Palindrome Partitioning - GeeksforGeeks

Source: https://www.geeksforgeeks.org/palindrome-partitioning-dp-17/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Palindrome Partitioning Last Updated : 23 Jul, 2025 Given a string s , the task is to find the minimum number of cuts needed for palindrome partitioning of the given string. A partitioning of the string is a palindrome partitioning if every sub-string of the partition is a palindrome. Examples: Input: s = "geek" Output: 2 Explanation: We need to make minimum 2 cuts, i.e., "g | ee | k". Input: s= "aaaa" Output : 0 Explanation: The string is already a palindrome. Input: s = "ababbbabbababa" Output: 3 Explanation: We need to make minimum 3 cuts, i.e., "aba | bb | babbab | aba". Try it on GfG Practice Table of Content [Naive Approach] Using Recursion - O(n*2^n) Time and O(n) Space [Better Approach 1] Using Top-Down DP (Memoization) - O(n^3) Time and O(n^2) Space [Better Approach 2] Using Bottom-Up DP (Tabulation) - O(n^3) Time and O(n^2) Space [Expected Approach] Using Optimised Bottom-Up DP (Tabulation) - O(n^2) Time and O(n^2) Space [Naive Approach] Using Recursion - O(n*2^n) Time and O(n) Space 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: Base Case: If the current string is a palindrome, then we simply return 0 , no Partitioning is required. 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. C++ // C++ Program for Palindrome Partitioning Problem // using Recursion #include <iostream> #include <vector> #include <limits.h> using namespace std ; // Function to Check if a substring is a palindrome bool isPalindrome (string & s , int i , int j) { while (i < j) { if (s [i] != s [j]) return false ; i ++ ; j -- ; } return true ; } // Recursive Function to find the minimum number of // cuts needed for palindrome partitioning int palPartitionRec (string & s , int i , int j) { // Base case: If the substring is empty or // a palindrome, no cuts needed if (i >= j || isPalindrome (s , i , j)) return 0 ; int res = INT_MAX , cuts ; // Iterate through all possible partitions and // find the minimum cuts needed for (int k = i ; k < j ; k ++) { cuts = 1 + palPartitionRec (s , i , k) + palPartitionRec (s , k + 1 , j); res = min (res , cuts); } return res ; } int palPartition (string & s) { return palPartitionRec (s , 0 , s . size () -1); } int main () { string s = "geek" ; cout << palPartition (s) << endl ; return 0 ; } C // C Program for Palindrome Partitioning Problem // using Recursion #include <stdio.h> #include <string.h> #include <limits.h> // Function to check if a substring is a palindrome int isPalindrome (char * s , int i , int j) { while (i < j) { if (s [i] != s [j]) return 0 ; i ++ ; j -- ; } return 1 ; } // Recursive function to find the minimum number // of cuts needed for palindrome partitioning int palPartitionRec (char * s , int i , int j) { // Base case: If the substring is empty // or a palindrome, no cuts needed if (i >= j || isPalindrome (s , i , j)) return 0 ; int res = INT_MAX , cuts ; // Iterate through all possible partitions // and find the minimum cuts needed for (int k = i ; k < j ; k ++) { cuts = 1 + palPartitionRec (s , i , k) + palPartitionRec (s , k + 1 , j); if (cuts < res) res = cuts ; } return res ; } int palPartition (char * s) { return palPartitionRec (s , 0 , strlen (s) - 1); } int main () { char s [] = "geek" ; printf ("%d \n " , palPartition (s)); return 0 ; } Java // Java Program for Palindrome Partitioning Problem // using Recursion import java.util.* ; class GfG { // Function to check if a substring is a palindrome static boolean isPalindrome (String s , int i , int j) { while (i < j) { if (s . charAt (i) != s . charAt (j)) return false ; i ++ ; j -- ; } return true ; } // Recursive function to find the minimum number of // cuts needed for palindrome partitioning static int palPartitionRec (String s , int i , int j) { // Base case: If the substring is empty // or a palindrome, no cuts needed if (i >= j || isPalindrome (s , i , j)) return 0 ; int res = Integer . MAX_VALUE , cuts ; // Iterate through all possible partitions // and find the minimum cuts needed for (int k = i ; k < j ; k ++) { cuts = 1

