Longest Common Subsequence (LCS) - GeeksforGeeks

Source: https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/

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 Common Subsequence (LCS) Last Updated : 23 Jul, 2025 Given two strings, s1 and s2 , the task is to find the length of the Longest Common Subsequence. If there is no common subsequence , return 0 . A subsequence is a string generated from the original string by deleting 0 or more characters, without changing the relative order of the remaining characters. For example, subsequences of "ABC" are "", "A", "B", "C", "AB", "AC", "BC" and "ABC". In general, a string of length n has 2 n subsequences. Examples: Input: s1 = "ABC", s2 = "ACD" Output: 2 Explanation: The longest subsequence which is present in both strings is "AC". Input: s1 = "AGGTAB", s2 = "GXTXAYB" Output: 4 Explanation: The longest common subsequence is "GTAB". Input: s1 = "ABC", s2 = "CBA" Output: 1 Explanation: There are three longest common subsequences of length 1, "A", "B" and "C". Try it on GfG Practice Table of Content [Naive Approach] Using Recursion - O(2 ^ min(m, n)) Time and O(min(m, n)) Space [Better Approach] Using Memoization - O(m * n) Time and O(m * n) Space [Expected Approach 1] Using Bottom-Up DP (Tabulation) - O(m * n) Time and O(m * n) Space [Expected Approach 2] Using Bottom-Up DP (Space-Optimization): Applications of LCS Problems based on LCS [Naive Approach] Using Recursion - O(2 ^ min(m, n)) Time and O(min(m, n)) Space The idea is to compare the last characters of s1 and s2 . While comparing the strings s1 and s2 two cases arise: Match : Make the recursion call for the remaining strings (strings of lengths m-1 and n-1) and add 1 to result. Do not Match : Make two recursive calls. First for lengths m-1 and n , and second for m and n-1. Take the maximum of two results. Base case : If any of the strings become empty, we return 0. For example , consider the input strings s1 = "ABX" and s2 = "ACX". LCS("ABX", "ACX") = 1 + LCS("AB", "AC") [Last Characters Match] LCS("AB", "AC") = max(LCS("A", "AC") , LCS("AB", "A")) [Last Characters Do Not Match] LCS("A", "AC") = max(LCS("", "AC") , LCS("A", "A")) = max(0, 1 + LCS("", "")) = 1 LCS("AB", "A") = max(LCS("A", "A") , LCS("AB", "")) = max(1 + LCS("", "", 0)) = 1 So overall result is 1 + 1 = 2 C++ // A Naive recursive implementation of LCS problem #include <bits/stdc++.h> using namespace std ; // Returns length of LCS for s1[0..m-1], s2[0..n-1] int lcsRec (string & s1 , string & s2 , int m , int n) { // Base case: If either string is empty, the length of LCS is 0 if (m == 0 || n == 0) return 0 ; // If the last characters of both substrings match if (s1 [m - 1] == s2 [n - 1]) // Include this character in LCS and recur for remaining substrings return 1 + lcsRec (s1 , s2 , m - 1 , n - 1); else // If the last characters do not match // Recur for two cases: // 1. Exclude the last character of s1 // 2. Exclude the last character of s2 // Take the maximum of these two recursive calls return max (lcsRec (s1 , s2 , m , n - 1), lcsRec (s1 , s2 , m - 1 , n)); } int lcs (string & s1 , string & s2){ int m = s1 . size (), n = s2 . size (); return lcsRec (s1 , s2 , m , n); } int main () { string s1 = "AGGTAB" ; string s2 = "GXTXAYB" ; int m = s1 . size (); int n = s2 . size (); cout << lcs (s1 , s2) << endl ; return 0 ; } C // A Naive recursive implementation of LCS problem #include <stdio.h> #include <string.h> int max (int x , int y) { return x > y ? x : y ; } // Returns length of LCS for s1[0..m-1], s2[0..n-1] int lcsRec (char s1 [], char s2 [], int m , int n) { // Base case: If either string is empty, the length of LCS is 0 if (m == 0 || n == 0) return 0 ; // If the last characters of both substrings match if (s1 [m - 1] == s2 [n - 1]) // Include this character in LCS and recur for remaining substrings return 1 + lcsRec (s1 , s2 , m - 1 , n - 1); else // If the last characters do not match // Recur for two cases: // 1. Exclude the last character of S1 // 2. Exclude the last character of S2 // Take the maximum of these two recursive calls return max (lcsRec (s1 , s2 , m , n - 1), lcsRec (s1 , s2 , m - 1 , n)); } int lcs (char s1 [], char s2 []){ int m = strlen (s1); int n = strlen (s2); return lcsRec (s1 , s2 , m , n); } int main () { char s1 [] = "AGGTAB" ; char s2 [] = "GXTXAYB" ; printf ("%d \n " , lcs (s1 , s2)); return 0 ; } Java // A Naive recursive implementation of LCS problem class GfG { // Returns length of LCS for s1[0..m-1], s2[0..n-1] static int lcsRec (String s1 , String s2 , int m , int n) { // Base case: If either string is empty, the length of LCS is 0 if (m == 0 || n == 0) return 0 ; // If the last characters of both substrings match if (s1 . charAt (m - 1) == s2 . charAt (n - 1)) // Include this character in LCS and recur for remaining substrings return 1 + lcsRec (s1 , s2 , m - 1 , n - 1); else // If the last characters do not match // Recur for two cases: // 1. Exclude the last character of S1 // 2. Exclude the last character of S2 // Take the maximum of these two recursive calls return Math . max (lcsRec (s1 , s2 , m , n - 1), lcsRec (s1 , s2 , m - 1 , n)); } static int lcs (String s1 , String s2){ int m = s1 . length (), n = s2 . length (); return lcsRec (s1 , s2 , m , n); } public static void main (String [] args) { String s1 = "AGGTAB" ; String s2 = "GXTXAYB" ; System . out . println (lcs (s1 , s2)); } } Python # A Naive recursive implementation of LCS problem # Returns length of LCS for s1[0..m-1], s2[0..n-1] def lcsRec (s1 , s2 , m , n): # Base case: If either string is empty, the length of LCS is 0 if m == 0 or n == 0 : return 0 # If the last characters of both substrings match if s1 [m - 1] == s2 [n - 1]: # Include this character in LCS and recur for remaining substrings return 1 + lcsRec (s1 , s2 , m - 1 , n - 1) else : # If the last characters do not match # Recur for two cases: # 1. Exclude the last character of S1 # 2. Exclude the last character of S2 # Take the maximum of these two recursive calls return max (lcsRec (s1 , s2 , m , n - 1), lcsRec (s1 , s2 , m - 1 , n)) def lcs (s1 , s2): m = len (s1) n = len (s2) return lcsRec (s1 , s2 , m , n) if __name__ == "__main__" : s1 = "AGGTAB" s2 = "GXTXAYB" print (lcs (s1 , s2)) C# // A Naive recursive implementation of LCS problem using System ; class GfG { // Returns length of LCS for s1[0..m-1], s2[0..n-1] static int lcsRec (string s1 , string s2 , int m , int n) { // Base case: If either string is empty, the length of LCS is 0 if (m == 0 || n == 0) return 0 ; // If the last characters of both substrings match if (s1 [m - 1] == s2 [n - 1]) // Include