Longest Palindromic Subsequence (LPS) - GeeksforGeeks

Source: https://www.geeksforgeeks.org/longest-palindromic-subsequence-dp-12/

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 Longest Palindromic Subsequence (LPS) Last Updated : 23 Jul, 2025 Given a string s , find the length of the Longest Palindromic Subsequence in it. Note: The Longest Palindromic Subsequence (LPS) is the maximum-length subsequence of a given string that is also a Palindrome. Longest Palindromic Subsequence Examples: Input: s = "bbabcbcab" Output : 7 Explanation : Subsequence "babcbab" is the longest subsequence which is also a palindrome. Input: s = "abcd" Output: 1 Explanation: "a", "b", "c" and "d" are palindromic and all have a length 1. Try it on GfG Practice Table of Content [Naive Approach] - Using Recursion - O(2^n) Time and O(1) Space [Better Approach 1] Using Memoization - O(n^2) Time and O(n^2) Space [Better Approach 2] Using Tabulation - O(n^2) Time and O(n^2) Space [Expected Approach] Using Tabulation - O(n^2) Time and O(n) Space [Alternate Approach] Using Longest Common Subsequence - O(n^2) Time and O(n) Space [Naive Approach] - Using Recursion - O(2^n) Time and O(1) Space The idea is to recursively generate all possible subsequences of the given string s and find the longest palindromic subsequence. To do so, create two counters low and high and set them to point to first and last character of string s . Start matching the characters from both the ends of the string. For each case, there are two possibilities: If characters are matching , increment the value low and decrement the value high by 1 and recur to find the LPS of new substring. And return the value result + 2. Else make two recursive calls for (low + 1, hi) and (lo, hi-1). And return the max of 2 calls. C++ // C++ program to find the lps #include <bits/stdc++.h> using namespace std ; // Returns the length of the longest // palindromic subsequence in seq int lps (const string & s , int low , int high) { // Base case if (low > high) return 0 ; // If there is only 1 character if (low == high) return 1 ; // If the first and last characters match if (s [low] == s [high]) return lps (s , low + 1 , high - 1) + 2 ; // If the first and last characters do not match return max (lps (s , low , high - 1), lps (s , low + 1 , high)); } int longestPalinSubseq (string & s) { return lps (s , 0 , s . size () - 1); } int main () { string s = "bbabcbcab" ; cout << longestPalinSubseq (s); return 0 ; } C // C program to find the lps #include <stdio.h> #include <string.h> // Returns the length of the longest // palindromic subsequence in seq int lps (const char * s , int low , int high) { // Base case if (low > high) return 0 ; // If there is only 1 character if (low == high) return 1 ; // If the first and last characters match if (s [low] == s [high]) return lps (s , low + 1 , high - 1) + 2 ; // If the first and last characters do not match int a = lps (s , low , high - 1); int b = lps (s , low + 1 , high); return (a > b) ? a : b ; } int longestPalinSubseq (char * s) { int n = strlen (s); return lps (s , 0 , n - 1); } int main () { char s [] = "bbabcbcab" ; printf ("%d" , longestPalinSubseq (s)); return 0 ; } Java // Java program to find the lps class GfG { // Returns the length of the longest // palindromic subsequence in seq static int lps (String s , int low , int high) { // Base case if (low > high) return 0 ; // If there is only 1 character if (low == high) return 1 ; // If the first and last characters match if (s . charAt (low) == s . charAt (high)) return lps (s , low + 1 , high - 1) + 2 ; // If the first and last characters do not match return Math . max (lps (s , low , high - 1), lps (s , low + 1 , high)); } static int longestPalinSubseq (String s) { return lps (s , 0 , s . length () - 1); } public static void main (String [] args) { String s = "bbabcbcab" ; System . out . println (longestPalinSubseq (s)); } } Python # Python program to find the lps # Returns the length of the longest # palindromic subsequence in seq def lps (s , low , high): # Base case if low > high : return 0 # If there is only 1 character if low == high : return 1 # If the first and last characters match if s [low] == s [high]: return lps (s , low + 1 , high - 1) + 2 # If the first and last

