Edit Distance - GeeksforGeeks

Source: https://www.geeksforgeeks.org/edit-distance-dp-5/

Edit Distance Last Updated : 23 Jul, 2025 Given two strings s1 and s2 and below operations that can be performed on s1 . The task is to find the minimum number of edits (operations) to convert ' s1 ' into ' s2 '. Insert : Insert any character before or after any index of s1 Remove: Remove a character of s1 Replace: Replace a character at any index of s1 with some other character. Note: All of the above operations are of equal cost. Examples: Input: s1 = "geek", s2 = "gesek" Output: 1 Explanation: We can convert s1 into s2 by inserting an 's' between two consecutive 'e' in s1. Input: s1 = "gfg", s2 = "gfg" Output: 0 Explanation: Both strings are same. Input: s1 = "abcd", s2 = "bcfe" Output: 3 Explanation: We can convert s1 into s2 by removing 'a', replacing 'd' with 'f' and inserting 'e' at the end. Try it on GfG Practice Illustration of Edit Distance: Let's suppose we have s1="GEEXSFRGEEKKS" and s2="GEEKSFORGEEKS" Now to convert s1 into s2 we would require 3 minimum operations: Operation 1: Replace ' X ' to ' K ' Operation 2: Insert ' O ' between ' F ' and ' R ' Operation 3: Remove second last character i.e. ' K ' Refer the below image for better understanding. [Naive Approach] Using Recursion-O(3^(m+n)) time and space O(m*n) The idea is to process all characters one by one starting from either from left or right sides of both strings. Let us process from the right end of the strings, there are two possibilities for every pair of characters being traversed, either they match or they don't match . If last characters of both string matches then we simply recursively calculate the answer for rest of part of the strings. When last characters do not match, we perform all three operations to match the last characters, i.e. insert, replace, and remove. And then recursively calculate the result for the remaining part of the string. Upon completion of these operations, we will select the minimum answer and add 1 to it. Below is the recursive tree for this problem considering the case when the last characters never match. When the last characters of strings matches. Make a recursive call editDistance(m-1, n-1) to calculate the answer for remaining part of the strings. When the last characters of strings don't matches. Make three recursive calls as show below: Insert s2[n-1] at last of s1 : editDistance(m, n-1) Replace s1[m-1] with s2[n-1] : editDistance(m-1, n-1) Remove s1[m-1] : editDistance(m-1, n) Recurrence Relations for Edit Distance: Case 1 : When the last character of both the strings are same. editDistance(s1, s2, m, n) = editDistance(s1, s2, m-1, n-1) Case 2 : When the last characters are different editDistance(s1, s2, m, n) = 1 + Minimum{ editDistance(s1, s2 ,m,n-1), editDistance(s1, s2 ,m-1, n), editDistance(s1, s2 , m-1, n-1)} Base Case for Edit Distance : Case 1 : When s1 becomes empty i.e. m =0 return n , as it require n insertions to convert an empty string to s2 of size n Case 2 : When s2 becomes empty i.e. n =0 return m , as it require m removals to convert s1 of size m to an empty string. C++ // A Naive recursive C++ program to find minimum number // of operations to convert s1 to s2 #include <iostream> #include <string> #include <algorithm> using namespace std ; // Recursive function to find number of operations // needed to convert s1 into s2. int editDistRec (string & s1 , string & s2 , int m , int n) { // If first string is empty, the only option is to // insert all characters of second string into first if (m == 0) return n ; // If second string is empty, the only option is to // remove all characters of first string if (n == 0) return m ; // If last characters of two strings are same, nothing // much to do. Get the count for // remaining strings. if (s1 [m - 1] == s2 [n - 1]) return editDistRec (s1 , s2 , m - 1 , n - 1); // If last characters are not same, consider all three // operations on last character of first string, // recursively compute minimum cost for all three // operations and take minimum of three values. return 1 + min ({ editDistRec (s1 , s2 , m , n - 1), editDistRec (s1 , s2 , m - 1 , n), editDistRec (s1 , s2 , m - 1 , n - 1)}); } // Wrapper function to initiate the recursive calculation int editDistance (string

