring(str);
    cout << longestPalSubstrLen << endl;

    return 0;
}
```

Input:

str = "abccbc", n=6n = 6n=6

Initial Setup:

1. **dp table**:
   A boolean n×nn \times nn×n table is used to check if str[i...j] is a palindrome.
2. **Initialization**:
   - Single-character substrings (dp[i][i]) are palindromes, so initialize dp[i][i] = true for all i.
   - len = 0 (to store the length of the longest palindromic substring).

Step 1: Check for substrings of length 2.

- For each pair of adjacent characters (i,i+1)(i, i+1)(i,i+1):
  - If str[i] == str[i+1], set dp[i][i+1] = true and update len = 2.
  - Otherwise, set dp[i][i+1] = false.

DP Table After Length 2 Check:

i\j	**a**	**b**	**c**	**c**	**b**	**c**
a	T	F	-	-	-	-
b	-	T	F	-	-	-
c	-	-	T	T	-	-
c	-	-	-	T	F	-
b	-	-	-	-	T	F
c	-	-	-	-	-	T

len = 2 (because cc is a palindrome of length 2).

Step 2: Check for substrings of length > 2.

We now iterate for substrings of increasing length (g = 2, 3, ..., n-1).

Gap = 2 (g = 2):

For substrings of length 3, check if:

str[i] == str[j] and dp[i+1][j-1]\text{str[i] == str[j] \text{ and } dp[i+1][j-1]}
str[i] == str[j] and dp[i+1][j-1]

Substrings	Result	DP Update	Reason
"abc"	False	dp[0][2] = false	a != c
"bcc"	False	dp[1][3] = false	b != c
"ccb"	False	dp[2][4] = false	c != b
"cbc"	True	dp[3][5] = true	c == c and dp[4] [4] == true

DP Table After Gap 2:

i\j	a	b	c	c	b	c
a	T	F	F	-	-	-
b	-	T	F	F	-	-
c	-	-	T	T	F	-
c	-	-	-	T	F	T
b	-	-	-	-	T	F
c	-	-	-	-	-	T

len = 3 (because cbc is a palindrome of length 3).

Gap = 3 (g = 3):

For substrings of length 4, check if:

str[i] == str[j] and dp[i+1][j-1]\text{str[i] == str[j] \text{ and } dp[i+1][j-1]}
str[i] == str[j] and dp[i+1][j-1]

Substrings	Result	DP Update	Reason
"abcc"	False	dp[0][3] = false	a != c
"bccb"	True	dp[1][4] = true	b == b and dp[2][3] == true
"ccbc"	False	dp[2][5] = false	c != c

DP Table After Gap 3:

i\j	a	b	c	c	b	c
a	T	F	F	F	-	-
b	-	T	F	F	T	-
c	-	-	T	T	F	-
c	-	-	-	T	F	T
b	-	-	-	-	T	F
c	-	-	-	-	-	T

len = 4 (because bccb is a palindrome of length 4).

Gap = 4 (g = 4):

For substrings of length 5, check if:

str[i] == str[j] and dp[i+1][j-1]\text{str[i] == str[j] \text{ and } dp[i+1][j-1]}
str[i] == str[j] and dp[i+1][j-1]

Substrings	Result	DP Update	Reason
"abccb"	False	dp[0][4] = false	a != b
"bccbc"	False	dp[1][5] = false	b != c

DP Table After Gap 4:

No new updates, and **len = 4** remains unchanged.

Gap = 5 (g = 5):

For substrings of length 6, check:

str[i] == str[j] and dp[i+1][j-1]\text{str[i] == str[j] \text{ and } dp[i+1][j-1]}
str[i] == str[j] and dp[i+1][j-1]

Substrings	Result	DP Update	Reason
"abccbc"	False	dp[0][5] = false	a != c

Final Result:

- Longest palindromic substring length = **4** (bccb).

Output:-