characters do not match return max (lps (s , low , high - 1), lps (s , low + 1 , high)) def longestPalinSubseq (s): return lps (s , 0 , len (s) - 1) if __name__ == "__main__" : s = "bbabcbcab" print (longestPalinSubseq (s)) C# // C# program to find the lps using System ; class GfG { // Returns the length of the longest // palindromic subsequence in seq static int lps (string s , int low , int high) { // Base case if (low > high) return 0 ; // If there is only 1 character if (low == high) return 1 ; // If the first and last characters match if (s [low] == s [high]) return lps (s , low + 1 , high - 1) + 2 ; // If the first and last characters do not match return Math . Max (lps (s , low , high - 1), lps (s , low + 1 , high)); } static int longestPalinSubseq (string s) { return lps (s , 0 , s . Length - 1); } static void Main (string [] args) { string s = "bbabcbcab" ; Console . WriteLine (longestPalinSubseq (s)); } } JavaScript // JavaScript program to find the lps // Returns the length of the longest // palindromic subsequence in seq function lps (s , low , high) { // Base case if (low > high) return 0 ; // If there is only 1 character if (low === high) return 1 ; // If the first and last characters match if (s [low] === s [high]) return lps (s , low + 1 , high - 1) + 2 ; // If the first and last characters do not match return Math . max (lps (s , low , high - 1), lps (s , low + 1 , high)); } function longestPalinSubseq (s) { return lps (s , 0 , s . length - 1); } const s = "bbabcbcab" ; console . log (longestPalinSubseq (s)); Output 7 [Better Approach 1] Using Memoization - O(n^2) Time and O(n^2) Space In the above approach, lps() function is calculating the same substring multiple times. The idea is to use memoization to store the result of subproblems thus avoiding repetition. To do so, create a 2d array memo[][] of order n*n , where memo[i][j] stores the length of LPS of substring s[i] to s[j] . At each step, check if the substring is already calculated , if so return the stored value else operate as in above approach. C++ // C++ program to find the lps #include <bits/stdc++.h> using namespace std ; // Returns the length of the longest // palindromic subsequence in seq int lps (const string & s , int low , int high , vector < vector < int >> & memo) { // Base case if (low > high) return 0 ; // If there is only 1 character if (low == high) return 1 ; // If the value is already calculated if (memo [low][high] != -1) return memo [low][high]; // If the first and last characters match if (s [low] == s [high]) return memo [low][high] = lps (s , low + 1 , high - 1 , memo) + 2 ; // If the first and last characters do not match return memo [low][high] = max (lps (s , low , high - 1 , memo), lps (s , low + 1 , high , memo)); } int longestPalinSubseq (string & s) { // create memoization table vector < vector < int >> memo (s . size (), vector < int > (s . size (), -1)); return lps (s , 0 , s . size () - 1 , memo); } int main () { string s = "bbabcbcab" ; cout << longestPalinSubseq (s); return 0 ; } Java // Java program to find the lps class GfG { // Returns the length of the longest // palindromic subsequence in seq static int lps (String s , int low , int high , int [][] memo) { // Base case if (low > high) return 0 ; // If there is only 1 character if (low == high) return 1 ; // If the value is already calculated if (memo [low][high] != - 1) return memo [low][high] ; // If the first and last characters match if (s . charAt (low) == s . charAt (high)) return memo [low][high] = lps (s , low + 1 , high - 1 , memo) + 2 ; // If the first and last characters do not match return memo [low][high] = Math . max (lps (s , low , high - 1 , memo), lps (s , low + 1 , high , memo)); } static int longestPalinSubseq (String s) { // create memoization table int n = s . length (); int [][] memo = new int [n][n] ; for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < n ; j ++) { memo [i][j] = - 1 ; } } return lps (s , 0 , n - 1 , memo); } public static void main (String [] args) { String s = "bbabcbcab" ; System . out . println (longestPalinSubseq (s)); } } Python # Python program to find the lps # Returns the length of the longest # palindromic subsequence in seq def lps (s , low , high , memo): # Base case if low > high : return 0 # If there is only 1 character if low == high : return 1 # If the value is already calculated if memo [low][high] != - 1 : return memo [low][high] # If the first and last characters match if