& s1 , string & s2) { return editDistRec (s1 , s2 , s1 . length (), s2 . length ()); } int main () { string s1 = "abcd" ; string s2 = "bcfe" ; cout << editDistance (s1 , s2); return 0 ; } C #include <stdio.h> #include <string.h> // Function to find minimum of three numbers int min (int x , int y , int z) { return (x < y) ? (x < z ? x : z) : (y < z ? y : z); } // A Naive recursive C program to find minimum number // of operations to convert s1 to s2 int editDistRec (char * s1 , char * s2 , int m , int n) { // If first string is empty, the only option is to // insert all characters of second string into first if (m == 0) return n ; // If second string is empty, the only option is to // remove all characters of first string if (n == 0) return m ; // If last characters of two strings are same, nothing // much to do. Get the count for // remaining strings. if (s1 [m - 1] == s2 [n - 1]) return editDistRec (s1 , s2 , m - 1 , n - 1); // If last characters are not same, consider all three // operations on last character of first string, // recursively compute minimum cost for all three // operations and take minimum of three values. return 1 + min (editDistRec (s1 , s2 , m , n - 1), // Insert editDistRec (s1 , s2 , m - 1 , n), // Remove editDistRec (s1 , s2 , m - 1 , n - 1) // Replace); } int main () { char s1 [] = "abcd" ; char s2 [] = "bcfe" ; printf ("%d \n " , editDistRec (s1 , s2 , strlen (s1), strlen (s2))); return 0 ; } Java // A Naive recursive Java program to find minimum number // of operations to convert s1 to s2. class GfG { // Recursive function to find number of operations // needed to convert s1 into s2. static int editDistRec (String s1 , String s2 , int m , int n) { // If first string is empty, the only option is to // insert all characters of second string into first if (m == 0) return n ; // If second string is empty, the only option is to // remove all characters of first string if (n == 0) return m ; // If last characters of two strings are same, nothing // much to do. Get the count for // remaining strings. if (s1 . charAt (m - 1) == s2 . charAt (n - 1)) return editDistRec (s1 , s2 , m - 1 , n - 1); // If last characters are not same, consider all three // operations on last character of first string, // recursively compute minimum cost for all three // operations and take minimum of three values. return 1 + Math . min (Math . min (editDistRec (s1 , s2 , m , n - 1), editDistRec (s1 , s2 , m - 1 , n)), editDistRec (s1 , s2 , m - 1 , n - 1)); } // Wrapper function to initiate the recursive calculation static int editDistance (String s1 , String s2) { return editDistRec (s1 , s2 , s1 . length (), s2 . length ()); } public static void main (String [] args) { String s1 = "abcd" ; String s2 = "bcfe" ; System . out . println (editDistance (s1 , s2)); } } Python # A Naive recursive Python program to find minimum number # of operations to convert s1 to s2. # Recursive function to find number of operations # needed to convert s1 into s2. def editDistRec (s1 , s2 , m , n): # If first string is empty, the only option is to # insert all characters of second string into first if m == 0 : return n # If second string is empty, the only option is to # remove all characters of first string if n == 0 : return m # If last characters of two strings are same, nothing # much to do. Get the count for # remaining strings. if s1 [m - 1] == s2 [n - 1]: return editDistRec (s1 , s2 , m - 1 , n - 1) # If last characters are not same, consider all three # operations on last character of first string, # recursively compute minimum cost for all three # operations and take minimum of three values. return 1 + min (editDistRec (s1 , s2 , m , n - 1), editDistRec (s1 , s2 , m - 1 , n), editDistRec (s1 , s2 , m - 1 , n - 1)) # Wrapper function to initiate # the recursive calculation def editDistance (s1 , s2): return editDistRec (s1 , s2 , len (s1), len (s2)) if __name__ == "__main__" : s1 = "abcd" s2 = "bcfe" print (editDistance (s1 , s2)) C# // A Naive recursive C# program to find minimum number // of operations to convert s1 to s2. using System ; class GfG { // Recursive function to find number of operations // needed to convert s1 into s2. static int editDistRec (string s1 , string s2 , int m , int n) { // If first string is empty, the only option is to // insert all characters