this character in LCS and recur for remaining substrings return 1 + lcsRec (s1 , s2 , m - 1 , n - 1); else // If the last characters do not match // Recur for two cases: // 1. Exclude the last character of S1 // 2. Exclude the last character of S2 // Take the maximum of these two recursive calls return Math . Max (lcsRec (s1 , s2 , m , n - 1), lcsRec (s1 , s2 , m - 1 , n)); } static int lcs (string s1 , string s2){ int m = s1 . Length , n = s2 . Length ; return lcsRec (s1 , s2 , m , n); } static void Main () { string s1 = "AGGTAB" ; string s2 = "GXTXAYB" ; Console . WriteLine (lcs (s1 , s2)); } } JavaScript // A Naive recursive implementation of LCS problem // Returns length of LCS for s1[0..m-1], s2[0..n-1] function lcsRec (s1 , s2 , m , n) { // Base case: If either string is empty, the length of LCS is 0 if (m === 0 || n === 0) return 0 ; // If the last characters of both substrings match if (s1 [m - 1] === s2 [n - 1]) // Include this character in LCS and recur for remaining substrings return 1 + lcsRec (s1 , s2 , m - 1 , n - 1); else return Math . max (lcsRec (s1 , s2 , m , n - 1), lcsRec (s1 , s2 , m - 1 , n)); } function lcs (s1 , s2){ let m = s1 . length ; let n = s2 . length ; return lcsRec (s1 , s2 , m , n); } // driver code let s1 = "AGGTAB" ; let s2 = "GXTXAYB" ; let m = s1 . length ; let n = s2 . length ; console . log (lcs (s1 , s2 , m , n)); Output 4 [Better Approach] Using Memoization (Top Down DP) - O(m * n) Time and O(m * n) Space To optimize the recursive solution, we use a 2D memoization table of size (m+1)×(n+1)(m+1) \times (n+1)(m+1)×(n+1), initialized to –1-1–1 to track computed values. Before making recursive calls, we check this table to avoid redundant computations of overlapping subproblems. This prevents repeated calculations, improving efficiency through memoization or tabulation . Overlapping Subproblems in Longest Common Subsequence C++ // C++ implementation of Top-Down DP // of LCS problem #include <bits/stdc++.h> using namespace std ; // Returns length of LCS for s1[0..m-1], s2[0..n-1] int lcsRec (string & s1 , string & s2 , int m , int n , vector < vector < int >> & memo) { // Base Case if (m == 0 || n == 0) return 0 ; // Already exists in the memo table if (memo [m][n] != -1) return memo [m][n]; // Match if (s1 [m - 1] == s2 [n - 1]) return memo [m][n] = 1 + lcsRec (s1 , s2 , m - 1 , n - 1 , memo); // Do not match return memo [m][n] = max (lcsRec (s1 , s2 , m , n - 1 , memo), lcsRec (s1 , s2 , m - 1 , n , memo)); } int lcs (string & s1 , string & s2){ int m = s1 . length (); int n = s2 . length (); vector < vector < int >> memo (m + 1 , vector < int > (n + 1 , -1)); return lcsRec (s1 , s2 , m , n , memo); } int main () { string s1 = "AGGTAB" ; string s2 = "GXTXAYB" ; cout << lcs (s1 , s2) << endl ; return 0 ; } C // C implementation of Top-Down DP // of LCS problem #include <stdio.h> #include <string.h> // Define a maximum size for the strings #define MAX 1000 // Function to find the maximum of two integers int max (int a , int b) { return (a > b) ? a : b ; } // Returns length of LCS for s1[0..m-1], s2[0..n-1] int lcsRec (const char * s1 , const char * s2 , int m , int n , int memo [MAX][MAX]) { // Base Case if (m == 0 || n == 0) { return 0 ; } // Already exists in the memo table if (memo [m][n] != -1) { return memo [m][n]; } // Match if (s1 [m - 1] == s2 [n - 1]) { return memo [m][n] = 1 + lcsRec (s1 , s2 , m - 1 , n - 1 , memo); } // Do not match return memo [m][