s [low] == s [high]: memo [low][high] = lps (s , low + 1 , high - 1 , memo) + 2 else : # If the first and last characters do not match memo [low][high] = max (lps (s , low , high - 1 , memo), lps (s , low + 1 , high , memo)) return memo [low][high] def longestPalinSubseq (s): n = len (s) memo = [[- 1 for _ in range (n)] for _ in range (n)] return lps (s , 0 , n - 1 , memo) if __name__ == "__main__" : s = "bbabcbcab" print (longestPalinSubseq (s)) C# // C# program to find the lps using System ; class GfG { // Returns the length of the longest // palindromic subsequence in seq static int lps (string s , int low , int high , int [,] memo) { // Base case if (low > high) return 0 ; // If there is only 1 character if (low == high) return 1 ; // If the value is already calculated if (memo [low , high] != - 1) return memo [low , high]; // If the first and last characters match if (s [low] == s [high]) return memo [low , high] = lps (s , low + 1 , high - 1 , memo) + 2 ; // If the first and last characters do not match return memo [low , high] = Math . Max (lps (s , low , high - 1 , memo), lps (s , low + 1 , high , memo)); } static int longestPalinSubseq (string s) { // create memoization table int n = s . Length ; int [,] memo = new int [n , n]; for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < n ; j ++) { memo [i , j] = - 1 ; } } return lps (s , 0 , n - 1 , memo); } static void Main (string [] args) { string s = "bbabcbcab" ; Console . WriteLine (longestPalinSubseq (s)); } } JavaScript // JavaScript program to

find the lps // Returns the length of the longest // palindromic subsequence in seq function lps (s , low , high , memo) { // Base case if (low > high) return 0 ; // If there is only 1 character if (low === high) return 1 ; // If the value is already calculated if (memo [low][high] !== - 1) return memo [low][high]; // If the first and last characters match if (s [low] === s [high]) { memo [low][high] = lps (s , low + 1 , high - 1 , memo) + 2 ; } else { // If the first and last characters do not match memo [low][high] = Math . max (lps (s , low , high - 1 , memo), lps (s , low + 1 , high , memo)); } return memo [low][high]; } function longestPalinSubseq (s) { const n = s . length ; const memo = Array . from ({ length : n }, () => Array (n). fill (- 1)); return lps (s , 0 , n - 1 , memo); } const s = "bbabcbcab" ; console . log (longestPalinSubseq (s)); Output 7 [Better Approach 2] Using Tabulation - O(n^2) Time and O(n^2) Space The above approach can be implemented using tabulation to minimize the auxiliary space required for recursive stack. The idea is create a 2d array dp[][] of order n*n, where element dp[i][j] stores the length of LPS of substring s[i] to s[j] . Start from the smaller substring and try to build answers for longer ones. At each step there are two possibilities: if s[i] == s[j], then dp[i][j] = dp[i+1][j-1] + 2 else, dp[i][j] = max(dp[i+1][j], dp[i][j-1]) dp[0][n-1] stores the length of the longest palindromic subsequence of string s. C++ // C++ program to find the lps #include <bits/stdc++.h> using namespace std ; // Function to find the LPS int longestPalinSubseq (string & s) { int n = s . length (); // Create a DP table vector < vector < int >> dp (n , vector < int > (n)); // Build the DP table for all the substrings for (int i = n - 1 ; i >= 0 ; i --) { for (int j = i ; j < n ; j ++) { // If there is only one character if (i == j){ dp [i][j] = 1 ; continue ; } // If characters at position i and j are the same if (s [i] == s [j]) { if (i + 1 == j) dp [i][j] = 2 ; else dp [i][j] = dp [i + 1][j - 1] + 2 ; } else { // Otherwise, take the maximum length // from either excluding i or j dp [i][j] = max (dp [i + 1][j], dp [i][j - 1]); } } } // The final answer is stored in dp[0][n-1] return dp [0][n - 1]; } int main () { string s = "bbabcbcab" ; cout << longestPalinSubseq (s); return 0 ; } Java // Java program to find the lps class GfG { // Function to find the LPS static int longestPalinSubseq (String s) { int n = s . length (); // Create a DP table int [][] dp = new int [n][n] ; // Build the DP table for all the substrings for (int i = n - 1 ; i >= 0 ; i --) { for (int j = i ; j < n ; j ++) { // If there is only one character if (i == j) { dp [i][j] = 1 ; continue ; } // If characters at position i and j are the same if (s . charAt (i) == s . charAt (j)) { if (i + 1 == j) dp [i][j] = 2 ; else dp [i][j] = dp [i + 1][j - 1] + 2 ; } else { // Otherwise, take the maximum length // from either excluding i or j dp [i][j] = Math . max (dp [i + 1][j] , dp [i][j - 1]); } } } // The final answer is stored in dp[0][n-1] return dp [0][n - 1] ; } public static void main (String [] args) { String s = "bbabcbcab" ; System . out . println (longestPalinSubseq (s)); } } Python # Python program to find the lps # Function to find the LPS def longestPalinSubseq (s): n = len (s) # Create a DP table dp = [[0] * n for _ in range (n)] # Build the DP table for all the substrings for i in range (n - 1 , - 1 , - 1): for j in range (i , n): # If there is only one character if i == j : dp [i][j] = 1 continue # If characters at position i and j are the same if s [i] == s [j]: if i + 1 == j : dp [i][j] = 2 else : dp [i][j] = dp [i + 1][j - 1] + 2 else : # Otherwise, take the maximum length # from either excluding i or j dp [i][j] = max (dp [i + 1][j], dp [i][j - 1]) # The final answer is stored in dp[0][n-1] return dp [0][n - 1] if __name__ == "__main__" : s = "bbabcbcab" print (longestPalinSubseq (s)) C# // C# program to find the lps using System ; class GfG { // Function to find the LPS static int longestPalinSubseq (string s) { int n = s . Length ; // Create a DP table int [,] dp = new int [n , n]; // Build the DP table for all the substrings for (int i = n - 1 ; i >= 0 ; i --) { for (int j = i ; j < n ; j ++) { // If there is only one character if (i == j) { dp [i , j] = 1 ; continue ; } // If characters at position i and j are the same if (s [i] == s [j]) { if (i + 1 == j) dp [i , j] = 2 ; else dp [i , j] = dp [i + 1 , j - 1] + 2 ; } else { // Otherwise, take the maximum length // from either excluding i or j dp [i , j] = Math . Max (dp [i + 1 , j], dp [i , j - 1]); } } } // The final answer is stored in dp[0][n-1] return dp [0 , n - 1]; } static void Main (string [] args) { string s = "bbabcbcab" ; Console . WriteLine (longestPalinSubseq (s)); } } JavaScript // JavaScript program to find the lps // Function to find the LPS function longestPalinSubseq (s) { const n = s . length ; // Create a DP table const dp = Array . from ({ length : n }, () => Array (n). fill (0)); // Build the DP table for all the substrings for (let i = n - 1 ; i >= 0 ; i --) { for (let j = i ; j < n ; j ++) { // If there is only one character if (i === j) { dp [i][j] = 1 ; continue ; } // If characters at position i and j are the same if (s [i] === s [j]) { if (i + 1 === j) dp [i][j] = 2 ; else dp [i][j] = dp [i + 1][j - 1] + 2 ; } else { // Otherwise, take the maximum length // from either excluding i or j dp [i][j] = Math . max (dp [i + 1][j], dp [i][j - 1]); } } } // The final answer is stored in dp[0][n-1] return dp [0][n - 1]; } const s = "bbabcbcab" ; console . log (longestPalinSubseq (s)); Output 7 [Expected Approach] Using Tabulation - O(n^2) Time and O(n) Space In the above approach, for calculating the LPS of substrings starting from index i , only the LPS of substrings starting from index i+1 are required. Thus instead of creating 2d array, idea is to create two arrays of size, curr[] and prev[] , where curr[j] stores the lps of substring from s[i] to s[j] , while prev[j] stores the lps of substring from s[i+1] to s[j] . Else everything will

be similar to above approach. C++ // C++ program to find longest // palindromic subsequence #include <bits/stdc++.h> using namespace std ; // Function to find the length of the lps int longestPalinSubseq (const string & s) { int n = s . size (); // Create two vectors: one for the current state (dp) // and one for the previous state (dpPrev) vector < int > curr (n), prev (n); // Loop through the string in reverse (starting from the end) for (int i = n - 1 ; i >= 0 ; -- i){ // Initialize the current state of dp curr [i] = 1 ; // Loop through the characters ahead of i for (int j = i + 1 ; j < n ; ++ j){ // If the characters at i and j are the same if (s [i] == s [j]){ // Add 2 to the length of the palindrome between them curr [j] = prev [j - 1] + 2 ; } else { // Take the maximum between excluding either i or j curr [j] = max (prev [j], curr [j - 1]); } } // Update previous to the current state of dp prev = curr ; } return curr [n -1]; } int main () { string s = "bbabcbcab" ; cout << longestPalinSubseq (s); return 0 ; } Java // Java program to find longest // palindromic subsequence import java.util.