of second string into first if (m == 0) return n ; // If second string is empty, the only option is to // remove all characters of first string if (n == 0) return m ; // If last characters of two strings are same, nothing // much to do. Get the count for // remaining strings. if (s1 [m - 1] == s2 [n - 1]) return editDistRec (s1 , s2 , m - 1 , n - 1); // If last characters are not same, consider all three // operations on last character of first string, // recursively compute minimum cost for all three // operations and take minimum of three values. return 1 + Math . Min (Math . Min (editDistRec (s1 , s2 , m , n - 1), editDistRec (s1 , s2 , m - 1 , n)), editDistRec (s1 , s2 , m - 1 , n - 1)); } // Wrapper function to initiate the recursive calculation static int editDistance (string s1 , string s2) { return editDistRec (s1 , s2 , s1 . Length , s2 . Length); } static void Main (string [] args) { string s1 = "abcd" ; string s2 = "bcfe" ; Console . WriteLine (editDistance (s1 , s2)); } } JavaScript // A Naive recursive JavaScript program to find minimum number // of operations to convert s1 to s2. // Recursive function to find number of operations // needed to convert s1 into s2. function editDistRec (s1 , s2 , m , n) { // If first string is empty, the only option is to // insert all characters of second string into first if (m === 0) return n ; // If second string is empty, the only option is to // remove all characters of first string if (n === 0) return m ; // If last characters of two strings are same, nothing // much to do. Get the count for // remaining strings. if (s1 [m - 1] === s2 [n - 1]) return editDistRec (s1 , s2 , m - 1 , n - 1); // If last characters are not same, consider all three // operations on last character of first string, //

recursively compute minimum cost for all three // operations and take minimum of three values. return 1 + Math . min (editDistRec (s1 , s2 , m , n - 1), editDistRec (s1 , s2 , m - 1 , n), editDistRec (s1 , s2 , m - 1 , n - 1)); } // Wrapper function to initiate the recursive calculation function editDistance (s1 , s2) { return editDistRec (s1 , s2 , s1 . length , s2 . length); } // Driver Code const s1 = "abcd" ; const s2 = "bcfe" ; console . log (editDistance (s1 , s2)); Output 3 [Better Approach 1] Using Top-Down DP (Memoization) - O(m*n) time and O(m*n) space In the above recursive approach, there are several overlapping subproblems: editDist(m-1, n-1) is called Three times editDist(m-1, n-2) is called Two times editDist(m-2, n-1) is called Two times. And so on... So, we can use Memoization to store the result of each subproblems to avoid recalculating the result again and again. Below is the illustration of overlapping subproblems during the recursion. C++ // C++ program to find minimum number // of operations to convert s1 to s2 #include <iostream> #include <algorithm> #include <vector> using namespace std ; // Recursive function to find number of operations // needed to convert s1 into s2. int editDistRec (string & s1 , string & s2 , int m , int n , vector < vector < int >> & memo) { // If first string is empty, the only option is to // insert all characters of second string into first if (m == 0) return n ; // If second string is empty, the only option is to // remove all characters of first string if (n == 0) return m ; // If value is memoized if (memo [m][n] != -1) return memo [m][n]; // If last characters of two strings are same, nothing // much to do. Get the count for // remaining strings. if (s1 [m - 1] == s2 [n - 1]) return memo [m][n] = editDistRec (s1 , s2 , m - 1 , n - 1 , memo); // If last characters are not same, consider all three // operations on last character of first string, // recursively compute minimum cost for all three // operations and take minimum of three values. return memo [m][n] = 1 + min ({ editDistRec (s1 , s2 , m , n - 1 , memo), editDistRec (s1 , s2 , m - 1 , n , memo), editDistRec (s1 , s2 , m - 1 , n - 1 , memo)}); } // Wrapper function to initiate the recursive calculation int editDistance (string & s1 , string & s2) { int m = s1 . length (), n = s2 . length (); vector < vector < int >> memo (m + 1 , vector < int > (n + 1 , -1)); return editDistRec (s1 , s2 , m , n , memo); } int main () { string s1 = "abcd" ; string s2 = "bcfe" ; cout << editDistance (s1 , s2); return 0 ; } Java // Java program to find minimum number // of operations to convert s1 to s2 import java.util.Arrays ; class GfG { // Recursive function to find number of operations // needed to convert s1 into s2. static int editDistRec (String s1 , String s2 , int m , int n , int [][] memo) { // If first string is empty, the only option is to // insert all characters of second string into first if (m == 0) return n ; // If second string is empty, the only option is to // remove all characters of first string if (n == 0) return m ; // If value is memoized if (memo [m][n] != - 1) return memo [m][n] ; // If last characters of two strings are same, nothing // much to do. Get the count for // remaining strings. if (s1 . charAt (m - 1) == s2 . charAt (n - 1)) return memo [m][n] = editDistRec (s1 , s2 , m - 1 , n - 1 , memo); // If last characters are not same, consider all three // operations on last character of first string, // recursively compute minimum cost for all three // operations and take minimum of three values. return memo [m][n] = 1 + Math . min (editDistRec (s1 , s2 , m , n - 1 , memo), Math . min (editDistRec (s1 , s2 , m - 1 , n , memo), editDistRec (s1 , s2 , m - 1 , n - 1 , memo))); } // Wrapper function to initiate the recursive calculation static int editDistance (String s1 , String s2) { int m = s1 . length (), n = s2 . length (); int [][] memo = new int [m + 1][n + 1] ; for (int [] row : memo) Arrays . fill (row , - 1); return editDistRec (s1 , s2 , m , n , memo); } public static void main (String [] args) { String s1 = "abcd" ; String s2 = "bcfe" ; System . out . println (editDistance (s1 , s2)); } } Python # Python program to find minimum number # of operations to convert s1 to s2 # Recursive function to find number of operations # needed to convert s1 into s2. def editDistRec (s1 , s2 , m , n , memo): # If first string is empty, the only option is to # insert all characters of second string into first if m == 0 : return n # If second string is empty, the only option is to # remove all characters of first string if n == 0 : return m # If value is memoized if memo [m][n] != - 1 : return memo [m][n] # If last characters of two strings are same, nothing # much to do. Get the count for # remaining strings. if s1 [m - 1] == s2 [n - 1]: memo [m][n] = editDistRec (s1 , s2 , m - 1 , n - 1 , memo) return memo [m][n] # If last characters are not same, consider all three # operations on last character of first string, # recursively compute minimum cost for all three # operations and take minimum of three values. memo [m][n] = 1 + min (editDistRec (s1 , s2 , m , n - 1 , memo), editDistRec (s1 , s2 , m - 1 , n , memo), editDistRec (s1 , s2 , m - 1 , n - 1 , memo)) return memo [m][n] # Wrapper function to initiate the recursive calculation def editDistance (s1 , s2): m , n = len (s1), len (s2) memo = [[- 1 for _ in range (n + 1)] for _ in range (m + 1)] return editDistRec (s1 , s2 , m , n , memo) if __name__ == "__main__" : s1 = "abcd" s2 = "bcfe" print (editDistance (s1 , s2)) C# // C# program to find minimum number // of operations to convert s1 to s2 using System ; class GfG { // Recursive function to find number of operations // needed to convert s1 into s2. static int editDistRec (string s1 , string s2 , int m , int n , int [,] memo) { // If first string is empty, the only option is to // insert all characters of second string into first if (m == 0) return n

; // If second string is empty, the only option is to // remove all characters of first string if (n == 0) return m ; // If value is memoized if (memo [m , n] != - 1) return memo [m , n]; // If last characters of two strings are same, nothing // much to do. Get the count for // remaining strings. if (s1 [m - 1] == s2 [n - 1]) return memo [m , n] = editDistRec (s1 , s2 , m - 1 , n - 1 , memo); // If last characters are not same, consider all three // operations on last character of first string, // recursively compute minimum cost for all three // operations and take minimum of three values. return memo [m , n] = 1 + Math . Min (editDistRec (s1 , s2 , m , n - 1 , memo), Math . Min (editDistRec (s1 , s2 , m - 1 , n , memo), editDistRec (s1 , s2 , m - 1 , n - 1 , memo))); } // Wrapper function to initiate the recursive calculation static int editDistance (string s1 , string