Longest Bitonic Subsequence - GeeksforGeeks

Source: https://www.geeksforgeeks.org/longest-bitonic-subsequence-dp-15/

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 Bitonic Subsequence Last Updated : 23 Jul, 2025 Given an array arr[] containing n positive integers, a subsequence of numbers is called bitonic if it is first strictly increasing, then strictly decreasing. The task is to find th e length of the longest bitonic subsequence. Note: Only strictly increasing (no decreasing part) or a strictly decreasing sequence should not be considered as a bitonic sequence. Examples: Input: arr[]= [12, 11, 40, 5, 3, 1] Output : 5 Explanation: The Longest Bitonic Subsequence is {12, 40, 5, 3, 1} which is of length 5. Input: arr[] = [80, 60, 30] Output: 0 Explanation: There is no possible Bitonic Subsequence. Try it on GfG Practice Table of Content Using Recursion - O(n*(2^n)) 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) Space Using Recursion - O(n*(2^n)) Time and O(n) Space The recursive approach for finding the longest bitonic subsequence relies on breaking the problem into two parts for each potential peak: Calculate the Longest Increasing Subsequence (LIS) ending at the current element. Calculate the Longest Decreasing Subsequence (LDS) starting from the current element. At each step, the algorithm explores two choices: Include the current element in the subsequence if it satisfies the increasing or decreasing condition. Skip the current element and move to the next index. Recurrence Relation: For any index i, the solution involves: Finding the LIS to the left: if nums[i] < nums[prev] LIS(i, prev) = max(LIS(i-1, prev),1 + LIS(i-1, i)) Finding the LDS to the right: if nums[i] < nums[prev] LDS(i, prev) = max(LDS(i+1, prev), 1 + LDS(i+1, i)) Base Cases: For LIS: LIS(i, prev) = 0, if i < 0 For LDS: LDS(i, prev) = 0, if i >= n C++ // C++ implementation to find longest Bitonic // subsequence using Recursion #include <bits/stdc++.h> using namespace std ; // Function to find the longest decreasing subsequence // to the left int left (int prev , int idx , vector < int >& arr) { if (idx < 0) { return 0 ; } // Check if nums[idx] can be included // in decreasing subsequence int include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + left (idx , idx - 1 , arr); } // Return the maximum of including // or excluding nums[idx] return max (include , left (prev , idx - 1 , arr)); } // Function to find the longest decreasing // subsequence to the right int right (int prev , int idx , vector < int >& arr) { if (idx >= arr . size ()) { return 0 ; } // Check if nums[idx] can be included // in decreasing subsequence int include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + right (idx , idx + 1 , arr); } // Return the maximum of including or // excluding nums[idx] return max (include , right (prev , idx + 1 , arr)); } // Function to find the longest bitonic sequence int LongestBitonicSequence (vector < int >& arr) { int maxLength = 0 ; // Iterate over potential peaks in the array for (int i = 1 ; i < arr . size () - 1 ; i ++) { // Find the longest decreasing subsequences // on both sides of arr[i] int leftLen = left (i , i - 1 , arr); int rightLen = right (i , i + 1 , arr); // Ensure both left and right subsequences are valid if (leftLen == 0 || rightLen == 0) { leftLen = 0 ; rightLen = 0 ; } // Update maximum bitonic sequence length maxLength = max (maxLength , leftLen + rightLen + 1); } // If no valid bitonic sequence, return 0 return (maxLength < 3) ? 0 : maxLength ; } int main () { vector < int > arr = { 12 , 11 , 40 , 5 , 3 , 1 }; cout << LongestBitonicSequence (arr) << endl ; return 0 ; } Java // Java implementation to find longest Bitonic // subsequence using Recursion import java.util.ArrayList ; import java.util.* ; class GfG { // Function to find the longest decreasing subsequence // to the left static int left (int prev , int idx , ArrayList < Integer > arr) { if (idx < 0) { return 0 ; } // Check if nums[idx] can be included // in decreasing subsequence int include = 0 ; if (arr . get (idx) < arr . get (prev)) { include = 1 + left (idx , idx - 1 , arr); } // Return the maximum of including // or excluding nums[idx] return Math . max (include , left (prev , idx - 1 , arr)); } // Function to find the longest decreasing // subsequence to the right static int right (int prev , int idx , ArrayList <