* ; // Java program to find the length of the lps class GfG { // Function to find the length of the lps static int longestPalinSubseq (String s) { int n = s . length (); // Create two arrays: one for the current state (dp) // and one for the previous state (dpPrev) int [] curr = new int [n] ; int [] prev = new int [n] ; // Loop through the string in reverse (starting from the end) for (int i = n - 1 ; i >= 0 ; -- i) { // Initialize the current state of dp curr [i] = 1 ; // Loop through the characters ahead of i for (int j = i + 1 ; j < n ; ++ j) { // If the characters at i and j are the same if (s . charAt (i) == s . charAt (j)) { // Add 2 to the length of the palindrome between them curr [j] = prev [j - 1] + 2 ; } else { // Take the maximum between excluding either i or j curr [j] = Math . max (prev [j] , curr [j - 1]); } } // Update previous to the current state of dp prev = curr . clone (); } return curr [n - 1] ; } public static void main (String [] args) { String s = "bbabcbcab" ; System . out . println (longestPalinSubseq (s)); } } Python # Python program to find the length of the lps # Function to find the length of the lps def longestPalinSubseq (s): n = len (s) # Create two arrays: one for the current state (dp) # and one for the previous state (dpPrev) curr = [0] * n prev = [0] * n # Loop through the string in reverse (starting from the end) for i in range (n - 1 , - 1 , - 1): # Initialize the current state of dp curr [i] = 1 # Loop through the characters ahead of i for j in range (i + 1 , n): # If the characters at i and j are the same if s [i] == s [j]: # Add 2 to the length of the palindrome between them curr [j] = prev [j - 1] + 2 else : # Take the maximum between excluding either i or j curr [j] = max (prev [j], curr [j - 1]) # Update previous to the current state of dp prev = curr [:] return curr [n - 1] if __name__ == "__main__" : s = "bbabcbcab" print (longestPalinSubseq (s)) C# // C# program to find longest // palindromic subsequence using System ; // C# program to find the length of the lps class GfG { // Function to find the length of the lps static int longestPalinSubseq (string s) { int n = s . Length ; // Create two arrays: one for the current state (dp) // and one for the previous state (dpPrev) int [] curr = new int [n]; int [] prev = new int [n]; // Loop through the string in reverse (starting from the end) for (int i = n - 1 ; i >= 0 ; -- i) { // Initialize the current state of dp curr [i] = 1 ; // Loop through the characters ahead of i for (int j = i + 1 ; j < n ; ++ j) { // If the characters at i and j are the same if (s [i] == s [j]) { // Add 2 to the length of the palindrome between them curr [j] = prev [j - 1] + 2 ; } else { // Take the maximum between excluding either i or j curr [j] = Math . Max (prev [j], curr [j - 1]); } } // Update previous to the current state of dp Array . Copy (curr , prev , n); } return curr [n - 1]; } static void Main (string [] args) { string s = "bbabcbcab" ; Console . WriteLine (longestPalinSubseq (s)); } } JavaScript // JavaScript program to find the length of the lps // Function to find the length of the lps function longestPalinSubseq (s) { const n = s . length ; // Create two arrays: one for the current state (dp) // and one for the previous state (dpPrev) let curr = new Array (n). fill (0); let prev = new Array (n). fill (0); // Loop through the string in reverse (starting from the end) for (let i = n - 1 ; i >= 0 ; -- i) { // Initialize the current state of dp curr [i] = 1 ; // Loop through the characters ahead of i for (let j = i + 1 ; j < n ; ++ j) { // If the characters at i and j are the same if (s [i] === s [j]) { // Add 2 to the length of the palindrome between them curr [j] = prev [j - 1] + 2 ; } else { // Take the maximum between excluding either i or j curr [j] = Math . max (prev [j], curr [j - 1]); } } // Update previous to the current state of dp prev = [... curr]; } return curr [n - 1]; } const s = "bbabcbcab" ; console . log (longestPalinSubseq (s)); Output 7 [Alternate Approach] Using Longest Common Subsequence - O(n^2) Time and O(n) Space The idea is to reverse the given string s and find the length of the longest common subsequence of original and reversed string. Comment Article Tags: Article Tags: Strings Dynamic Programming DSA Amazon Linkedin PayPal palindrome subsequence LCS Rivigo strings + 7 More