s2) { int m = s1 . Length , n = s2 . Length ; int [,] memo = new int [m + 1 , n + 1]; for (int i = 0 ; i <= m ; i ++) for (int j = 0 ; j <= n ; j ++) memo [i , j] = - 1 ; return editDistRec (s1 , s2 , m , n , memo); } static void Main (string [] args) { string s1 = "abcd" ; string s2 = "bcfe" ; Console . WriteLine (editDistance (s1 , s2)); } } JavaScript // JavaScript program to find minimum number // of operations to convert s1 to s2 // Recursive function to find number of operations // needed to convert s1 into s2. function editDistRec (s1 , s2 , m , n , memo) { // If first string is empty, the only option is to // insert all characters of second string into first if (m === 0) return n ; // If second string is empty, the only option is to // remove all characters of first string if (n === 0) return m ; // If value is memoized if (memo [m][n] !== - 1) return memo [m][n]; // If last characters of two strings are same, nothing // much to do. Get the count for // remaining strings. if (s1 [m - 1] === s2 [n - 1]) return memo [m][n] = editDistRec (s1 , s2 , m - 1 , n - 1 , memo); // If last characters are not same, consider all three // operations on last character of first string, // recursively compute minimum cost for all three // operations and take minimum of three values. return memo [m][n] = 1 + Math . min (editDistRec (s1 , s2 , m , n - 1 , memo), Math . min (editDistRec (s1 , s2 , m - 1 , n , memo), editDistRec (s1 , s2 , m - 1 , n - 1 , memo))); } // Wrapper function to initiate the recursive calculation function editDistance (s1 , s2) { const m = s1 . length , n = s2 . length ; const memo = Array . from ({ length : m + 1 }, () => Array (n + 1). fill (- 1)); return editDistRec (s1 , s2 , m , n , memo); } // Driver Code const s1 = "abcd" ; const s2 = "bcfe" ; console . log (editDistance (s1 , s2)); Output 3 [Better Approach 2] Using Bottom-Up DP (Tabulation)-O(m*n) time and O(m*n) space Use a table to store solutions of subproblems to avoiding recalculate the same subproblems multiple times. By doing this, if same subproblems repeated during, we retrieve the solutions from the table itself. Below are the steps to convert the recursive approach to Bottom up approach: 1. Choosing Dimensions of Table: The state of smaller sub-problems depends on the input parameters m and n because at least one of them will decrease after each recursive call. So we need to construct a 2D table dp[][] to store the solution of the sub-problems. 2. Choosing Correct size of Table: The range of parameters goes from 0 to m and 0 to n. So we choose dimensions as (m + 1)*(n + 1) 3. Filling the table: It consist of two stages, table initialisation and building the solution from the smaller subproblems: Table initialisation: Before building the solution, we need to initialise the table with the known values i.e. base case. Here m = 0 and n = 0 is the situation of the base case, so we initialise first-column dp[i][0] with i and first-row dp[0][j] with j . Building the solution of larger problems from the smaller subproblems: We can easily define the iterative structure by using the recursive structure of the above recursive solution. if (s1[i - 1] == s2[j - 1]) dp[i][j] = dp[i - 1][j - 1]; if (s1[i - 1] != s2[j - 1]) dp[i][j] = 1 + min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1]); 4. Returning final solution: After filling the table iteratively, our final solution gets stored at the bottom right corner of the 2-D table i.e. we return dp[m][n] as an output. Illustration: C++ // C++ program to find minimum number // of operations to convert s1 to s2 #include <iostream> #include <vector> #include <algorithm> using namespace std ; int editDistance (string & s1 , string & s2) { int m = s1 . length (); int n = s2 . length (); // Create a table to store results of subproblems vector < vector < int >> dp (m + 1 , vector < int > (n + 1)); // Fill the known entries in dp[][] // If one string is empty, then answer // is length of the other string for (int i = 0 ; i <= m ; i ++) dp [i][0] = i ; for (int j = 0 ; j <= n ; j ++) dp [0][j] = j ; // Fill the rest of dp[][] 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]; else dp [i][j] = 1 + min ({ dp [i][j - 1], dp [i - 1][j], dp [i - 1][j - 1]}); } } return dp [m][n]; } int main () { string s1 = "abcd" ; string s2 = "bcfe" ; cout << editDistance (s1 , s2); return 0 ; } Java // Java program to find minimum number // of operations to convert s1 to s2 import java.util.