Integer > arr) { if (idx >= arr . size ()) { return 0 ; } // Check if nums[idx] can be included // in decreasing subsequence int include = 0 ; if (arr . get (idx) < arr . get (prev)) { include = 1 + right (idx , idx + 1 , arr); } // Return the maximum of including or // excluding nums[idx] return Math . max (include , right (prev , idx + 1 , arr)); } // Function to find the longest bitonic sequence static int LongestBitonicSequence (ArrayList < Integer > arr) { int maxLength = 0 ; // Iterate over potential peaks in the array for (int i = 1 ; i < arr . size () - 1 ; i ++) { // Find the longest decreasing subsequences // on both sides of nums[i] int leftLen = left (i , i - 1 , arr); int rightLen = right (i , i + 1 , arr); // Ensure both left and right subsequences are valid if (leftLen == 0 || rightLen == 0) { leftLen = 0 ; rightLen = 0 ; } // Update maximum bitonic sequence length maxLength = Math . max (maxLength , leftLen + rightLen + 1); } // If no valid bitonic sequence, return 0 return (maxLength < 3) ? 0 : maxLength ; } public static void main (String [] args) { ArrayList < Integer > arr = new ArrayList <> (Arrays . asList (12 , 11 , 40 , 5 , 3 , 1)); System . out . println (LongestBitonicSequence (arr)); } } Python # Python implementation to find longest Bitonic # subsequence using Recursion # Function to find the longest decreasing # subsequence to the left def left (prev , idx , arr): if idx < 0 : return 0 # Check if arr[idx] can be included in the # decreasing subsequence include = 0 if arr [idx] < arr [prev]: include = 1 + left (idx , idx - 1 , arr) # Return the maximum of including or excluding arr[idx] return max (include , left (prev , idx - 1 , arr)) # Function to find the longest decreasing subsequence # to the right def right (prev , idx , arr): if idx >= len (arr): return 0 # Check if nums[idx] can be included in the # decreasing subsequence include = 0 if arr [idx] < arr [prev]: include = 1 + right (idx , idx + 1 , arr) # Return the maximum of including or excluding arr[idx] return max (include , right (prev , idx + 1 , arr)) # Function to find the longest bitonic sequence def LongestBitonicSequence (arr): max_length = 0 # Iterate over potential peaks in the array for i in range (1 , len (arr) - 1): # Find the longest decreasing subsequences # on both sides of arr[i] left_len = left (i , i - 1 , arr) right_len = right (i , i + 1 , arr) # Ensure both left and right subsequences are valid if left_len == 0 or right_len == 0 : left_len = 0 right_len = 0 # Update maximum bitonic sequence length max_length = max (max_length , left_len + right_len + 1) # If no valid bitonic sequence, return 0 return 0 if max_length < 3 else max_length if __name__ == "__main__" : arr = [12 , 11 , 40 , 5 , 3 , 1] print (LongestBitonicSequence (arr)) C# // C# implementation to find longest Bitonic // subsequence using Recursion using System ; using System.Collections.Generic ; class GfG { // Function to find the longest decreasing // subsequence to the left static int Left (int prev , int idx , List < int > arr) { if (idx < 0) { return 0 ; } // Check if nums[idx] can be included in // decreasing subsequence int include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + Left (idx , idx - 1 , arr); } // Return the maximum of including or excluding nums[idx] return Math . Max (include , Left (prev , idx - 1 , arr)); } // Function to find the longest decreasing // subsequence to the right static int Right (int prev , int idx , List < int > arr) { if (idx >= arr . Count) { return 0 ; } // Check if nums[idx] can be included // in decreasing subsequence int include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + Right (idx , idx + 1 , arr); } // Return the maximum of including or excluding nums[idx] return Math . Max (include , Right (prev , idx + 1 , arr)); } // Function to find the longest bitonic sequence static int LongestBitonicSequence (List < int > arr) { int maxLength = 0 ; // Iterate over potential peaks in the array for (int i = 1 ; i < arr . Count - 1 ; i ++) { // Find the longest decreasing subsequences // on both sides of nums[i] int leftLen = Left (i , i - 1 , arr); int rightLen = Right (i , i + 1 , arr); // Ensure both left and right subsequences are valid if (leftLen == 0 || rightLen == 0) { leftLen = 0 ; rightLen = 0 ; } // Update maximum bitonic sequence length maxLength = Math . Max (maxLength , leftLen + rightLen + 1); } // If no valid bitonic sequence, return 0 return (maxLength < 3) ? 