+ palPartitionRec (s , i , k) + palPartitionRec (s , k + 1 , j); res = Math . min (res , cuts); } return res ; } static int palPartition (String s) { return palPartitionRec (s , 0 , s . length () - 1); } public static void main (String [] args) { String s = "geek" ; System . out . println (palPartition (s)); } } Python # Python Program for Palindrome Partitioning Problem # using Recursion import sys # Function to check if a substring is a palindrome def isPalindrome (s , i , j): while i < j : if s [i] != s [j]: return False i += 1 j -= 1 return True # Recursive function to find the minimum number # of cuts needed for palindrome partitioning def palPartitionRec (s , i , j): # Base case: If the substring is empty # or a palindrome, no cuts needed if i >= j or isPalindrome (s , i , j): return 0 res = sys . maxsize # Iterate through all possible partitions # and find the minimum cuts needed for k in range (i , j): cuts = 1 + palPartitionRec (s , i , k) \ + palPartitionRec (s , k + 1 , j) res = min (res , cuts) return res def palPartition (s): return palPartitionRec (s , 0 , len (s) - 1) if __name__ == "__main__" : s = "geek" print (palPartition (s)) C# // C# Program for Palindrome Partitioning Problem // using Recursion using System ; class GfG { // Function to check if a substring is a palindrome static bool isPalindrome (string s , int i , int j) { while (i < j) { if (s [i] != s [j]) return false ; i ++ ; j -- ; } return true ; } // Recursive function to find the minimum number of // cuts needed for palindrome partitioning static int palPartitionRec (string s , int i , int j) { // Base case: If the substring is empty // or a palindrome, no cuts needed if (i >= j || isPalindrome (s , i , j)) return 0 ; int res = int . MaxValue , cuts ; // Iterate through all possible partitions // and find the minimum cuts needed for (int k = i ; k < j ; k ++) { cuts = 1 + palPartitionRec (s , i , k) + palPartitionRec (s , k + 1 , j); res = Math . Min (res , cuts); } return res ; } static int palPartition (string s) { return palPartitionRec (s , 0 , s . Length - 1); } static void Main () { string s = "geek" ; Console . WriteLine (palPartition (s)); } } JavaScript // JavaScript Program for Palindrome Partitioning Problem // using Recursion // Function to check if a substring is a palindrome function isPalindrome (s , i , j) { while (i < j) { if (s [i] !== s [j]) return false ; i ++ ; j -- ; } return true ; } // Recursive function to find the minimum number // of cuts needed for palindrome partitioning function palPartitionRec (s , i , j) { // Base case: If the substring is empty // or a palindrome, no cuts needed if (i >= j || isPalindrome (s , i , j)) return 0 ; let res = Number . MAX_SAFE_INTEGER , cuts ; // Iterate through all possible partitions // and find the minimum cuts needed for (let k = i ; k < j ; k ++) { cuts = 1 + palPartitionRec (s , i , k) + palPartitionRec (s , k + 1 , j); res = Math . min (res , cuts); } return res ; } function palPartition (s) { return palPartitionRec (s , 0 , s . length - 1); } // Driver Code const s = "geek" ; console . log (palPartition (s)); Output 2 [Better Approach 1] Using Top-Down DP (Memoization) - O(n^3) Time and O(n^2) Space The above recursive approach has overlapping subproblems , leading to redundant computations thereby resulting in exponential time complexity . This redundant computation can be solved by using Memoization . To avoid redundant computations, we can memoize results of each subproblem and reuse them as needed. A 2D array can serve as a memoization table to store solutions for overlapping subproblems. The size of this memo table will be n*n , as there are n possible starting indices and n possible ending indices for any subarray. C++ // C++ Program for Palindrome Partitioning Problem // Using Top-Down DP #include <iostream> #include <vector> #include <limits.h> using namespace std ; // Function to Check if a substring is a palindrome bool isPalindrome (string & s , int i , int j) { while (i < j) { if (s [i] != s [j]) return false ; i ++ ; j -- ; } return true ; } // Recursive Function to find the minimum number of // cuts needed for palindrome partitioning int palPartitionRec (string & s , int i , int j , vector < vector < int >>& memo) { // check in memo for if (memo [i][j] != -1) return memo [i][j]; // Base case: If the substring is empty or // a palindrome, no cuts needed if (i >= j || isPalindrome (s , i , j)) return memo [i][j] = 0 ; int res = INT_MAX , cuts ; // Iterate through all possible partitions and // find the minimum cuts needed for (int k = i ; k < j ; k ++) { cuts = 1 + palPartitionRec (s , i , k , memo) + palPartitionRec (s , k + 1 , j , memo); res = min (res , cuts); } return memo [i][j] = res ; } int palPartition (string & s) { int n = s . size (); // declare a memo array to store the result // and initialise it with -1 vector < vector < int >> memo (n , vector < int > (n , -1)); return palPartitionRec (s , 0 , n - 1 , memo); } int main () { string s = "geek" ; cout << palPartition (s) << endl ; return 0 ; } Java // Java Program for Palindrome Partitioning Problem // Using Top-Down DP import java.util.Arrays ; class GfG { // Function to check if a substring is a palindrome static boolean isPalindrome (String s , int i , int j) { while (i < j) { if (s . charAt (i) != s . charAt (j)) return false ; i ++ ; j -- ; } return true ; } // Recursive function to find the minimum number of // cuts needed for palindrome partitioning static int palPartitionRec (String s , int i , int j , int [][] memo) { // check in memo for previously computed results if (memo [i][j] != - 1) return memo [i][j] ; // Base case: If the substring is empty or // a palindrome, no cuts needed if (i >= j || isPalindrome (s , i , j)) return memo [i][j] = 0 ; int res = Integer . MAX_VALUE , cuts ; // Iterate through all possible partitions and // find the minimum cuts needed for (int k = i ; k < j ; k ++) { cuts = 1 + palPartitionRec (s , i , k , memo) + palPartitionRec (s , k + 1 , j , memo); res = Math . min (res , cuts); } return memo [i][j] = res ; } static int palPartition (String s) { int n = s . length (); int [][] memo = new int [n][n] ; // Initialize memo array with -1 for (int [] row : memo) Arrays . fill (row , - 1); return palPartitionRec (s , 0 , n - 1 , memo); } public static void main (String [] args) { String s = "geek" ; System . out . println (palPartition (s)); } } Python # Python Program for Palindrome Partitioning Problem # Using Top-Down DP import sys # Function to check if a substring is a palindrome def isPalindrome (s , i , j): while i < j : if s [i] != s [j]: return False i += 1 j -= 1 return True # Recursive function to find the minimum number of # cuts needed for palindrome partitioning def palPartitionRec (s , i , j , memo): # Check memo for previously computed results if memo [i][j] != - 1 : return memo [i][j] # Base case: If the substring is empty or # a palindrome, no cuts needed if i >= j or isPalindrome (s , i , j): memo [i][j] = 0 return 0 res = sys . maxsize # Iterate through all possible partitions and # find the minimum cuts needed for k in range (i , j): cuts = 1 + palPartitionRec (s , i , k , memo) \ + palPartitionRec (s , k + 1 , j , memo) res = min (res , cuts) memo [i][j] = res return res def palPartition (s): n = len (s) memo = [[- 1 for _ in range (n)] for _ in range (n)] return palPartitionRec (s , 0 , n - 1 , memo) if __name__ == "__main__" : s = "geek" print (palPartition (s)) C# // C# Program for Palindrome Partitioning Problem // Usin Top-Down DP using System ; class GfG { // Function to check if a substring is a palindrome static bool isPalindrome (string s , int i , int j) { while (i < j) { if (s [i] != s [j]) return false ; i ++ ; j -- ; } return true ; } // Recursive function to find the minimum number of // cuts needed for palindrome partitioning static int palPartitionRec (string s , int i , int j , int [,] memo) { // Check memo for previously computed results if (memo [i , j] != - 1) return memo [i , j]; // Base case: If the substring is empty // or a palindrome, no cuts needed if (i >= j || isPalindrome (s , i , j)) return memo [i , j] = 0 ; int res = int . MaxValue , cuts ; // Iterate through all possible partitions // and find the minimum cuts needed for (int k = i ; k < j ; k ++) { cuts = 1 + palPartitionRec (s , i , k , memo) + palPartitionRec (s , k + 1 , j , memo); res = Math . Min (res , cuts); } return memo [i , j] = res ; } static int palPartition (string s) { int n = s . Length ; int [,] memo = new int [n , n]; // Initialize memo array with -1 for (int i = 0 ; i < n ; i ++) for (int j = 0 ; j < n ; j ++) memo [i , j] = - 1 ; return palPartitionRec (s , 0 , n - 1 , memo); } static void Main () { string s = "geek" ; Console . WriteLine (palPartition (s)); } } JavaScript // JavaScript Program for Palindrome Partitioning Problem // Using Top-Down DP // Function to check if a substring is a palindrome function isPalindrome (s , i , j) { while (i < j) { if (s [i] !== s [j]) return false ; i ++ ; j -- ; } return true ; } // Recursive function to find the minimum number of // cuts needed for palindrome partitioning function palPartitionRec (s , i , j , memo) { // Check memo for previously computed results if (memo [i][j] !== - 1) return memo [i][j]; // Base case: If the substring is empty or // a palindrome, no cuts needed if (i >= j || isPalindrome (s , i , j)) return memo [i][j] = 0 ; let res = Infinity , cuts ; // Iterate through all possible partitions // and find the minimum cuts needed for (let k = i ; k < j ; k ++) { cuts = 1 + palPartitionRec (s , i , k , memo) + palPartitionRec (s , k + 1 , j , memo); res = Math . min (res , cuts); } return memo [i][j] = res ; } function palPartition (s) { const n = s . length ; const memo = Array . from ({ length : n }, () => Array (n). fill (- 1)); return palPartitionRec (s , 0 , n - 1 , memo); } // Driver Code const s = "geek" ; console . log (palPartition (s)); Output 2 [Better Approach 2] Using Bottom-Up DP (Tabulation) - O(n^3) Time and O(n^2) Space The approach is similar to the previous one; just instead of breaking down the problem recursively, we iteratively build up the solution by calculating in bottom-up manner. Here, we can use two 2D array dp[][] and isPalin[][] , for storing the computed result. dp[i][j] stores the minimum cuts for palindrome partitioning of the substring s[i ... j] isPalin[i][j] tells us whether substring s[i ... j] is a palindromic string or not. We starts with smaller substrings and gradually builds up to the result for entire string. C++ // C++ Solution for Palindrome Partitioning Problem // using Bottom Up Dynamic Programming #include <iostream> #include <vector> #include <climits> using namespace std ; // Function to find the minimum number of cuts // needed to partition a string such that // every part is a palindrome int palPartition (string & s) { int n = s . length (); // dp[i][j] = Minimum number of cuts needed for // palindrome partitioning of substring s[i..j] int dp [n][n]; // isPalin[i][j] = true if substring s[i..j] // is palindrome,else false bool isPalin [n][n]; // Every substring of length 1 is a palindrome for (int i = 0 ; i < n ; i ++) { isPalin [i][i] = true ; dp [i][i] = 0 ; } for (int len = 2 ; len <= n ; len ++) { // Build solution for all substrings s[i ... j] // of length len for (int i = 0 , j = i + len - 1 ; j < n ; i ++ , j ++) { // If len is 2, then we just need to // compare two characters. if (len == 2) isPalin [i][j] = (s [i] == s [j]); // Else need to check two corner characters // and value of isPalin[i+1][j-1] else isPalin [i][j] = (s [i] == s [j]) && isPalin [i + 1][j - 1]; // IF s[i..j] is palindrome, then dp[i][j] is 0 if (isPalin [i][j] == true) dp [i][j] = 0 ; else { // Make a cut at every possible location starting // from i to j, and get the minimum cost cut. dp [i][j] = INT_MAX ; for (int k = i ; k <= j - 1 ; k ++) dp [i][j] = min (dp [i][j], 1 + dp [i][k] + dp [k + 1][j]); } } } // Return the min cut value for // complete string. i.e., s[0..n-1] return dp [0][n - 1]; } int main () {