* ; class GfG { // Function to find the minimum number // of operations to convert s1 to s2 static int editDistance (String s1 , String s2) { int m = s1 . length (); int n = s2 . length (); // Create a table to store results of subproblems int [][] dp = new int [m + 1][n + 1] ; // Fill the known entries in dp[][] // If one string is empty, then answer // is length of the other string for (int i = 0 ; i <= m ; i ++) dp [i][0] = i ; for (int j = 0 ; j <= n ; j ++) dp [0][j] = j ; // Fill the rest of dp[][] 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] ; else dp [i][j

] = 1 + Math . min (Math . min (dp [i][j - 1] , dp [i - 1][j]), dp [i - 1][j - 1]); } } return dp [m][n] ; } public static void main (String [] args) { String s1 = "abcd" ; String s2 = "bcfe" ; System . out . println (editDistance (s1 , s2)); } } Python # Python program to find minimum number # of operations to convert s1 to s2 # Function to find the minimum number # of operations to convert s1 to s2 def editDistance (s1 , s2): m = len (s1) n = len (s2) # Create a table to store results of subproblems dp = [[0] * (n + 1) for _ in range (m + 1)] # Fill the known entries in dp[][] # If one string is empty, then answer # is length of the other string for i in range (m + 1): dp [i][0] = i for j in range (n + 1): dp [0][j] = j # Fill the rest of dp[][] 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] else : dp [i][j] = 1 + min (dp [i][j - 1], dp [i - 1][j], dp [i - 1][j - 1]) return dp [m][n] if __name__ == "__main__" : s1 = "abcd" s2 = "bcfe" print (editDistance (s1 , s2)) C# // C# program to find minimum number // of operations to convert s1 to s2 using System ; class GfG { // Function to find the minimum number // of operations to convert s1 to s2 static int editDistance (string s1 , string s2) { int m = s1 . Length ; int n = s2 . Length ; // Create a table to store results of subproblems int [,] dp = new int [m + 1 , n + 1]; // Fill the known entries in dp[][] // If one string is empty, then answer // is length of the other string for (int i = 0 ; i <= m ; i ++) dp [i , 0] = i ; for (int j = 0 ; j <= n ; j ++) dp [0 , j] = j ; // Fill the rest of dp[][] 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]; else dp [i , j] = 1 + Math . Min (Math . Min (dp [i , j - 1], dp [i - 1 , j]), dp [i - 1 , j - 1]); } } return dp [m , n]; } static void Main (string [] args) { string s1 = "abcd" ; string s2 = "bcfe" ; Console . WriteLine (editDistance (s1 , s2)); } } JavaScript // JavaScript program to find minimum number // of operations to convert s1 to s2 // Function to find the minimum number // of operations to convert s1 to s2 function editDistance (s1 , s2) { let m = s1 . length ; let n = s2 . length ; // Create a table to store results of subproblems let dp = Array . from ({ length : m + 1 }, () => Array (n + 1). fill (0)); // Fill the known entries in dp[][] // If one string is empty, then answer // is length of the other string for (let i = 0 ; i <= m ; i ++) dp [i][0] = i ; for (let j = 0 ; j <= n ; j ++) dp [0][j] = j ; // Fill the rest of dp[][] 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]; else dp [i][j] = 1 + Math . min (dp [i][j - 1], dp [i - 1][j], dp [i - 1][j - 1]); } } return dp [m][n]; } // Driver Code let s1 = "abcd" ; let s2 = "bcfe" ; console . log (editDistance (s1 , s2)); Output 3 [Expected Approach 1] Using Space Optimised DP-O(m*n) time and space O(n) To fill a row in DP array we require only one row i.e. the upper row. For example, if we are filling the row where i=10 in DP array then we require only values of 9th row. So we create two one dimensional arrays, prev[] and curr[]. The array prev[] stores values of row i-1 and the array curr[] stores values of the current row i. This approach reduces the space complexity from O(m*n) to O(n) C++ // C++ program to find minimum number // of operations to convert s1 to s2 #include <bits/stdc++.h> using namespace std ; int editDistance (string & s1 , string & s2) { int m = s1 . size (); int n = s2 . size (); // prev stores results for (i-1) th row // and curr for i-th row vector < int > prev (n + 1 , 0), curr (n + 1 , 0); // For 0-th row for (int j = 0 ; j <= n ; j ++) prev [j] = j ; // Rest of the rows for (int i = 1 ; i <= m ; i ++) { curr [0] = i ; // j = 0 for (int j = 1 ; j <= n ; j ++) { if (s1 [i - 1] == s2 [j - 1]) curr [j] = prev [j - 1]; else curr [j] = 1 + min ({ curr [j - 1], prev [j], prev [j - 1]}); } prev = curr ; } return prev [n]; } int main () { string s1 = "abcd" ; string s2 = "bcfe" ; cout << editDistance (s1 , s2); return 0 ; } Java // Java program to find minimum number // of operations to convert s1 to s2 import java.util.