Longest repeating and non-overlapping substring - GeeksforGeeks

Source: https://www.geeksforgeeks.org/longest-repeating-and-non-overlapping-substring/

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 repeating and non-overlapping substring Last Updated : 16 Nov, 2024 Given a string s , the task is to find the longest repeating non-overlapping substring in it. In other words, find 2 identical substrings of maximum length which do not overlap. Return -1 if no such string exists. Note: Multiple Answers are possible but we have to return the substring whose first occurrence is earlier. Examples: Input: s = "acdcdacdc" Output: "acdc" Explanation: The string "acdc" is the longest Substring of s which is repeating but not overlapping. Input: s = "geeksforgeeks" Output: "geeks" Explanation: The string "geeks" is the longest subString of s which is repeating but not overlapping. Try it on GfG Practice Table of Content Using Brute Force Method - O(n^3) Time and O(n) Space Using Top-Down DP (Memoization) - O(n^2) Time and O(n^2) Space Using Bottom-Up DP (Tabulation) - O(n^2) Time and O(n^2) Space Using Space Optimized DP – O(n^2) Time and O(n) Space Using Brute Force Method - O(n^3) Time and O(n) Space The idea is to generate all the possible substrings and check if the substring exists in the remaining string. If substring exists and its length is greater than answer substring, then set answer to current substring. C++ // C++ program to find longest repeating // and non-overlapping substring // using recursion #include <bits/stdc++.h> using namespace std ; string longestSubstring (string & s) { int n = s . length (); string ans = "" ; int len = 0 ; int i = 0 , j = 0 ; while (i < n && j < n) { string curr = s . substr (i , j - i + 1); // If substring exists, compare its length // with ans if (s . find (curr , j + 1) != string :: npos && j - i + 1 > len) { len = j - i + 1 ; ans = curr ; } // Otherwise increment i else i ++ ; j ++ ; } return len > 0 ? ans : "-1" ; } int main () { string s = "geeksforgeeks" ; cout << longestSubstring (s) << endl ; return 0 ; } Java // Java program to find longest repeating // and non-overlapping substring // using recursion class GfG { static String longestSubstring (String s) { int n = s . length (); String ans = "" ; int len = 0 ; int i = 0 , j = 0 ; while (i < n && j < n) { String curr = s . substring (i , j + 1); // If substring exists, compare its length // with ans if (s . indexOf (curr , j + 1) != - 1 && j - i + 1 > len) { len = j - i + 1 ; ans = curr ; } // Otherwise increment i else i ++ ; j ++ ; } return len > 0 ? ans : "-1" ; } public static void main (String [] args) { String s = "geeksforgeeks" ; System . out . println (longestSubstring (s)); } } Python # Python program to find longest repeating # and non-overlapping substring # using recursion def longestSubstring (s): n = len (s) ans = "" lenAns = 0 i , j = 0 , 0 while i < n and j < n : curr = s [i : j + 1] # If substring exists, compare its length # with ans if s . find (curr , j + 1) != - 1 and j - i + 1 > lenAns : lenAns = j - i + 1 ans = curr # Otherwise increment i else : i += 1 j += 1 if lenAns > 0 : return ans return "-1" if __name__ == "__main__" : s = "geeksforgeeks" print (longestSubstring (s)) C# // C# program to find longest repeating // and non-overlapping substring // using recursion using System ; class GfG { static string longestSubstring (string s) { int n = s . Length ; string ans = "" ; int len = 0 ; int i = 0 , j = 0 ; while (i < n && j < n) { string curr = s . Substring (i , j - i + 1); // If substring exists, compare its length // with ans if (s . IndexOf (curr , j + 1) != - 1 && j - i + 1 > len) { len = j - i + 1 ; ans = curr ; } // Otherwise increment i else i ++ ; j ++ ; } return len > 0 ? ans : "-1" ; } static void Main (string [] args) { string s = "geeksforgeeks" ; Console . WriteLine (longestSubstring (s)); } } JavaScript // JavaScript program to find longest repeating // and non-overlapping substring // using recursion function longestSubstring (s) { const n = s . length ; let ans = "" ; let len = 0 ; let i = 0 , j = 0 ; while (i < n && j < n) { const curr = s . substring (i , j + 1); // If substring exists, compare its length // with ans if (s . indexOf (curr , j + 1) !== - 1 && j - i + 1 > len) { len = j - i + 1 ; ans = curr ; } //