string s = "geek" ; cout << palPartition (s) << endl ; return 0 ; } Java // Java Solution for Palindrome Partitioning Problem // using Bottom Up Dynamic Programming import java.util.Arrays ; class GfG { // Function to find the minimum number of cuts // needed to partition a string such that // every part is a palindrome static int palPartition (String s) { int n = s . length (); // dp[i][j] = Minimum number of cuts needed for // palindrome partitioning of substring s[i..j] int [][] dp = new int [n][n] ; // isPalin[i][j] = true if substring s[i..j] // is palindrome, else false boolean [][] isPalin = new boolean [n][n] ; // Every substring of length 1 is a palindrome for (int i = 0 ; i < n ; i ++) { isPalin [i][i] = true ; dp [i][i] = 0 ; } for (int len = 2 ; len <= n ; len ++) { // Build solution for all substrings s[i ... j] // of length len for (int i = 0 , j = i + len - 1 ; j < n ; i ++ , j ++) { // If len is 2, then we just need to // compare two characters. if (len == 2) isPalin [i][j] = (s . charAt (i) == s . charAt (j)); // Else need to check two corner characters // and value of isPalin[i+1][j-1] else isPalin [i][j] = (s . charAt (i) == s . charAt (j)) && isPalin [i + 1][j - 1] ; // IF s[i..j] is palindrome, then dp[i][j] is 0 if (isPalin [i][j]) dp [i][j] = 0 ; else { // Make a cut at every possible location starting // from i to j, and get the minimum cost cut. dp [i][j] = Integer . MAX_VALUE ; for (int k = i ; k <= j - 1 ; k ++) dp [i][j] = Math . min (dp [i][j] , 1 + dp [i][k] + dp [k + 1][j]); } } } // Return the min cut value for // complete string. i.e., s[0..n-1] return dp [0][n - 1] ; } public static void main (String [] args) { String s = "geek" ; System . out . println (palPartition (s)); } } Python # Python Solution for Palindrome Partitioning Problem # using Bottom Up Dynamic Programming # Function to find the minimum number of cuts # needed to partition a string such that # every part is a palindrome def palPartition (s): n = len (s) # dp[i][j] = Minimum number of cuts needed for # palindrome partitioning of substring s[i..j] dp = [[0] * n for _ in range (n)] # isPalin[i][j] = true if substring s[i..j] # is palindrome, else false isPalin = [[False] * n for _ in range (n)] # Every substring of length 1 is a palindrome for i in range (n): isPalin [i][i] = True dp [i][i] = 0 for length in range (2 , n + 1): # Build solution for all substrings s[i ... j] # of length len for i in range (n - length + 1): j = i + length - 1 # If len is 2, then we just need to # compare two characters. if length == 2 : isPalin [i][j] = (s [i] == s [j]) # Else need to check two corner characters # and value of isPalin[i+1][j-1] else : isPalin [i][j] = (s [i] == s [j]) and isPalin [i + 1][j - 1] # IF s[i..j] is palindrome, then dp[i][j] is 0 if isPalin [i][j]: dp [i][j] = 0 else : # Make a cut at every possible location starting # from i to j, and get the minimum cost cut. dp [i][j] = min (1 + dp [i][k] + dp [k + 1][j] for k in range (i , j)) # Return the min cut value for # complete string. i.e., s[0..n-1] return dp [0][n - 1] if __name__ == "__main__" : s = "geek" print (palPartition (s)) C# // C# Solution for Palindrome Partitioning Problem // using Bottom Up Dynamic Programming using System ; class GfG { // Function to find the minimum number of cuts // needed to partition a string such that // every part is a palindrome static int palPartition (string s) { int n = s . Length ; // dp[i][j] = Minimum number of cuts needed for // palindrome partitioning of substring s[i..j] int [,] dp = new int [n , n]; // isPalin[i][j] = true if substring s[i..j] // is palindrome, else false bool [,] isPalin = new bool [n , n]; // Every substring of length 1 is a palindrome for (int i = 0 ; i < n ; i ++) { isPalin [i , i] = true ; dp [i , i] = 0 ; } for (int len = 2 ; len <= n ; len ++) { // Build solution for all substrings s[i ... j] // of length len for (int i = 0 , j = i + len - 1 ; j < n ; i ++ , j ++) { // If len is 2, then we just need to // compare two characters. if (len == 2) isPalin [i , j] = (s [i] == s [j]); // Else need to check two corner characters // and value of isPalin[i+1][j-1] else isPalin [i , j] = (s [i] == s [j]) && isPalin [i + 1 , j - 1]; // IF s[i..j] is palindrome, then dp[i][j] is 0 if (isPalin [i , j]) dp [i , j] = 0 ; else { // Make a cut at every