* ; class GfG { // Function to find the minimum number // of operations to convert s1 to s2 static int editDistance (String s1 , String s2) { int m = s1 . length (); int n = s2 . length (); // prev stores results for (i-1) th row // and curr for i-th row int [] prev = new int [n + 1] ; int [] curr = new int [n + 1] ; // For 0-th row for (int j = 0 ; j <= n ; j ++) prev [j] = j ; // Rest of the rows for (int i = 1 ; i <= m ; i ++) { curr [0] = i ; // j = 0 for (int j = 1 ; j <= n ; j ++) { if (s1 . charAt (i - 1) == s2 . charAt (j - 1)) curr [j] = prev [j - 1] ; else curr [j] = 1 + Math . min (Math . min (curr [j - 1] , prev [j]), prev [j - 1]); } prev = curr . clone (); } return prev [n] ; } public static void main (String [] args) { String s1 = "abcd" ; String s2 = "bcfe" ; System . out . println (editDistance (s1 , s2)); } } Python # Python program to find minimum number # of operations to convert s1 to s2 # Function to find the minimum number # of operations to convert s1 to s2 def editDistance (s1 , s2): m = len (s1) n = len (s2) # prev stores results for (i-1) th row # and curr for i-th row prev = [0] * (n + 1) curr = [0] * (n + 1) # For 0-th row for j in range (n + 1): prev [j] = j # Rest of the rows for i in range (1 , m + 1): curr [0] = i # j = 0 for j in range (1 , n + 1): if s1 [i - 1] == s2 [j - 1]: curr [j] = prev [j - 1] else : curr [j] = 1 + min (curr [j - 1], prev [j], prev [j - 1]) prev = curr [:] return prev [n] if __name__ == "__main__" : s1 = "abcd" s2 = "bcfe" print (editDistance (s1 , s2)) C# // C# program to find minimum number // of operations to convert s1 to s2 using System ; class GfG { // Function to find the minimum number // of operations to convert s1 to s2 static int editDistance (string s1 , string s2) { int m = s1 . Length ; int n = s2 . Length ; // prev stores results for (i-1) th row // and curr for i-th row int [] prev = new int [n + 1]; int [] curr = new int [n + 1]; // For 0-th row for (int j = 0 ; j <= n ; j ++) prev [j] = j ; // Rest of the rows for (int i = 1 ; i <= m ; i ++) { curr [0] = i ; // j = 0 for (int j = 1 ; j <= n ; j ++) { if (s1 [i - 1] == s2 [j - 1]) curr [j] = prev [j - 1]; else curr [j] = 1 + Math . Min (Math . Min (curr [j - 1], prev [j]), prev [j - 1]); } Array . Copy (curr , prev , n + 1); } return prev [n]; } static void Main (string [] args) { string s1 = "abcd" ; string s2 = "bcfe" ; Console . WriteLine (editDistance (s1 , s2)); } } JavaScript // JavaScript program to find minimum number // of operations to convert s1 to s2 // Function to find the minimum number // of operations to convert s1 to s2 function editDistance (s1 , s2) { let m = s1 . length ; let n = s2 . length ; // prev stores results for (i-1) th row // and curr for i-th row let prev = Array (n + 1). fill (0); let curr = Array (n + 1). fill (0); // For 0-th row for (let j = 0 ; j <= n ; j ++) prev [j] = j ; // Rest of the rows for (let i = 1 ; i <= m ; i ++) { curr [0] = i ; // j = 0 for (let j = 1 ; j <= n ; j ++) { if (s1 [i - 1] === s2 [j - 1]) curr [j] = prev [j - 1]; else curr [j] = 1 + Math . min (curr [j - 1], prev [j], prev [j - 1]); } prev = [... curr]; } return prev [n]; } // Driver Code let s1 = "abcd" ; let s2 = "bcfe" ; console . log (editDistance (s1 , s2)); Output 3 [Expected Approach 2] Using Space Optimised DP – O(m*n) Time and O(n) Space In the previous approach The curr[] array is updated using 3 values only : Value 1: curr[j] = prev[j-1] when s1[i-1] is equal to s2[j-1] Value 2: curr[j] = prev[j] when s1[i-1] is not equal to s2[j-1] Value 3: curr[j] = curr[j-1] when s1[i-1] is not equal to s2[j-1] By keeping the track of these three values we can achiever our task using only a single 1-D array C++ // C++ program to find minimum number // of operations to convert s1 to s2 #include <bits/stdc++.h> using