Word Break - GeeksforGeeks

Source: https://www.geeksforgeeks.org/word-break-problem-dp-32/

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 Break Last Updated : 23 Jul, 2025 Given a string s and y a dictionary of n words dictionary , check if s can be segmented into a sequence of valid words from the dictionary, separated by spaces. Examples: Input: s = "ilike", dictionary[] = ["i", "like", "gfg"] Output: true Explanation: The string can be segmented as "i like". Input : s = "ilikegfg", dictionary[] = ["i", "like", "man", "india", "gfg"] Output : true Explanation: The string can be segmented as "i like gfg". Input: "ilikemangoes", dictionary = ["i", "like", "gfg"] Output: false Explanation: The string cannot be segmented. Try it on GfG Practice Table of Content [Naive Approach] Using Recursion - O(2^n) Time and O(n) Space [Expected Approach - 1] Using Top-Down DP - O(n^2) Time and O(n+m) Spacce [Expected Approach - 2] Using Bottom Up DP - O(n*m*k) time and O(n) space [Naive Approach] Using Recursion - O(2^n) Time and O(n) Space The idea is to consider each prefix and search for it in dictionary. If the prefix is present in dictionary, we recur for rest of the string (or suffix). If the recursive call for suffix returns true, we return true, otherwise we try next prefix. If we have tried all prefixes and none of them resulted in a solution, we return false. C++ // C++ program to implement word break. #include <bits/stdc++.h> using namespace std ; // Function to check if the given string can be broken // down into words from the word list bool wordBreakRec (int i , string & s , vector < string > & dictionary) { // If end of string is reached, // return true. if (i == s . length ()) return true ; int n = s . length (); string prefix = "" ; // Try every prefix for (int j = i ; j < n ; j ++) { prefix += s [j]; // if the prefix s[i..j] is a dictionary word // and rest of the string can also be broken into // valid words, return true if (find (dictionary . begin (), dictionary . end (), prefix) != dictionary . end () && wordBreakRec (j + 1 , s , dictionary)) { return true ; } } return false ; } bool wordBreak (string & s , vector < string > & dictionary) { return wordBreakRec (0 , s , dictionary); } int main () { string s = "ilike" ; vector < string > dictionary = { "i" , "like" , "gfg" }; cout << (wordBreak (s , dictionary) ? "true" : "false") << endl ; return 0 ; } Java import java.util.* ; class GfG { static boolean wordBreakRec (int i , String s , String [] dictionary) { if (i == s . length ()) return true ; String prefix = "" ; for (int j = i ; j < s . length (); j ++) { prefix += s . charAt (j); // Check if the prefix exists in the dictionary if (Arrays . asList (dictionary). contains (prefix) && wordBreakRec (j + 1 , s , dictionary)) { return true ; } } return false ; } static boolean wordBreak (String s , String [] dictionary) { return wordBreakRec (0 , s , dictionary); } public static void main (String [] args) { String s = "ilike" ; String [] dictionary = { "i" , "like" , "gfg" }; System . out . println (wordBreak (s , dictionary) ? "true" : "false"); } } Python def wordBreakRec (i , s , dictionary): # If end of string is reached, # return true. if i == len (s): return 1 n = len (s) prefix = "" # Try every prefix for j in range (i , n): prefix += s [j] # if the prefix s[i..j] is a dictionary word # and rest of the string can also be broken into # valid words, return true if prefix in dictionary and wordBreakRec (j + 1 , s , dictionary) == 1 : return 1 return 0 def wordBreak (s , dictionary): return wordBreakRec (0 , s , dictionary) if __name__ == "__main__" : s = "ilike" dictionary = { "i" , "like" , "gfg" } print ("true" if wordBreak (s , dictionary) else "false") C# using System ; class Program { static bool WordBreakRec (int index , string s , string [] dictionary) { // If end of the string is reached, return true. if (index == s . Length) return true ; string prefix = "" ; // Try every prefix from the current index for (int j = index ; j < s . Length ; j ++) { prefix += s [j]; // Check if the current prefix exists in the dictionary if (Array . Exists (dictionary , word => word == prefix)) { // If prefix is valid, recursively check for the remaining substring if (WordBreakRec (j + 1 , s , dictionary)) return true ; } } return false ; } static bool WordBreak (string s , string [] dictionary) { return WordBreakRec (0 , s , dictionary); } static void Main () { string [] dictionary = { "i" , "like" , "gfg" }; string s = "ilike" ; Console . WriteLine (WordBreak (s , dictionary) ? "true" :

