Word Ladder - Shortest Chain To Reach Target Word - GeeksforGeeks

Source: https://www.geeksforgeeks.org/word-ladder-length-of-shortest-chain-to-reach-a-target-word/

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 Word Ladder - Shortest Chain To Reach Target Word Last Updated : 23 Jul, 2025 Given an array of strings arr[] , and two different strings start and target , representing two words. The task is to find the length of the smallest chain from string start to target , such that only one character of the adjacent words differs and each word exists in arr[]. Note: Print 0 if it is not possible to form a chain. Each word in array arr[] is of same size m and contains only lowercase English alphabets. Examples: Input: start = "toon", target = "plea", arr[] = ["poon", "plee", "same", "poie", "plea", "plie", "poin"] Output: 7 Explanation: toon $\rightarrow$ poon $\rightarrow$ poin $\rightarrow$ poie $\rightarrow$ plie $\rightarrow$ plee $\rightarrow$ plea Input: start = "abcv", target = "ebad", arr[] = ["abcd", "ebad", "ebcd", "xyza"] Output: 4 Explanation: abcv $\rightarrow$ abcd $\rightarrow$ ebcd $\rightarrow$ ebad [Naive Approach]: Using backtracking, explore all possible path We use backtracking to solve this problem because it allows us to systematically explore all possible transformation sequences from the start word to the target word while ensuring we don’t revisit the same word within a given path. In each step, we try every possible one-letter change in the current word and proceed recursively if the resulting word exists in the dictionary and hasn’t been visited yet.Backtracking enables the algorithm to “go back” once it reaches a dead end or completes a path, and then try a different option. This is especially useful in problems like this where multiple paths exist, and we need to find the shortest valid transformation sequence. Although this method is not the most optimized, it is conceptually simple and effective for smaller datasets where performance is not a critical issue. By exploring all valid transformation paths, it guarantees that the minimum number of steps is found among all possible sequences. C++ #include <iostream> #include <vector> #include <string> #include <map> #include <climits> #include <algorithm> using namespace std ; // Recursive function to find the shortest transformation chain int minWordTransform (string start , string target , map < string , int > & mp) { // If start word is the same as target, no transformation is needed if (start == target) return 1 ; int mini = INT_MAX ; // Mark current word as visited mp [start] = 1 ; // Try changing each character of the word for (int i = 0 ; i < start . size (); i ++) { char originalChar = start [i]; // Try all possible lowercase letters at position i for (char ch = 'a' ; ch <= 'z' ; ch ++) { start [i] = ch ; // If the new word exists in dictionary and is not visited if (mp . find (start) != mp . end () && mp [start] == 0) { // Recursive call for next transformation mini = min (mini , 1 + minWordTransform (start , target , mp)); } } // Restore original character before moving to the next position start [i] = originalChar ; } // Mark current word as unvisited (backtracking) mp [start] = 0 ; return mini ; } // Wrapper function to prepare the map and call recursive function int wordLadder (string start , string target , vector < string >& arr) { map < string , int > mp ; // Initialize all words from the dictionary as unvisited for (auto word : arr) { mp [word] = 0 ; } int result = minWordTransform (start , target , mp); if (result == INT_MAX) result = 0 ; return result ; } // Driver code int main () { vector < string > arr = { "poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea" }; string start = "toon" ; string target = "plea" ; cout << wordLadder (start , target , arr) << endl ; return 0 ; } Java import java.util.* ; public class GfG { // Recursive function to find the shortest transformation chain public static int minWordTransform (String start , String target , Map < String , Integer > mp) { // If start word is the same as target, no transformation is needed if (start . equals (target)) return 1 ; int mini = Integer . MAX_VALUE ; // Mark current word as visited mp . put (start , 1); // Try changing each character of the word for (int i = 0 ; i < start . length (); i ++) { char [] chars = start . toCharArray (); char originalChar =