Otherwise increment i else i ++ ; j ++ ; } return len > 0 ? ans : "-1" ; } const s = "geeksforgeeks" ; console . log (longestSubstring (s)); Output geeks Using Top-Down DP (Memoization) - O(n^2) Time and O(n^2) Space The approach is to compute the longest repeating suffix for all prefix pairs in the string s . For indices i and j , if s[i] == s[j], then recursively compute suffix(i+1, j+1) and set suffix(i, j) as min(suffix(i+1, j+1) + 1, j - i - 1) to prevent overlap . If the characters do not match, set suffix(i, j) = 0. Note: To avoid overlapping we have to ensure that the length of suffix is less than (j-i) at any instant. The maximum value of suffix(i, j) provides the length of the longest repeating substring and the substring itself can be found using the length and the starting index of the common suffix. suffix(i, j) stores the length of the longest common suffix between indices i and j, ensuring it doesn’t exceed j - i - 1 to avoid overlap. C++ // C++ program to find longest repeating // and non-overlapping substring // using memoization #include <bits/stdc++.h> using namespace std ; int findSuffix (int i , int j , string & s , vector < vector < int >> & memo) { // base case if (j == s . length ()) return 0 ; // return memoized value if (memo [i][j] != -1) return memo [i][j]; // if characters match if (s [i] == s [j]) { memo [i][j] = 1 + min (findSuffix (i + 1 , j + 1 , s , memo), j - i - 1); } else { memo [i][j] = 0 ; } return memo [i][j]; } string longestSubstring (string s) { int n = s . length (); vector < vector < int >> memo (n , vector < int > (n , -1)); // find length of non-overlapping // substrings for all pairs (i,j) for (int i = 0 ; i < n ; i ++) { for (int j = i + 1 ; j < n ; j ++) { findSuffix (i , j , s , memo); } } string ans = "" ; int ansLen = 0 ; // If length of suffix is greater // than ansLen, update ans and ansLen for (int i = 0 ; i < n ; i ++) { for (int j = i + 1 ; j < n ; j ++) { if (memo [i][j] > ansLen) { ansLen = memo [i][j]; ans = s . substr (i , ansLen); } } } return ansLen > 0 ? ans : "-1" ; } int main () { string s = "geeksforgeeks" ; cout << longestSubstring (s) << endl ; return 0 ; } Java // Java program to find longest repeating // and non-overlapping substring // using memoization import java.util.Arrays ; class GfG { static int findSuffix (int i , int j , String s , int [][] memo) { // base case if (j == s . length ()) return 0 ; // return memoized value if (memo [i][j] != - 1) return memo [i][j] ; // if characters match if (s . charAt (i) == s . charAt (j)) { memo [i][j] = 1 + Math . min (findSuffix (i + 1 , j + 1 , s , memo), j - i - 1); } else { memo [i][j] = 0 ; } return memo [i][j] ; } static String longestSubstring (String s) { int n = s . length (); int [][] memo = new int [n][n] ; for (int [] row : memo) { Arrays . fill (row , - 1); } // find length of non-overlapping // substrings for all pairs (i, j) for (int i = 0 ; i < n ; i ++) { for (int j = i + 1 ; j < n ; j ++) { findSuffix (i , j , s , memo); } } String ans = "" ; int ansLen = 0 ; // If length of suffix is greater // than ansLen, update ans and ansLen for (int i = 0 ; i < n ; i ++) { for (int j = i + 1 ; j < n ; j ++) { if (memo [i][j] > ansLen) { ansLen = memo [i][j] ; ans = s . substring (i , i + ansLen); } } } return ansLen > 0 ? ans : "-1" ; } public static void main (String [] args) { String s = "geeksforgeeks" ; System . out . println (longestSubstring (s)); } } Python # Python program to find longest repeating # and non-overlapping substring # using memoization def findSuffix (i , j , s , memo): # base case if j == len (s): return 0 # return memoized value if memo [i][j] != - 1 : return memo [i][j] # if characters match if s [i] == s [j]: memo [i][j] = 1 + min (findSuffix (i + 1 , j + 1 , s , memo), \ j - i - 1) else : memo [i][j] = 0 return memo [i][j] def longestSubstring (s): n = len (s) memo = [[- 1] * n for _ in range (n)] # find length of non-overlapping # substrings for all pairs (i, j) for i in range (n): for j in range (i + 1 , n): findSuffix (i , j , s , memo) ans = "" ansLen = 0 # If length of suffix is greater # than ansLen, update ans and ansLen for i in range (n): for j in range (i + 1 , n): if memo [i][j] > ansLen : ansLen = memo [i][j] ans = s [i : i + ansLen] if ansLen > 0 : return ans return "-1" if __name__ == "__main__" : s = "geeksforgeeks" print (longestSubstring (s)) C# // C# program to find longest repeating // and non-overlapping substring // using memoization using System ; class GfG { static int findSuffix (int i , int j , string s , int [,] memo) { // base case if (j == s . Length) return 0 ; // return memoized value if (memo [i , j] != - 1) return memo [i , j]; // if characters match if (s [i] == s [j]) { memo [i , j] = 1 + Math . Min (findSuffix (i + 1 , j + 1 , s , memo), j - i - 1); } else { memo [i , j] = 0 ; } return memo [i , j]; } static string longestSubstring (string s) { 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 ; } } // find length of non-overlapping // substrings for all pairs (i, j) for (int i = 0 ; i < n ; i ++) { for (int j = i + 1 ; j < n ; j ++) { findSuffix (i , j , s , memo); } } string ans = "" ; int ansLen = 0 ; // If length of suffix is greater // than ansLen, update ans and ansLen for (int i = 0 ; i < n ; i ++) { for (int j = i + 1 ; j < n ; j ++) { if (memo [i , j] > ansLen) { ansLen = memo [i , j]; ans = s . Substring (i , ansLen); } } } return ansLen > 0 ? ans : "-1" ; } static void Main (string [] args) { string s = "geeksforgeeks" ; Console . WriteLine (longestSubstring (s)); } } JavaScript // JavaScript program to find longest repeating // and non-overlapping substring // using memoization function findSuffix (i , j , s , memo) { // base case if (j === s . length) return 0 ; // return memoized value if (memo [i][j] !== - 1) return memo [i][j]; // if characters match if (s [i] === s [j]) { memo [i][j] = 1 + Math . min (findSuffix (i + 1 , j + 1 , s , memo), j - i - 1); } else { memo [i][j] = 0 ; } return memo [i][j]; } function longestSubstring (s) { const n = s

. length ; const memo = Array . from ({ length : n }, () => Array (n). fill (- 1)); // find length of non-overlapping // substrings for all pairs (i, j) for (let i = 0 ; i < n ; i ++) { for (let j = i + 1 ; j < n ; j ++) { findSuffix (i , j , s , memo); } } let ans = "" ; let ansLen = 0 ; // If length of suffix is greater // than ansLen, update ans and ansLen for (let i = 0 ; i < n ; i ++) { for (let j = i + 1 ; j < n ; j ++) { if (memo [i][j] > ansLen) { ansLen = memo [i][j]; ans = s . substring (i , i + ansLen); } } } return ansLen > 0 ? ans : "-1" ; } const s = "geeksforgeeks" ; console . log (longestSubstring (s)); Output geeks Using Bottom-Up DP (Tabulation) - O(n^2) Time and O(n^2) Space The idea is to create a 2D matrix of size (n+1)*(n+1) and calculate the longest repeating suffixes for all index pairs (i, j) iteratively. We start from the end of the string and work backwards to fill the table. For each (i, j), if s[i] == s[j], we set suffix[i][j] to min(suffix[i+1][j+1]+1, j-i-1) to avoid overlap; otherwise, suffix[i][j] = 0. C++ // C++ program to find longest repeating // and non-overlapping substring // using tabulation #include <bits/stdc++.h> using namespace std ; string longestSubstring (string s) { int n = s . length (); vector < vector < int >> dp (n + 1 , vector < int > (n + 1 , 0)); string ans = "" ; int ansLen = 0 ; // find length of non-overlapping // substrings for all pairs (i,j) for (int i = n -1 ; i >= 0 ; i --) { for (int j = n -1 ; j > i ; j --) { // if characters match, set value // and compare with ansLen. if (s [i] == s [j]) { dp [i][j] = 1 + min (dp [i + 1][j + 1], j - i -1); if (dp [i][j] >= ansLen) { ansLen = dp [i][j]; ans = s . substr (i , ansLen); } } } } return ansLen > 0 ? ans : "-1" ; } int main () { string s = "geeksforgeeks" ; cout << longestSubstring (s) << endl ; return 0 ; } Java // Java program to find longest repeating // and non-overlapping substring // using tabulation class GfG { static String longestSubstring (String s) { int n = s . length (); int [][] dp = new int [n + 1][n + 1] ; String ans = "" ; int ansLen = 0 ; // find length of non-overlapping // substrings for all pairs (i, j) for (int i = n - 1 ; i >= 0 ; i --) { for (int j = n - 1 ; j > i ; j --) { // if characters match, set value // and compare with ansLen. if (s . charAt (i) == s . charAt (j)) { dp [i][j] = 1 + Math . min (dp [i + 1][j + 1] , j - i - 1); if (dp [i][j] >= ansLen) { ansLen = dp [i][j] ; ans = s . substring (i , i + ansLen); } } } } return ansLen > 0 ? ans : "-1" ; } public static void main (String [] args) { String s = "geeksforgeeks" ; System . out . println (longestSubstring (s)); } } Python # Python program to find longest repeating # and non-overlapping substring # using tabulation def longestSubstring (s): n = len (s) dp = [[0] * (n + 1) for _ in range (n + 1)] ans = "" ansLen = 0 # find length of non-overlapping # substrings for all pairs (i, j) for i in range (n - 1 , - 1 , - 1): for j in range (n - 1 , i , - 1): # if characters match, set value # and compare with ansLen. if s [i] == s [j]: dp [i][j] = 1 + min (dp [i + 1][j + 1], j - i - 1) if dp [i][j] >= ansLen : ansLen = dp [i][j] ans = s [i : i + ansLen] return ans if ansLen > 0 else "-1" if __name__ == "__main__" : s = "geeksforgeeks" print (longestSubstring (s)) C# // C# program to find longest repeating // and non-overlapping substring // using tabulation using System ; class GfG { static string longestSubstring (string s) { int n = s . Length ; int [,] dp = new int [n + 1 , n + 1]; string ans = "" ; int ansLen = 0 ; // find length of non-overlapping // substrings for all pairs (i, j) for (int i = n - 1 ; i >= 0 ; i --) { for (int j = n - 1 ; j > i ; j --) { // if characters match, set value // and compare with ansLen. if (s [i] == s [j]) { dp [i , j] = 1 + Math . Min (dp [i + 1 , j + 1], j - i - 1); if (dp [i , j] >= ansLen) { ansLen = dp [i , j]; ans = s . Substring (i , ansLen); } } } } return ansLen > 0 ? ans : "-1" ; } static void Main (string [] args) { string s = "geeksforgeeks" ; Console . WriteLine (longestSubstring (s)); } } JavaScript // JavaScript program to find longest repeating // and non-overlapping substring // using tabulation function longestSubstring (s) { const n = s . length ; const dp = Array . from ({ length : n + 1 }, () => Array (n + 1). fill (0)); let ans = "" ; let ansLen = 0 ; // find length of non-overlapping // substrings for all pairs (i, j) for (let i = n - 1 ; i >= 0 ; i --) { for (let j = n - 1 ; j > i ; j --) { // if characters match, set value // and compare with ansLen. if (s [i] === s [j]) { dp [i][j] = 1 + Math . min (dp [i + 1][j + 1], j - i - 1); if (dp [i][j] >= ansLen) { ansLen = dp [i][j]; ans = s . substring (i , i + ansLen); } } } } return ansLen > 0 ? ans : "-1" ; } const s = "geeksforgeeks" ; console . log (longestSubstring (s)); Output geeks Using Space Optimized DP – O(n^2) Time and O(n) Space The idea is to use a single 1D array instead of a 2D matrix by keeping track of only the "next row" values required to compute suffix[i][j]. Since each value s uffix[i][j] depends only on suffix[i+1][j+1] in the row below, we can maintain the