"false"); } } JavaScript // JavaScript program to implement word break. // Function to check if the given string can be broken // down into words from the word list. // Returns 1 if string can be segmented function wordBreakRec (i , s , dictionary) { // If end of string is reached, // return true. if (i === s . length) return 1 ; let n = s . length ; let prefix = "" ; // Try every prefix for (let j = i ; j < n ; j ++) { prefix += s [j]; // if the prefix s[i..j] is a dictionary word // and rest of the string can also be broken into // valid words, return true if (dictionary . find ((pre) => pre == prefix) !== undefined && wordBreakRec (j + 1 , s , dictionary) === 1) { return 1 ; } } return 0 ; } function wordBreak (s , dictionary) { return wordBreakRec (0 , s , dictionary); } let s = "ilike" ; let dictionary = ["i" , "like" , "gfg"]; console . log (wordBreak (s , dictionary) ? "true" : "false"); Output true [Expected Approach - 1] Using Top-Down DP - O(n^2) Time and O(n+m) Space The idea is to use dynamic programming in the recursive solution to avoid recomputing same subproblems. To further improve the time complexity, store the words of the dictionary in a set to improve the time complexity of looking for a word in dictionary from O(m) to O(1). If we notice carefully, we can observe that the above recursive solution holds the following two properties of Dynamic Programming : 1. Optimal Substructure : To check if the string can be segmented starting from index i , i.e., wordBreakRec(i) , depends on the solutions of the subproblems wordBreakRec(j) where j lies between i and n. Return true if s[i:j] is present in dictionary and wordBreakRec(j) returns true. 2. Overlapping Subproblems : While applying a recursive approach in this problem, we notice that certain subproblems are computed multiple times. For example, for wordBreakRec(0), wordBreakRec(1) and wordBreakRec(2) is called. wordBreakRec(1) will again call wordBreakRec(2). There is only one parameter: i that changes in the recursive solution. So we create a 1D array of size n for memoization . We initialize this array as -1 to indicate nothing is computed initially. Now we modify our recursive solution to first check if the value is -1, then only make recursive calls. This way, we avoid re-computations of the same subproblems. C++ #include <bits/stdc++.h> using namespace std ; bool wordBreakRec (int ind , string & s , vector < string > & dictionary , vector < int > & dp) { if (ind >= s . size ()) { return true ; } if (dp [ind] != -1) return dp [ind]; bool possible = false ; for (int i = 0 ; i < dictionary . size (); i ++) { string temp = dictionary [i]; if (temp . size () > s . size () - ind) continue ; bool ok = true ; int k = ind ; for (int j = 0 ; j < temp . size (); j ++) { if (temp [j] != s [k]) { ok = false ; break ; } else k ++ ; } if (ok) { possible |= wordBreakRec (ind + temp . size (), s , dictionary , dp); } } return dp [ind] = possible ; } bool wordBreak (string s , vector < string > & dictionary) { int n = s . size (); vector < int > dp (n + 1 , -1); string temp = "" ; return wordBreakRec (0 , s , dictionary , dp); } int main () { string s = "ilike" ; vector < string > dictionary = { "i" , "like" , "gfg" }; cout << (wordBreak (s , dictionary) ? "true" : "false") << endl ; return 0 ; } Java import java.util.* ; class GfG { static boolean wordBreakRec (int ind , String s , String [] dict , int [] dp) { if (ind >= s . length ()) { return true ; } if (dp [ind] != - 1) { return dp [ind] == 1 ; } boolean possible = false ; for (String temp : dict) { if (temp . length () > s . length () - ind) { continue ; } boolean ok = true ; int k = ind ; for (int j = 0 ; j < temp . length (); j ++) { if (temp . charAt (j) != s . charAt (k)) { ok = false ; break ; } k ++ ; } if (ok) { possible |= wordBreakRec (ind + temp . length (), s , dict , dp); } } dp [ind] = possible ? 1 : 0 ; return possible ; } public static boolean wordBreak (String s , String [] dict) { int n = s . length (); int [] dp = new int [n + 1] ; Arrays . fill (dp , - 1); return wordBreakRec (0 , s , dict , dp); } public static void main (String [] args) { String s = "ilike" ; String [] dict = { "i" , "like" , "gfg" }; System . out . println (wordBreak (s , dict) ? "true" : "false"); } } Python def wordBreakRec (ind , s , dict , dp): if ind >= len (s): return True if dp [ind] != - 1 : return dp [ind] == 1 possible = False for temp in dict : if len (temp) > len (s) - ind : continue if s [ind : ind + len (temp)] == temp : possible |= wordBreakRec (ind + len (temp), s , dict , dp) dp [ind] = 1 if possible else 0 return possible def word_break (s , dict): n = len (s) dp = [- 1] * (n + 1) return wordBreakRec (0 , s , dict , dp) s = "ilike" dict = ["i" , "like" , "gfg"] print ("true" if word_break (s , dict) else "false") JavaScript function wordBreakRec (ind , s , dict , dp) { if (ind >= s . length) { return true ; } if (dp [ind] !== - 1) { return dp [ind] === 1 ; } let possible = false ; for (let temp of dict) { if (temp . length > s . length - ind) { continue ; } if (s . substring (ind , ind + temp . length) === temp) { possible ||= wordBreakRec (ind + temp . length , s , dict , dp); } } dp [ind] = possible ? 