n] = max (lcsRec (s1 , s2 , m , n - 1 , memo), lcsRec (s1 , s2 , m - 1 , n , memo)); } int lcs (char s1 [], char s2 []){ int m = strlen (s1); int n = strlen (s2); // Create memo table with fixed size int memo [MAX][MAX]; for (int i = 0 ; i <= m ; i ++) { for (int j = 0 ; j <= n ; j ++) { // Initialize memo table with -1 memo [i][j] = -1 ; } } return lcsRec (s1 , s2 , m , n , memo); } int main () { const char * s1 = "AGGTAB" ; const char * s2 = "GXTXAYB" ; printf ("%d \n " , lcs (s1 , s2)); return 0 ; } Java // Java implementation of Top-Down DP of LCS problem import java.util.Arrays ; class GfG { // Returns length of LCS for s1[0..m-1], s2[0..n-1] static int lcsRec (String s1 , String s2 , int m , int n , int [][] memo) { // Base Case if (m == 0 || n == 0) return 0 ; // Already exists in the memo table if (memo [m][n] != - 1) return memo [m][n] ; // Match if (s1 . charAt (m - 1) == s2 . charAt (n - 1)) { return memo [m][n] = 1 + lcsRec (s1 , s2 , m - 1 , n - 1 , memo); } // Do not match return memo [m][n] = Math . max (lcsRec (s1 , s2 , m , n - 1 , memo), lcsRec (s1 , s2 , m - 1 , n , memo)); } static int lcs (String s1 , String s2){ int m = s1 . length (); int n = s2 . length (); int [][] memo = new int [m + 1][n + 1] ; // Initialize the memo table with -1 for (int i = 0 ; i <= m ; i ++) { Arrays . fill (memo [i] , - 1); } return lcsRec (s1 , s2 , m , n , memo); } public static void main (String [] args) { String s1 = "AGGTAB" ; String s2 = "GXTXAYB" ; System . out . println (lcs (s1 , s2)); } } Python def lcsRec (s1 , s2 , m , n , memo): # Base Case if m == 0 or n == 0 : return 0 # Already exists in the memo table if memo [m][n] != - 1 : return memo [m][n] # Match if s1 [m - 1] == s2 [n - 1]: memo [m][n] = 1 + lcsRec (s1 , s2 , m - 1 , n - 1 , memo) return memo [m][n] # Do not match memo [m][n] = max (lcsRec (s1 , s2 , m , n - 1 , memo), lcsRec (s1 , s2 , m - 1 , n , memo)) return memo [m][n] def lcs (s1 , s2): m = len (s1) n = len (s2) memo = [[- 1 for _ in range (n + 1)] for _ in range (m + 1)] return lcsRec (s1 , s2 , m , n , memo) if __name__ == "__main__" : s1 = "AGGTAB" s2 = "GXTXAYB" print (lcs (s1 , s2)) C# // C# implementation of Top-Down DP of LCS problem using System ; class GfG { // Returns length of LCS for s1[0..m-1], s2[0..n-1] static int lcsRec (string s1 , string s2 , int m , int n , int [,] memo) { // Base Case if (m == 0 || n == 0) return 0 ; // Already exists in the memo table if (memo [m , n] != - 1) return memo [m , n]; // Match if (s1 [m - 1] == s2 [n - 1]) { return memo [m , n] = 1 + lcsRec (s1 , s2 , m - 1 , n - 1 , memo); } // Do not match return memo [m , n] = Math . Max (lcsRec (s1 , s2 , m , n - 1 , memo), lcsRec (s1 , s2 , m - 1 , n , memo)); } static int lcs (string s1 , string s2){ int m = s1 . Length ; int n = s2 . Length ; int [,] memo = new int [m + 1 , n + 1]; // Initialize memo array with -1 for (int i = 0 ; i <= m ; i ++) { for (int j = 0 ; j <= n ; j ++) { memo [i , j] = - 1 ; } } return lcsRec (s1 , s2 , m , n , memo); } public static void Main () { string s1 = "AGGTAB" ; string s2 = "GXTXAYB" ; Console . WriteLine (lcs (s1 , s2)); } } JavaScript // A Top-Down DP implementation of LCS problem // Returns length of LCS for s1[0..m-1], s2[0..n-1] function lcsRec (s1 , s2 , m , n , memo) { // Base Case if (m === 0 || n === 0) return 0 ; // Already exists in the memo table if (memo [m][n] !