possible location starting // from i to j, and get the minimum cost cut. dp [i , j] = int . MaxValue ; for (int k = i ; k <= j - 1 ; k ++) dp [i , j] = Math . Min (dp [i , j], 1 + dp [i , k] + dp [k + 1 , j]); } } } // Return the min cut value for // complete string. i.e., s[0..n-1] return dp [0 , n - 1]; } static void Main () { string s = "geek" ; Console . WriteLine (palPartition (s)); } } JavaScript // JavaScript Solution for Palindrome Partitioning Problem // using Bottom Up Dynamic Programming // Function to find the minimum number of cuts // needed to partition a string such that // every part is a palindrome function palPartition (s) { let n = s . length ; // dp[i][j] = Minimum number of cuts needed for // palindrome partitioning of substring s[i..j] let dp = Array . from ({ length : n }, () => Array (n). fill (0)); // isPalin[i][j] = true if substring s[i..j] // is palindrome, else false let isPalin = Array . from ({ length : n }, () => Array (n). fill (false)); // Every substring of length 1 is a palindrome for (let i = 0 ; i < n ; i ++) { isPalin [i][i] = true ; dp [i][i] = 0 ; } for (let len = 2 ; len <= n ; len ++) { // Build solution for all substrings s[i ... j] // of length len for (let i = 0 ; i <= n - len ; i ++) { let j = i + len - 1 ; // If len is 2, then we just need to // compare two characters. if (len === 2) isPalin [i][j] = (s [i] === s [j]); // Else need to check two corner characters // and value of isPalin[i+1][j-1] else isPalin [i][j] = (s [i] === s [j]) && isPalin [i + 1][j - 1]; // IF s[i..j] is palindrome, then dp[i][j] is 0 if (isPalin [i][j]) dp [i][j] = 0 ; else { // Make a cut at every possible location starting // from i to j, and get the minimum cost cut. dp [i][j] = Infinity ; for (let k = i ; k <= j - 1 ; k ++) dp [i][j] = Math . min (dp [i][j], 1 + dp [i][k] + dp [k + 1][j]); } } } // Return the min cut value for // complete string. i.e., s[0..n-1] return dp [0][n - 1]; } // Driver Code let s = "geek" ; console . log (palPartition (s)); Output 2 [Expected Approach] Using Optimized Bottom-Up DP (Tabulation) - O(n^2) Time and O(n^2) Space Here we precompute a palindrome table isPalin[][], where i sPalin[i][j] = true if s[i..j] is a palindrome. This helps efficiently check palindrome partitions. Then we fill dp[], where dp[i] stores the minimum cuts needed to partition s[0..i] into palindromic substrings. If s[0..i] is a palindrome, no cut is needed (dp[i] = 0) . Otherwise, we check all valid partitions and update dp[i] accordingly. To fill dp[i], we find 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 minimizing in dp[i] . Similarly, we need to compute results for all such i . (1 <= i <= n – 1) and finally, dp[n - 1] will be the minimum number of cuts needed for palindrome partitioning of the given string. C++ // C++ Solution for Palindrome Partitioning Problem // using Optimised Bottom Up Dynamic Programming #include <iostream> #include <vector> #include <climits> using namespace std ; // Function to fill isPalin array such that isPalin[i][j] // stores whether substring s[i, j] is a palindrome or not void generatePal (string & s , vector < vector < bool >>& isPalin) { int n = s . size (); // Substring s[i .. i] of len 1 // is always palindromic for (int i = 0 ; i < n ; i ++) { isPalin [i][i] = true ; } // Iterate over different lengths of substrings for (int len = 2 ; len <= n ; len ++) { for (int i = 0 , j = i + len - 1 ; j < n ; i ++ , j ++) { // Check whether s[i] == s[j] and the // substring between them is a palindrome if (s [i] == s [j] && (len == 2 || isPalin [i + 1][j - 1])) { // Mark the substring from i to j as a // palindrome isPalin [i][j] = true ; } } } } // Function to calculate the minimum number of cuts required // to make all substrings of 's' palindromic int palPartition (string & s) { int n = s . size (); // 2D array to store whether substring // s[i, j] is a palindrome or not vector < vector < bool >> isPalin (n , vector < bool > (n , false)); generatePal (s , isPalin); // dp[i] stores minimum cuts for Palindrome // Partitioning of substring s[0...i] vector < int > dp (n , n); // There