1 : 0 ; return possible ; } function wordBreak (s , dict) { let n = s . length ; let dp = new Array (n + 1). fill (- 1); return wordBreakRec (0 , s , dict , dp); } let s = "ilike" ; let dict = ["i" , "like" , "gfg"]; console . log (wordBreak (s , dict) ? "true" : "false"); Output true [Expected Approach - 2] Using Bottom Up DP - O(n*m*k) time and O(n) space The idea is to use bottom-up dynamic programming to determine if a string can be segmented into dictionary words. Create a boolean array d[] where each position dp[i] represents whether the substring from 0 to that position can be broken into dictionary words. Step by step approach: Start from the beginning of the string and mark it as valid (base case). i.e., dp[0] = true For each position, check if any dictionary word ends at that position and leads to an already valid position. If such a word exists, mark the current position as valid, i.e., dp[i] = true At the end return the last entry of dp[] C++ // C++ program to implement word break. #include <bits/stdc++.h> using namespace std ; bool wordBreak (string & s , vector < string > & dictionary) { int n = s . size (); vector < bool > dp (n + 1 , 0); dp [0] = 1 ; // Traverse through the given string for (int i = 1 ; i <= n ; i ++) { // Traverse through the dictionary words for (string & w : dictionary) { // Check if current word is present // the prefix before the word is also // breakable int start = i - w . size (); if (start >= 0 && dp [start] && s . substr (start , w . size ()) == w) { dp [i] = 1 ; break ; } } } return dp [n]; } int main () { string s = "ilike" ; vector < string > dictionary = { "i" , "like" , "gfg" }; cout << (wordBreak (s , dictionary) ? "true" : "false") << endl ; return 0 ; } Java import java.util.* ; class GfG { static boolean wordBreak (String s , String [] dictionary) { int n = s . length (); boolean [] dp = new boolean [n + 1] ; dp [0] = true ; // Traverse through the given string for (int i = 1 ; i <= n ; i ++) { // Traverse through the dictionary words for (String w : dictionary) { // Check if the current word is present and // the prefix before the word is also // breakable int start = i - w . length (); if (start >= 0 && dp [start] && s . substring (start , start + w . length ()) . equals (w)) { dp [i] = true ; break ; } } } return dp [n] ; // Returning true or false } public static void main (String [] args) { String s = "ilike" ; String [] dictionary = { "i" , "like" , "gfg" }; // Using String array System . out . println (wordBreak (s , dictionary) ? "true" : "false"); } } Python # Python program to implement word break def wordBreak (s , dictionary): n = len (s) dp = [False] * (n + 1) dp [0] = True # Traverse through the given string for i in range (1 , n + 1): # Traverse through the dictionary words for w in dictionary : # Check if current word is present # the prefix before the word is also # breakable start = i - len (w) if start >= 0 and dp [start] and s [start : start + len (w)] == w : dp [i] = True break return 1 if dp [n] else 0 if __name__ == '__main__' : s = "ilike" dictionary = ["i" , "like" , "gfg"] print ("true" if wordBreak (s , dictionary) else "false") C# using System ; using System.Collections.Generic ; class GfG { static bool wordBreak (string s , string [] dictionary) { int n = s . Length ; bool [] dp = new bool [n + 1]; dp [0] = true ; // Traverse through the given string for (int i = 1 ; i <= n ; i ++) { // Traverse through the dictionary words foreach (string w in dictionary) { // Check if current word is present and the // prefix before the word is also breakable int start = i - w . Length ; if (start >= 0 && dp [start] && s . Substring (start , w . Length) == w) { dp [i] = true ; break ; } } } return dp [n]; // Return true if word break is // possible, else false } public static void Main () { string s = "ilike" ; string [] dictionary = { "i" , "like" , "gfg" }; // Using string array Console . WriteLine (wordBreak (s , dictionary) ? "true" : "false"); } } JavaScript // JavaScript program to implement word break function wordBreak (s , dictionary) { const n = s . length ; const dp = new Array (n + 1). fill (false); dp [0] = true ; // Traverse through the given string for (let i = 1 ; i <= n ; i ++) { // Traverse through the dictionary words for (const w of dictionary) { // Check if current word is present // the prefix before the word is also // breakable const start = i - w . length ; if (start >= 0 && dp [start] && s . substring (start , start + w . length) === w) { dp [i] = true ; break ; } } } return (dp [n]) ? 1 : 0 ; } const s = "ilike" ; const dictionary = ["i" , "like" , "gfg"]; console . log (wordBreak (s , dictionary) ? "true" : "false"); Output true Time Complexity: O(n * m * k), where n is the length of string and m is the number of dictionary words and k is the length of maximum sized string in dictionary. Space Complexity: O(n) Related Articles: Word Break Problem | (Trie solution) Word Break Problem using Backtracking Comment Article Tags: Article Tags: Dynamic Programming DSA Microsoft Amazon Google Walmart Zoho MAQ Software IBM + 5 More