0 : maxLength ; } static void Main () { List < int > arr = new List < int > { 12 , 11 , 40 , 5 , 3 , 1 }; Console . WriteLine (LongestBitonicSequence (arr)); } } JavaScript // Javascript implementation to find longest Bitonic // subsequence using Recursion<script> // Function to find the longest decreasing subsequence to the left function left (prev , idx , arr) { if (idx < 0) { return 0 ; } // Check if arr[idx] can be included in decreasing subsequence let include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + left (idx , idx - 1 , arr); } // Return the maximum of including or excluding arr[idx] return Math . max (include , left (prev , idx - 1 , arr)); } // Function to find the longest decreasing // subsequence to the right function right (prev , idx , arr) { if (idx >= arr . length) { return 0 ; } // Check if arr[idx] can be included in // decreasing subsequence let include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + right (idx , idx + 1 , arr); } // Return the maximum of including or // excluding arr[idx] return Math . max (include , right (prev , idx + 1 , arr)); } // Function to find the longest bitonic sequence function longestBitonicSequence (nums) { let maxLength = 0 ; // Iterate over potential peaks in the array for (let i = 1 ; i < arr . length - 1 ; i ++) { // Find the longest decreasing subsequences // on both sides of arr[i] let leftLen = left (i , i - 1 , arr); let rightLen = right (i , i + 1 , arr); // Ensure both left and right subsequences are valid if (leftLen === 0 || rightLen === 0) { leftLen = 0 ; rightLen = 0 ; } //

Update maximum bitonic sequence length maxLength = Math . max (maxLength , leftLen + rightLen + 1); } // If no valid bitonic sequence, return 0 return maxLength < 3 ? 0 : maxLength ; } const arr = [12 , 11 , 40 , 5 , 3 , 1]; console . log (longestBitonicSequence (arr)); Output 5 Using Top-Down DP (Memoization) - O(n^2) Time and O(n^2) Space 1. Optimal Substructure: The solution for finding the longest bitonic subsequence around a peak element can be derived from solutions to smaller subproblems. Specifically: The left subsequence is determined by comparing each element on the left to the current peak, recursively finding the longest decreasing sequence. left(prev, idx) = max(1 + left(idx, idx - 1), left(prev, idx - 1)) if arr[idx] < arr[prev]. Otherwise, it is left(prev, idx - 1). Base case: left(prev, idx) = 0 if idx < 0. The right subsequence is similarly found by exploring decreasing sequences to the right of the peak. r ight(prev, idx) = max(1 + right(idx, idx + 1), right(prev, idx + 1)) if arr[idx] < arr[prev]. Otherwise, it is right(prev, idx + 1). Base case: right(prev, idx) = 0 if idx >= arr.size(). Total length at a peak nums[i] is given by: Total Length = leftLen + rightLen + 1. 2. Overlapping Subproblems: Subproblems like left(prev, idx) or right(prev, idx) for specific values of prev and idx are repeatedly computed, memoization avoids redundant calculations by storing results in 2D arrays leftMemo and rightMemo. If leftMemo[prev][idx] != -1 or rightMemo[prev][idx] != -1 , return the stored result. C++ // C++ implementation to find longest Bitonic // subsequence using Recursion and Memoization #include <bits/stdc++.h> using namespace std ; // Function to find the longest decreasing subsequence // to the left with memoization int left (int prev , int idx , vector < int >& arr , vector < vector < int >>& leftMemo) { if (idx < 0) { return 0 ; } if (leftMemo [prev][idx] != -1) { return leftMemo [prev][idx]; } // Check if arr[idx] can be included in // decreasing subsequence int include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + left (idx , idx - 1 , arr , leftMemo); } // Store and return the result return leftMemo [prev][idx] = max (include , left (prev , idx - 1 , arr , leftMemo)); } // Function to find the longest decreasing subsequence // to the right with memoization int right (int prev , int idx , vector < int >& arr , vector < vector < int >>& rightMemo) { if (idx >= arr . size ()) { return 0 ; } if (rightMemo [prev][idx] != -1) { return rightMemo [prev][idx]; } // Check if arr[idx] can be included // in decreasing subsequence int include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + right (idx , idx + 1 , arr , rightMemo); } // Store and return the result return rightMemo [prev][idx] = max (include , right (prev , idx + 1 , arr , rightMemo)); } // Function to find the longest bitonic sequence int LongestBitonicSequence (vector < int >& arr) { int