is no cut required for single character // as it is always palindrome dp [0] = 0 ; // Iterate over the given string for (int i = 1 ; i < n ; i ++) { // Check if string 0 to i is palindrome. if (isPalin [0][i]) { // if palindrome then cuts required is 0 dp [i] = 0 ; } else { for (int j = i ; j >= 1 ; j --) { // if substring s[j...i] is palindromic // then we can make a cut over here if (isPalin [j][i]) { // update dp[i] with minimum cuts dp [i] = min (dp [i], 1 + dp [j -1]); } } } } // Return the minimum cuts required // for the entire string 's' return dp [n - 1]; } int main () { string s = "geek" ; cout << palPartition (s) << endl ; return 0 ; } Java // Java Solution for Palindrome Partitioning Problem // using Optimised Bottom Up Dynamic Programming import java.util.Arrays ; class GfG { // Function to fill isPalin array such that isPalin[i][j] // stores whether substring s[i, j] is a palindrome or not static void generatePal (String s , boolean [][] isPalin) { int n = s . length (); // Substring s[i .. i] of len 1 // is always palindromic for (int i = 0 ; i < n ; i ++) { isPalin [i][i] = true ; } // Iterate over different lengths of substrings for (int len = 2 ; len <= n ; len ++) { for (int i = 0 , j = i + len - 1 ; j < n ; i ++ , j ++) { // Check whether s[i] == s[j] and the // substring between them is a palindrome if (s . charAt (i) == s . charAt (j) && (len == 2 || isPalin [i + 1][j - 1])) { // Mark the substring from i to j as a // palindrome isPalin [i][j] = true ; } } } } // Function to calculate the minimum number of cuts required // to make all substrings of 's' palindromic static int palPartition (String s) { int n = s . length (); // 2D array to store whether substring // s[i, j] is a palindrome or not boolean [][] isPalin = new boolean [n][n] ; generatePal (s , isPalin); // dp[i] stores minimum cuts for Palindrome // Partitioning of substring s[0...i] int [] dp = new int [n] ; Arrays . fill (dp , n); // There is no cut required for single character // as it is always palindrome dp [0] = 0 ; // Iterate over the given string for (int i = 1 ; i < n ; i ++) { // Check if string 0 to i is palindrome. if (isPalin [0][i]) { // if palindrome then cuts required is 0 dp [i] = 0 ; } else { for (int j = i ; j >= 1 ; j --) { // if substring s[j...i] is palindromic // then we can make a cut over here if (isPalin [j][i]) { // update dp[i] with minimum cuts dp [i] = Math . min (dp [i] , 1 + dp [j - 1]); } } } } // Return the minimum cuts required // for the entire string 's' return dp [n - 1] ; } public static void main (String [] args) { String s = "geek" ; System . out . println (palPartition (s)); } } Python # Python Solution for Palindrome Partitioning Problem # using Optimised Bottom Up Dynamic Programming # Function to fill isPalin array such that isPalin[i][j] # stores whether substring s[i, j] is a palindrome or not def generatePal (s , isPalin): n = len (s) # Substring s[i .. i] of len 1 # is always palindromic for i in range (n): isPalin [i][i] = True # Iterate over different lengths of substrings for length in range (2 , n + 1): for i in range (n - length + 1): j = i + length - 1 # Check whether s[i] == s[j] and the # substring between them is a palindrome if s [i] == s [j] and (length == 2 or isPalin [i + 1][j - 1]): # Mark the substring from i to j as a # palindrome isPalin [i][j] = True # Function to calculate the minimum number of cuts required # to make all substrings of 's' palindromic def palPartition (s): n = len (s) # 2D array to store whether substring # s[i, j] is a palindrome or not isPalin = [[False] * n for _ in range (n)] generatePal (s , isPalin) # dp[i] stores minimum cuts for Palindrome # Partitioning of substring s[0...i] dp = [n] * n # There is no cut required for single character # as it is always palindrome dp [0] = 0 # Iterate over the given string for i in range (1 , n): # Check if string 0 to i is palindrome. if isPalin [0][i]: # if palindrome then cuts required is 0 dp [i] = 0 else : for j in range (i , 0 , - 1): # if substring s[j...i] is palindromic # then we can make a cut over here if isPalin [j][i]: # update