chars [i] ; // Try all possible lowercase letters at position i for (char ch = 'a' ; ch <= 'z' ; ch ++) { chars [i] = ch ; String transformed = new String (chars); // If the new word exists in dictionary and is not visited if (mp . containsKey (transformed) && mp . get (transformed) == 0) { // Recursive call for next transformation mini = Math . min (mini , 1 + minWordTransform (transformed , target , mp)); } } // Restore original character before moving to the next position chars [i] = originalChar ; } // Mark current word as unvisited (backtracking) mp . put (start , 0); return mini ; } // Wrapper function to prepare the map and call recursive function public static int wordLadder (String start , String target , ArrayList < String > arr) { Map < String , Integer > mp = new HashMap <> (); // Initialize all words from the dictionary as unvisited for (String word : arr) { mp . put (word , 0); } int result = minWordTransform (start , target , mp); if (result == Integer . MAX_VALUE) result = 0 ; return result ; } public static void main (String [] args) { ArrayList < String > arr = new ArrayList <> (Arrays . asList ("poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea")); String start = "toon" ; String target = "plea" ; System . out . println (wordLadder (start , target , arr)); } } Python # Recursive function to find the shortest transformation chain def minWordTransform (start , target , mp): # If start word is the same as target, no transformation is needed if start == target : return 1 mini = float ('inf') # Mark current word as visited mp [start] = 1 # Try changing each character of the word for i in range (len (start)): original_char = start [i] # Try all possible lowercase letters at position i for ch in 'abcdefghijklmnopqrstuvwxyz' : new_word = start [: i] + ch + start [i + 1 :] # If the new word exists in dictionary and is not visited if new_word in mp and mp [new_word] == 0 : # Recursive call for next transformation mini = min (mini , 1 + minWordTransform (new_word , target , mp)) # Mark current word as unvisited (backtracking) mp [start] = 0 return mini # Wrapper function to prepare the map and call recursive function def wordLadder (start , target , arr): mp = { word : 0 for word in arr } result = minWordTransform (start , target , mp) if (result == float ('inf')): result = 0 return result # Driver code arr = ["poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea"] start = "toon" target = "plea" print (wordLadder (start , target , arr)) C# using System ; using System.Collections.Generic ; class GfG { // Recursive function to find the shortest transformation chain static int MinWordTransform (string start , string target , Dictionary < string , int > mp){ // If start word is the same as target, no transformation is needed if (start == target) return 1 ; int mini = int . MaxValue ; // Mark current word as visited mp [start] = 1 ; // Try changing each character of the word for (int i = 0 ; i < start . Length ; i ++) { char [] chars = start . ToCharArray (); char originalChar = chars [i]; // Try all possible lowercase letters at position i for (char ch = 'a' ; ch <= 'z' ; ch ++) { chars [i] = ch ; string transformed = new string (chars); // If the new word exists in dictionary and is not visited if (mp . ContainsKey (transformed) && mp [transformed] == 0) { // Recursive call for next transformation mini = Math . Min (mini , 1 + MinWordTransform (transformed , target , mp)); } } // Restore original character before moving to the next position chars [i] = originalChar ; } // Mark current word as unvisited (backtracking) mp [start] = 0 ; return mini ; } // Wrapper function to prepare the map and call recursive function static int WordLadder (string start , string target , List < string > arr){ Dictionary < string , int > mp = new Dictionary < string , int > (); // Initialize all words from the dictionary as unvisited foreach (var word in arr){ mp [word] = 0 ; } int result = minWordTransform (start , target , mp); if (result == int . MaxValue) result = 0 ; return result ; } static void Main (string [] args){ List < string > arr = new List < string > { "poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea" }; string start = "toon" ; string target = "plea" ; Console . WriteLine (WordLadder (start , target , arr)); } } JavaScript // Recursive function to find the shortest transformation chain function minWordTransform (start , target , mp) { // If start word is the same as target, no transformation is needed if (start === target) return 1 ; let mini = Infinity ; // Mark current word as visited mp [start] = 1 ; // Try changing each character of the word for (let i = 0 ; i < start . length ; i ++) { let originalChar = start [i]; // Try all possible lowercase letters at position i for (let chCode = 97 ; chCode <= 122 ; chCode ++) { let ch = String . fromCharCode (chCode); let newWord = start . slice (0 , i) + ch + start . slice (i + 1); // If the new word exists in dictionary and is not visited if (mp . hasOwnProperty (newWord) && mp [newWord] === 0) { // Recursive call for next transformation mini = Math . min (mini , 1 + minWordTransform (newWord , target , mp)); } } } // Mark current word as unvisited (backtracking) mp [start] = 0 ; return mini ; } // Wrapper function to prepare the map and call recursive function function wordLadder (start , target , arr) { let mp = {}; for (let word of arr) { mp [word] = 0 ; } let result = minWordTransform (start , target , mp); if (result == Infinity) result = 0 ; return result ; } // Driver code let arr = ["poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea"]; let start = "toon" ; let target = "plea" ; console . log (wordLadder (start , target , arr)); Output 6 Time Complexity: O(N·26 L)), where N is the number of words in the dictionary and L is the length of each word. Space Complexity: O(N) for storing the dictionary map and the recursive call stack, which can go up to N in the worst case. Using Breadth First Search The idea is to

use BFS to find the smallest chain between start and target . To do so, create a queue words to store the word to visit and push start initially. At each level, go through all the elements stored in queue words, and for each element, alter all of its character for ' a' to 'z ' and one by one and check if the new word is in dictionary or not. If found , push the new word in queue , else continue . Each level of queue defines the length of chain, and once the target is found return the value of that level + 1. C++ // C++ program to find length of the shortest // chain transformation from start to target #include <bits/stdc++.h> using namespace std ; int wordLadder (string start , string target , vector < string >& arr) { // set to keep track of unvisited words unordered_set < string > st (arr . begin (), arr . end ()); // store the current chain length int res = 0 ; int m = start . length (); // queue to store words to visit queue < string > words ; words . push (start); while (! words . empty ()) { res ++ ; int len = words . size (); // iterate through all words at same level for (int i = 0 ; i < len ; ++ i) { string word = words . front (); words . pop (); // For every character of the word for (int j = 0 ; j < m ; ++ j) { // Retain the original character // at the current position char ch = word [j]; // Replace the current character with // every possible lowercase alphabet for (char c = 'a' ; c <= 'z' ; ++ c) { word [j] = c ; // skip the word if already added // or not present in set if (st . find (word) == st . end ()) continue ; // If target word is found if (word == target) return res + 1 ; // remove the word from set st . erase (word); // And push the newly generated word // which will be a part of the chain words . push (word); } // Restore the original character // at the current position word [j] = ch ; } } } return 0 ; } int main () { vector < string > arr = { "poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea" }; string start = "toon" ; string target = "plea" ; cout << wordLadder (start , target , arr); return 0 ; } Java import java.util.* ; class GfG { static int wordLadder (String start , String target , String [] arr) { // Set to keep track of unvisited words Set < String > st = new HashSet < String > (); for (int i = 0 ; i < arr . length ; i ++) st . add (arr [i]); // Store the current chain length int res = 0 ; int m = start . length (); // Queue to store words to visit Queue < String > words = new LinkedList <> (); words . add (start); while (! words . isEmpty ()) { int len = words . size (); res ++ ; // Iterate through all words at the same level for (int i = 0 ; i < len ; ++ i) { String word = words . poll (); // For every character of the word for (int j = 0 ; j < m ; ++ j) { // Retain the original character // at the current position char [] wordArr = word . toCharArray (); char ch = wordArr [j] ; // Replace the current character with // every possible lowercase alphabet for (char c = 'a' ; c <= 'z' ; ++ c) { wordArr [j] = c ; String newWord = new String (wordArr); // Skip the word if already added // or not present in set if (! st . contains (newWord)) continue ; // If target word is found if (newWord . equals (target)) return res + 1 ; // Remove the word from set st . remove (newWord); // And push the newly generated word // which will be a part of the chain words . add (newWord); } // Restore the original character wordArr [j] = ch ; } } } return 0 ; } public static void main (String [] args) { String [] arr = new String [] { "poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea" }; String start = "toon" ; String target = "plea" ; System . out . println (wordLadder (start , target , arr)); } } Python # Python program