== - 1) return memo [m][n]; // Match if (s1 [m - 1] === s2 [n - 1]) { memo [m][n] = 1 + lcsRec (s1 , s2 , m - 1 , n - 1 , memo); return memo [m][n]; } // Do not match memo [m][n] = Math . max (lcsRec (s1 , s2 , m , n - 1 , memo), lcsRec (s1 , s2 , m - 1 , n , memo)); return memo [m][n]; } function lcs (s1 , s2) { const m = s1 . length ; const n = s2 . length ; const memo = Array . from ({ length : m + 1 }, () => Array (n + 1). fill (- 1)); return lcsRec (s1 , s2 , m , n , memo); } // driver code const s1 = "AGGTAB" ; const s2 = "GS1TS1AS2B" ; console . log (lcs (s1 , s2)); Output 4 [Expected Approach 1] Using Bottom-Up DP (Tabulation) - O(m * n) Time and O(m * n) Space There are two parameters that change in the recursive solution and these parameters go from 0 to m and 0 to n. So we create a 2D dp array of size (m+1) x (n+1). We first fill the known entries when m is 0 or n is 0. Then we fill the remaining entries using the recursive formula. Say the strings are S1 = "AXTY" and S2 = "AYZX" , Follow below : C++ #include <iostream> #include <vector> using namespace std ; // Returns length of LCS for s1[0..m-1], s2[0..n-1] int lcs (string & s1 , string & s2) { int m = s1 . size (); int n = s2 . size (); // Initializing a matrix of size (m+1)*(n+1) vector < vector < int >> dp (m + 1 , vector < int > (n + 1 , 0)); // Building dp[m+1][n+1] in bottom-up fashion for (int i = 1 ; i <= m ; ++ i) { for (int j = 1 ; j <= n ; ++ j) { if (s1 [i - 1] == s2 [j - 1]) dp [i][j] = dp [i - 1][j - 1] + 1 ; else dp [i][j] = max (dp [i - 1][j], dp [i][j - 1]); } } // dp[m][n] contains length of LCS for s1[0..m-1] // and s2[0..n-1] return dp [m][n]; } int main () { string s1 = "AGGTAB" ; string s2 = "GXTXAYB" ; cout << lcs (s1 , s2) << endl ; return 0 ; } C #include <stdio.h> #include <stdlib.h> #include <string.h> int max (int x , int y); // Function to find length of LCS for s1[0..m-1], s2[0..n-1] int lcs (const char * S1 , const char * S2) { int m = strlen (S1); int n = strlen (S2); // Initializing a matrix of size (m+1)*(n+1) int dp [m + 1][n + 1]; // Building dp[m+1][n+1] in bottom-up fashion for (int i = 0 ; i <= m ; i ++) { for (int j = 0 ; j <= n ; j ++) { if (i == 0 || j == 0) dp [i][j] = 0 ; else if (S1 [i - 1] == S2 [j - 1]) dp [i][j] = dp [i - 1][j - 1] + 1 ; else dp [i][j] = max (dp [i - 1][j], dp [i][j - 1]); } } return dp [m][n]; } int max (int x , int y) { return (x > y) ? x : y ; } int main () { const char * S1 = "AGGTAB" ; const char * S2 = "GXTXAYB" ; printf ("%d \n " , lcs (S1 , S2)); return 0 ; } Java import java.util.Arrays ; class GfG { // Returns length of LCS for s1[0..m-1], s2[0..n-1] static int lcs (String S1 , String S2) { int m = S1 . length (); int n = S2 . length (); // Initializing a matrix of size (m+1)*(n+1) int [][] dp = new int [m + 1][n + 1] ; // Building dp[m+1][n+1] in bottom-up fashion for (int i = 1 ; i <= m ; i ++) { for (int j = 1 ; j <= n ; j ++) { if (S1 . charAt (i - 1) == S2 . charAt (j - 1)) { dp [i][j] = dp [i - 1][j - 1] + 1 ; } else { dp [i][j] = Math . max (dp [i - 1][j] , dp [i][j - 1]); } } } // dp[m][n] contains