dp[i] with minimum cuts dp [i] = min (dp [i], 1 + dp [j - 1]) # Return the minimum cuts required # for the entire string 's' return dp [n - 1] if __name__ == "__main__" : s = "geek" print (palPartition (s)) C# // C# Solution for Palindrome Partitioning Problem // using Optimised Bottom Up Dynamic Programming using System ; class GfG { // Function to fill isPalin array such that isPalin[i][j] // stores whether substring s[i, j] is a palindrome or not static void GeneratePal (string s , bool [,] isPalin) { int n = s . Length ; // Substring s[i .. i] of len 1 // is always palindromic for (int i = 0 ; i < n ; i ++) { isPalin [i , i] = true ; } // Iterate over different lengths of substrings for (int len = 2 ; len <= n ; len ++) { for (int i = 0 , j = i + len - 1 ; j < n ; i ++ , j ++) { // Check whether s[i] == s[j] and the // substring between them is a palindrome if (s [i] == s [j] && (len == 2 || isPalin [i + 1 , j - 1])) { // Mark the substring from i to j as a // palindrome isPalin [i , j] = true ; } } } } // Function to calculate the minimum number of cuts required // to make all substrings of 's' palindromic static int palPartition (string s) { int n = s . Length ; // 2D array to store whether substring // s[i, j] is a palindrome or not bool [,] isPalin = new bool [n , n]; GeneratePal (s , isPalin); // dp[i] stores minimum cuts for Palindrome // Partitioning of substring s[0...i] int [] dp = new int [n]; Array . Fill (dp , n); // There is no cut required for single character // as it is always palindrome dp [0] = 0 ; // Iterate over the given string for (int i = 1 ; i < n ; i ++) { // Check if string 0 to i is palindrome. if (isPalin [0 , i]) { // if palindrome then cuts required is 0 dp [i] = 0 ; } else { for (int j = i ; j >= 1 ; j --) { // if substring s[j...i] is palindromic // then we can make a cut over here if (isPalin [j , i]) { // update dp[i] with minimum cuts dp [i] = Math . Min (dp [i], 1 + dp [j - 1]); } } } } // Return the minimum cuts required // for the entire string 's' return dp [n - 1]; } static void Main () { string s = "geek" ; Console . WriteLine (palPartition (s)); } } JavaScript // JavaScript Solution for Palindrome Partitioning Problem // using Optimised Bottom Up Dynamic Programming // Function to fill isPalin array such that isPalin[i][j] // stores whether substring s[i, j] is a palindrome or not function generatePal (s , isPalin) { let n = s . length ; // Substring s[i .. i] of len 1 // is always palindromic for (let i = 0 ; i < n ; i ++) { isPalin [i][i] = true ; } // Iterate over different lengths of substrings for (let len = 2 ; len <= n ; len ++) { for (let i = 0 , j = i + len - 1 ; j < n ; i ++ , j ++) { // Check whether s[i] == s[j] and the // substring between them is a palindrome if (s [i] === s [j] && (len === 2 || isPalin [i + 1][j - 1])) { // Mark the substring from i to j as a // palindrome isPalin [i][j] = true ; } } } } // Function to calculate the minimum number of cuts required // to make all substrings of 's' palindromic function palPartition (s) { let n = s . length ; // 2D array to store whether substring // s[i, j] is a palindrome or not let isPalin = Array . from ({ length : n }, () => Array (n). fill (false)); generatePal (s , isPalin); // dp[i] stores minimum cuts for Palindrome // Partitioning of substring s[0...i] let dp = new Array (n). fill (n); // There is no cut required for single character // as it is always palindrome dp [0] = 0 ; // Iterate over the given string for (let i = 1 ; i < n ; i ++) { // Check if string 0 to i is palindrome. if (isPalin [0][i]) { // if palindrome then cuts required is 0 dp [i] = 0 ; } else { for (let j = i ; j >= 1 ; j --) { // if substring s[j...i] is palindromic // then we can make a cut over here if (isPalin [j][i]) { // update dp[i] with minimum cuts dp [i] = Math . min (dp [i], 1 + dp [j - 1]); } } } } // Return the minimum cuts required // for the entire string 's' return dp [n - 1]; } // Driver Code let s = "geek" ; console . log (palPartition (s)); Output 2 Comment Article Tags: Article Tags: Strings Dynamic Programming DSA Amazon palindrome + 1 More