previous row's values in a 1D array and update them iteratively for each row. C++ // C++ program to find longest repeating // and non-overlapping substring // using space optimised #include <bits/stdc++.h> using namespace std ; string longestSubstring (string s) { int n = s . length (); vector < int > dp (n + 1 , 0); string ans = "" ; int ansLen = 0 ; // find length of non-overlapping // substrings for all pairs (i,j) for (int i = n -1 ; i >= 0 ; i --) { for (int j = i ; j < n ; j ++) { // if characters match, set value // and compare with ansLen. if (s [i] == s [j]) { dp [j] = 1 + min (dp [j + 1], j - i -1); if (dp [j] >= ansLen) { ansLen = dp [j]; ans = s . substr (i , ansLen); } } else dp [j] = 0 ; } } return ansLen > 0 ? ans : "-1" ; } int main () { string s = "geeksforgeeks" ; cout << longestSubstring (s) << endl ; return 0 ; } Java // Java program to find longest repeating // and non-overlapping substring // using space optimised class GfG {

static String longestSubstring (String s) { int n = s . length (); int [] dp = new int [n + 1] ; String ans = "" ; int ansLen = 0 ; // find length of non-overlapping // substrings for all pairs (i, j) for (int i = n - 1 ; i >= 0 ; i --) { for (int j = i ; j < n ; j ++) { // if characters match, set value // and compare with ansLen. if (s . charAt (i) == s . charAt (j)) { dp [j] = 1 + Math . min (dp [j + 1] , j - i - 1); if (dp [j] >= ansLen) { ansLen = dp [j] ; ans = s . substring (i , i + ansLen); } } else { dp [j] = 0 ; } } } return ansLen > 0 ? ans : "-1" ; } public static void main (String [] args) { String s = "geeksforgeeks" ; System . out . println (longestSubstring (s)); } } Python # Python program to find longest repeating # and non-overlapping substring # using space optimised def longestSubstring (s): n = len (s) dp = [0] * (n + 1) ans = "" ansLen = 0 # find length of non-overlapping # substrings for all pairs (i, j) for i in range (n - 1 , - 1 , - 1): for j in range (i , n): # if characters match, set value # and compare with ansLen. if s [i] == s [j]: dp [j] = 1 + min (dp [j + 1], j - i - 1) if dp [j] >= ansLen : ansLen = dp [j] ans = s [i : i + ansLen] else : dp [j] = 0 return ans if ansLen > 0 else "-1" if __name__ == "__main__" : s = "geeksforgeeks" print (longestSubstring (s)) C# // C# program to find longest repeating // and non-overlapping substring // using space optimised using System ; class GfG { static string longestSubstring (string s) { int n = s . Length ; int [] dp = new int [n + 1]; string ans = "" ; int ansLen = 0 ; // find length of non-overlapping // substrings for all pairs (i, j) for (int i = n - 1 ; i >= 0 ; i --) { for (int j = i ; j < n ; j ++) { // if characters match, set value // and compare with ansLen. if (s [i] == s [j]) { dp [j] = 1 + Math . Min (dp [j + 1], j - i - 1); if (dp [j] >= ansLen) { ansLen = dp [j]; ans = s . Substring (i , ansLen); } } else { dp [j] = 0 ; } } } return ansLen > 0 ? ans : "-1" ; } static void Main (string [] args) { string s = "geeksforgeeks" ; Console . WriteLine (longestSubstring (s)); } } JavaScript // JavaScript program to find longest repeating // and non-overlapping substring // using space optimised function longestSubstring (s) { const n = s . length ; const dp = new Array (n + 1). fill (0); let ans = "" ; let ansLen = 0 ; // find length of non-overlapping // substrings for all pairs (i, j) for (let i = n - 1 ; i >= 0 ; i --) { for (let j = i ; j < n ; j ++) { // if characters match, set value // and compare with ansLen. if (s [i] === s [j]) { dp [j] = 1 + Math . min (dp [j + 1], j - i - 1); if (dp [j] >= ansLen) { ansLen = dp [j]; ans = s . substring (i , i + ansLen); } } else { dp [j] = 0 ; } } } return ansLen > 0 ? ans : "-1" ; } const s = "geeksforgeeks" ; console . log (longestSubstring (s)); Output geeks Related articles: Longest Common Substring Comment Article Tags: Article Tags: Strings DSA