namespace std ; int editDistance (string & s1 , string & s2) { int m = s1 . size (); int n = s2 . size (); // Stores dp[i-1][j-1] int prev ; vector < int > curr (n + 1 , 0); for (int j = 0 ; j <= n ; j ++) curr [j] = j ; for (int i = 1 ; i <= m ; i ++) { prev = curr [0]; curr [0] = i ; for (int j = 1 ; j <= n ; j ++) { int temp = curr [j]; if (s1 [i - 1] == s2 [j - 1]) curr [j] = prev ; else curr [j] = 1 + min ({ curr [j - 1], prev , curr [j]}); prev = temp ; } } return curr [n]; } int main () { string s1 = "abcd" ; string s2 = "bcfe" ; cout << editDistance (s1 , s2); return 0 ; } Java // Java program to find minimum number // of operations to convert s1 to s2 import java.util.* ; class GfG { // Function to find the minimum number // of operations to convert s1 to s2 static int editDistance (String s1 , String s2) { int m = s1 . length (); int n = s2 . length (); // Stores dp[i-1][j-1] int prev ; int [] curr = new int [n + 1] ; for (int j = 0 ; j <= n ; j ++) curr [j] = j ; for (int i = 1 ; i <= m ; i ++) { prev = curr [0] ; curr [0] = i ; for (int j = 1 ; j <= n ; j ++) { int temp = curr [j] ; if (s1 . charAt (i - 1) == s2 . charAt (j - 1)) curr [j] = prev ; else curr [j] = 1 + Math . min (Math . min (curr [j - 1] , prev), curr [j]); prev = temp ; } } return curr [n] ; } public static void main (String [] args) { String s1 = "abcd" ; String s2 = "bcfe" ; System . out . println (editDistance (s1 , s2)); } } Python # Python program to find minimum number # of operations to convert s1 to s2 # Function to find the minimum number # of operations to convert s1 to s2 def editDistance (s1 , s2): m = len (s1) n = len (s2) # Stores dp[i-1][j-1] prev = 0 curr = [0] * (n + 1) for j in range (n + 1): curr [j] = j for i in range (1 , m + 1): prev = curr [0] curr [0] = i for j in range (1 , n + 1): temp = curr [j] if s1 [i - 1] == s2 [j - 1]: curr [j] = prev else : curr [j] = 1 + min (curr [j - 1], prev , curr [j]) prev = temp return curr [n] if __name__ == "__main__" : s1 = "abcd" s2 = "bcfe" print (editDistance (s1 , s2)) C# // C# program to find minimum number // of operations to convert s1 to s2 using System ; class GfG { // Function to find the minimum number // of operations to convert s1 to s2 static int editDistance (string s1 , string s2) { int m = s1 . Length ; int n = s2 . Length ; // Stores dp[i-1][j-1] int prev ; int [] curr = new int [n + 1]; for (int j = 0 ; j <= n ; j ++) curr [j] = j ; for (int i = 1 ; i <= m ; i ++) { prev = curr [0]; curr [0] = i ; for (int j = 1 ; j <= n ; j ++) { int temp = curr [j]; if (s1 [i - 1] == s2 [j - 1]) curr [j] = prev ; else curr [j] = 1 + Math . Min (Math . Min (curr [j - 1], prev), curr [j]); prev = temp ; } } return curr [n]; } static void Main (string [] args) { string s1 = "abcd" ; string s2 = "bcfe" ; Console . WriteLine (editDistance (s1 , s2)); } } JavaScript // JavaScript program to find minimum number // of operations to convert s1 to s2 // Function to find the minimum number // of operations to convert s1 to s2 function editDistance (s1 , s2) { let m = s1 . length ; let n = s2 . length ; // Stores dp[i-1][j-1] let prev = 0 ; let curr = new Array (n + 1). fill (0); for (let j = 0 ; j <= n ; j ++) curr [j] = j ; for (let i = 1 ; i <= m ; i ++) { prev = curr [0]; curr [0] = i ; for (let j = 1 ; j <= n ; j ++) { let temp = curr [j]; if (s1 [i - 1] === s2 [j - 1]) curr [j] = prev ; else curr [j] = 1 + Math . min (curr [j - 1], prev , curr [j]); prev = temp ; } } return curr [n]; } // Driver Code let s1 = "abcd" ; let s2 = "bcfe" ; console . log (editDistance (s1 , s2)); Output 3 Real-World Applications of Edit Distance Spell Checking and Auto-Correction DNA Sequence Alignment Plagiarism Detection Natural Language Processing Version Control Systems String Matching Edit Distance and Longest Common Subsequence If we do not consider the replace operation, then edit distance problem is same as the Longest Common Subsequence (LCS) problem. With only insert and delete operations allowed, the edit distance between two strings is (M + N - 2* LCS) Interview Question based on Edit Distance You may refer edit distance based articles to know interview based questions based on edit distance. Related Articles Longest

Common Subsequence Comment Article Tags: Article Tags: Dynamic Programming DSA Amazon edit-distance