Palindrome Partitioning

Given a string **str**, a partitioning of the string is a **palindrome** partitioning if every sub-string of the partition is a **palindrome,** the task is to find the minimum number of cuts needed for palindrome partitioning of the given string.

String :	ababbbabbababa
Palindrome partitioning:	a | babbbab | b | ababa

Palindrome Partition

Examples :
Input: *str = "geek"*
Output: *2*
Explanation: *We need to make minimum 2 cuts, i.e., "g ee k"*
Input: *str = "aaaa"*
Output*: 0*
Explanation: *The string is already a palindrome.*

Input: *str = "abcde"*
Output: *4*
Input: *str = "abbac"*
Output: *1*

Recursive Approach for Palindrome Partitioning:

This is the naive approach to solving the Palindrome Partitioning problem. In this approach, we will try to apply all possible partitions and at the end return the correct combination of partitions.

This approach is similar to that of Matrix Chain Multiplication problem.
In this approach, we recursively evaluate the following conditions:

- *If the current string is a palindrome, then we simply return true, as Palindrome Partitioning is possible.*
- *Else, like the Matrix Chain Multiplication problem,*
    - *we try making cuts at all possible places,*
    - *recursively calculate the cost for each cut*
    - *return the minimum value.*

Below is the implementation of the above approach.

```cpp
// C++ Code for Palindrome Partitioning
// Problem

#include <bits/stdc++.h>
using namespace std;

// Function to Check if a substring is a palindrome
bool isPalindrome(string String, int i, int j)
{
    while (i < j) {
        if (String[i] != String[j])
            return false;
        i++;
        j--;
    }
    return true;
}

// Function to find the minimum number of cuts needed for
// palindrome partitioning
int minPalPartion(string String, int i, int j)
{
    // Base case: If the substring is empty or a palindrome,
    // no cuts needed
    if (i >= j || isPalindrome(String, i, j))
        return 0;
```

```
    int ans = INT_MAX, count;

    // Iterate through all possible partitions and find the
    // minimum cuts needed
    for (int k = i; k < j; k++) {
        count = minPalPartion(String, i, k)
                + minPalPartion(String, k + 1, j) + 1;
        ans = min(ans, count);
    }

    return ans;
}

// Driver code
int main()
{
    string str = "ababbbabbababa";

    // Find the minimum cuts needed for palindrome
    // partitioning and display the result
    cout
        << "Min cuts needed for Palindrome Partitioning is "
        << minPalPartion(str, 0, str.length() - 1) << endl;

    return 0;
}
```


Output

```
Min cuts needed for Palindrome Partitioning is 3
```

Time Complexity: O(2n)
Auxiliary Space: O(n)

Optimising Overlapping Sub-problems in Recursive Approach for Palindrome Partitioning in (O(n3)):

Dynamic Programming Approach for Palindrome Partitioning in (O(n2)):

*The problem can be solved by finding the suffix starting from **j** and ending at index **i**, **(1 <= j <= i <= n – 1)**, which are **palindromes**. Hence, we can make a cut here that requires **1 +** min cut from rest substring **[0, j – 1]**. For all such palindromic suffixes starting at **j** and ending at **i**, keep minimising in **minCutDp[i]**. Similarly, we need to compute results for all such **i**. **(1 <= i <= n – 1)** and finally, **minCutDp[n – 1]** will be the minimum number of cuts needed for palindrome partitioning of the given string.*

Below is the implementation of the above approach:

```
#include <bits/stdc++.h>
using namespace std;

// Function to generate all possible palindromic substring
bool generatePalindrome(string& s,
                        vector<vector<bool> >& pal)
{
    int n = s.size();

    // Initialize the palindrome matrix for single
    // characters
    for (int i = 0; i < n; i++) {
        pal[i][i] = true;
    }
```

```
    // Iterate over different lengths of substrings
    for (int len = 2; len <= n; len++) {
        // Iterate over the starting positions of substrings
        // of current length
        for (int i = 0; i <= n - len; i++) {

            // Calculate the ending position of the
            // substring
            int j = i + len - 1;

            // Check if the characters at the starting and
            // ending positions are equal and if the
            // substring between them is a palindrome or a
            // single character
            if (s[i] == s[j]
                && (len == 2 || pal[i + 1][j - 1])) {

                // Mark the substring from i to j as a
                // palindrome
                pal[i][j] = true;
            }
        }
    }
}

// Function to calculate the minimum number of cuts required
```

```
// to make all substrings of 's' palindromic
int minCut(string s)
{
    if (s.empty())
        return 0;
    int n = s.size();

    // 2D vector to store whether substring [i, j] is a
    // palindrome
    vector<vector<bool> > pal(n, vector<bool>(n, false));

    generatePalindrome(s, pal);

    // vector to store minimum cuts required to make
    // substring [i, n-1] palindromic
    vector<int> minCutDp(n, INT_MAX);

    // There is no cut required for single character
    // as it is always palindrome
    minCutDp[0] = 0;

    // Iterate over the given string
    for (int i = 1; i < n; i++) {

        // Check if string 0 to i is palindrome.
        // Then minCut require will be 0.
        if (pal[0][i]) {
```

```
            minCutDp[i] = 0;
        }
        else {
            for (int j = i; j >= 1; j--) {

                // If s[i] and s[j] are equal and the inner
                // substring [i+1, j-1] is a palindrome or
                // it has a length of 1
                if (pal[j][i]) {

                    // Update the minimum cuts required if
                    // cutting at position 'j+1' results in
                    // a smaller value
                    if (minCutDp[j - 1] + 1 < minCutDp[i])
                        minCutDp[i] = minCutDp[j - 1] + 1;
                }
            }
        }
    }

    // Return the minimum cuts required for the entire
    // string 's'
    return minCutDp[n - 1];
}

int main()
{
```

```
    string str = "ababbbabbababa";

    int cuts = minCut(str);
    cout << "Minimum cuts required: " << cuts << endl;
    return 0;
}
```

Output

```
Minimum cuts required: 3
```

Time Complexity: $O(n^2)$
Auxiliary Space: $O(n^2)$