length of LCS for S1[0..m-1] // and S2[0..n-1] return dp [m][n] ; } public static void main (String [] args) { String S1 = "AGGTAB" ; String S2 = "GXTXAYB" ; System . out . println (lcs (S1 , S2)); } } Python def lcs (S1 , S2): m = len (S1) n = len (S2) # Initializing a matrix of size (m+1)*(n+1) dp = [[0] * (n + 1) for x in range (m + 1)] # Building dp[m+1][n+1] in bottom-up fashion for i in range (1 , m + 1): for j in range (1 , n + 1): if S1 [i - 1] == S2 [j - 1]: dp [i][j] = dp [i - 1][j - 1] + 1 else : dp [i][j] = max (dp [i - 1][j], dp [i][j - 1]) # dp[m][n] contains length of LCS for S1[0..m-1] # and S2[0..n-1] return dp [m][n] if __name__ == "__main__" : S1 = "AGGTAB" S2 = "GXTXAYB" print (lcs (S1 , S2)) C# using System ; class GfG { // Returns length of LCS for S1[0..m-1], S2[0..n-1] static int lcs (string S1 , string S2) { int m = S1 . Length ; int n = S2 . Length ; // Initializing a matrix of size (m+1)*(n+1) int [,] dp = new int [m + 1 , n + 1]; // Building dp[m+1][n+1] in bottom-up fashion for (int i = 1 ; i <= m ; i ++) { for (int j = 1 ; j <= n ; j ++) { if (S1 [i - 1] == S2 [j - 1]) { dp [i , j] = dp [i - 1 , j - 1] + 1 ; } else { dp [i , j] = Math . Max (dp [i - 1 , j], dp [i , j - 1]); } } } // dp[m, n] contains length of LCS for S1[0..m-1] // and S2[0..n-1] return dp [m , n]; } static void Main () { string S1 = "AGGTAB" ; string S2 = "GXTXAYB" ; Console . WriteLine (lcs (S1 , S2)); } } JavaScript function lcs (S1 , S2) { const m = S1 . length ; const n = S2 . length ; // Initializing a matrix of size (m+1)*(n+1) const dp = Array . from ({ length : m + 1 }, () => Array (n + 1). fill (0)); // Building dp[m+1][n+1] in bottom-up fashion for (let i = 1 ; i <= m ; i ++) { for (let j = 1 ; j <= n ; j ++) { if (S1 [i - 1] === S2 [j - 1]) { dp [i][j] = dp [i - 1][j - 1] + 1 ; } else { dp [i][j] = Math . max (dp [i - 1][j], dp [i][j - 1]); } } } // dp[m][n] contains length of LCS for // S1[0..m-1] and S2[0..n-1] return dp [m][n]; } const S1 = "AGGTAB" ; const S2 = "GXTXAYB" ; console . log (lcs (S1 , S2)); Output 4 [Expected Approach 2] Using Bottom-Up DP (Space-Optimization): One important observation in the above simple implementation is, in each iteration of the outer loop we only need values from all columns of the previous row. So there is no need to store all rows in our DP matrix, we can just store two rows at a time and use them. We can further optimize to use only one array. Please refer this post: A Space Optimized Solution of LCS Applications of LCS LCS is used to implement diff utility (find the difference between two data sources). It is also widely used by revision control systems such as Git for multiple changes made to a revision-controlled collection of files. Problems based on LCS LCS of 3 Strings Printing LCS Longest Palindromic Subsequence Shortest Common Supersequence Minimum Insertions and Deletions Edit Distance Minimum Insertions for Palindrome Longest Common Substring Longest Palindromic Substring Longest Repeated Subsequence Count Distinct Subsequences Regular Expression Matching Related Links: Print LCS of two Strings Dynamic Programming LCS and Edit Distance Dynamic Programming Practice Problems Comment Article Tags: Article Tags: Strings Dynamic Programming DSA Amazon Hike FactSet subsequence LCS strings + 5 More