to find length of the shortest # chain transformation from start to target from collections import deque def wordLadder (start , target , arr): if (start == target): return 0 # set to keep track of unvisited words st = set (arr) # store the current chain length res = 0 m = len (start) # queue to store words to visit words = deque () words . append (start) while words : res += 1 length = len (words) # iterate through all words at same level for _ in range (length): word = words . popleft () # For every character of the word for j in range (m): # Retain the original character # at the current position ch = word [j] # Replace the current character with # every possible lowercase alphabet for c in range (ord ('a'), ord ('z') + 1): word = word [: j] + chr (c) + word [j + 1 :] # skip the word if already added # or not present in set if word not in st : continue # If target word is found if word == target : return res + 1 # remove the word from set st . remove (word) # And push the newly generated word # which will be a part of the chain words . append (word) # Restore the original character # at the current position word = word [: j] + ch + word [j + 1 :] return 0 if __name__ == "__main__" : arr = ["poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea"] start = "toon" target = "plea" print (wordLadder (start , target , arr)) C# // C# program to find length of the shortest // chain transformation from start to target using System ; using System.Collections.Generic ; class GfG { static int WordLadder (string start , string target , string [] arr) { if (start == target) return 0 ; // set to keep track of unvisited words HashSet < string > st = new HashSet < string > (arr); // store the current chain length int res = 0 ; int m = start . Length ; // queue to store words to visit Queue < string > words = new Queue < string > (); words . Enqueue (start); while (words . Count > 0) { res += 1 ; int len = words . Count ; // iterate through all words at same level for (int i = 0 ; i < len ; ++ i) { string word = words . Dequeue (); // For every character of the word for (int j = 0 ; j < m ; ++ j) { // Retain the original character // at the current position char [] wordArray = word . ToCharArray (); // Replace the current character with // every possible lowercase alphabet for (char c = 'a' ; c <= 'z' ; ++ c) { wordArray [j] = c

; string newWord = new string (wordArray); // skip the word if already added // or not present in set if (! st . Contains (newWord)) continue ; // If target word is found if (newWord == target) return res + 1 ; // remove the word from set st . Remove (newWord); // And push the newly generated word // which will be a part of the chain words . Enqueue (newWord); } } } return 0 ; } static void Main () { string [] arr = { "poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea" }; string start = "toon" ; string target = "plea" ; Console . WriteLine (WordLadder (start , target , arr)); } } JavaScript // JavaScript program to find length of the shortest // chain transformation from start to target function wordLadder (start , target , arr) { // set to keep track of unvisited words let st = new Set (arr); // store the current chain length let res = 0 ; let m = start . length ; // queue to store words to visit let words = []; words . push (start); while (words . length > 0) { ++ res ; let len = words . length ; // iterate through all words at same level for (let i = 0 ; i < len ; ++ i) { let word = words . shift (); // For every character of the word for (let j = 0 ; j < m ; ++ j) { // Retain the original character // at the current position let wordArray = word . split (''); let ch = wordArray [j]; // Replace the current character with // every possible lowercase alphabet for (let c = 'a' . charCodeAt (0); c <= 'z' . charCodeAt (0); ++ c) { wordArray [j] = String . fromCharCode (c); let newWord = wordArray . join (''); // skip the word if already added // or not present in set if (! st . has (newWord)) continue ; // If target word is found if (newWord === target) return res + 1 ; // remove the word from set st . delete (newWord); // And push the newly generated word // which will be a part of the chain words . push (newWord); } // Restore the original character // at the current position wordArray [j] = ch ; } } } return 0 ; } // Driver code let arr = ["poon" , "plee" , "same" , "poie" , "plie" , "poin" , "plea"]; let start = "toon" ; let target = "plea" ; console . log (wordLadder (start , target , arr)); Output 7 Time Complexity: O(26 * n * m * m) = O(n * m * m), where n is the size of arr[] and m is the length of each word. Auxiliary Space: O(n * m) Comment Article Tags: Article Tags: Graph Backtracking DSA BFS