n = arr . size (); int maxLength = 0 ; // Initialize memoization tables vector < vector < int >> leftMemo (n , vector < int > (n , -1)); vector < vector < int >> rightMemo (n , vector < int > (n , -1)); // Iterate over potential peaks in the array for (int i = 1 ; i < n - 1 ; i ++) { // Find the longest decreasing subsequences // on both sides of arr[i] int leftLen = left (i , i - 1 , arr , leftMemo); int rightLen = right (i , i + 1 , arr , rightMemo); // Ensure both left and right subsequences are valid if (leftLen == 0 || rightLen == 0) { leftLen = 0 ; rightLen = 0 ; } // Update maximum bitonic sequence length maxLength = max (maxLength , leftLen + rightLen + 1); } // If no valid bitonic sequence, return 0 return (maxLength < 3) ? 0 : maxLength ; } int main () { vector < int > arr = { 12 , 11 , 40 , 5 , 3 , 1 }; cout << LongestBitonicSequence (arr) << endl ; return 0 ; } Java // Java implementation to find longest Bitonic // subsequence using Recursion and Memoization import java.util.* ; class GfG { // Function to find the longest decreasing subsequence // to the left with memoization static int left (int prev , int idx , ArrayList < Integer > arr , int [][] leftMemo) { if (idx < 0) { return 0 ; } if (leftMemo [prev][idx] != - 1) { return leftMemo [prev][idx] ; } // Check if arr[idx] can be included in // decreasing subsequence int include = 0 ; if (arr . get (idx) < arr . get (prev)) { include = 1 + left (idx , idx - 1 , arr , leftMemo); } // Store and return the result return leftMemo [prev][idx] = Math . max (include , left (prev , idx - 1 , arr , leftMemo)); } // Function to find the longest decreasing subsequence // to the right with memoization static int right (int prev , int idx , ArrayList < Integer > arr , int [][] rightMemo) { if (idx >= arr . size ()) { return 0 ; } if (rightMemo [prev][idx] != - 1) { return rightMemo [prev][idx] ; } // Check if arr[idx] can be included in // decreasing subsequence int include = 0 ; if (arr . get (idx) < arr . get (prev)) { include = 1 + right (idx , idx + 1 , arr , rightMemo); } // Store and return the result return rightMemo [prev][idx] = Math . max (include , right (prev , idx + 1 , arr , rightMemo)); } // Function to find the longest bitonic sequence static int LongestBitonicSequence (ArrayList < Integer > arr) { int n = arr . size (); int maxLength = 0 ; // Initialize memoization tables int [][] leftMemo = new int [n][n] ; int [][] rightMemo = new int [n][n] ; // Fill memoization tables with -1 for (int i = 0 ; i < n ; i ++) { Arrays . fill (leftMemo [i] , - 1); Arrays . fill (rightMemo [i] , - 1); } // Iterate over potential peaks in the array for (int i = 1 ; i < n - 1 ; i ++) { // Find the longest decreasing subsequences // on both sides of arr[i] int leftLen = left (i , i - 1 , arr , leftMemo); int rightLen = right (i , i + 1 , arr , rightMemo); // Ensure both left and right subsequences are valid if (leftLen == 0 || rightLen == 0) { leftLen = 0 ; rightLen = 0 ; } // Update maximum bitonic sequence length maxLength = Math . max (

maxLength , leftLen + rightLen + 1); } // If no valid bitonic sequence, return 0 return (maxLength < 3) ? 0 : maxLength ; } public static void main (String [] args) { ArrayList < Integer > arr = new ArrayList <> (Arrays . asList (12 , 11 , 40 , 5 , 3 , 1)); System . out . println (LongestBitonicSequence (arr)); } } Python # Python implementation to find longest Bitonic # subsequence using Recursion and Memoization # Function to find the longest decreasing subsequence # to the left def left (prev , idx , arr , leftMemo): if idx < 0 : return 0 if leftMemo [prev][idx] != - 1 : return leftMemo [prev][idx] # Check if arr[idx] can be included in # decreasing subsequence include = 0 if arr [idx] < arr [prev]: include = 1 + left (idx , idx - 1 , arr , leftMemo) # Store and return the result leftMemo [prev][idx] = max (include , left (prev , idx - 1 , arr , leftMemo)) return leftMemo [prev][idx] # Function to find the longest decreasing subsequence # to the right def right (prev , idx , arr , rightMemo): if idx >= len (arr): return 0 if rightMemo [prev][idx] != - 1 : return rightMemo [prev][idx] # Check if arr[idx] can be included in decreasing # subsequence include = 0 if arr [idx] < arr [prev]: include = 1 + right (idx , idx + 1 , arr , rightMemo) # Store and return the result rightMemo [prev][idx] = max (include , right (prev , idx + 1 , arr , rightMemo)) return rightMemo [prev][idx] def LongestBitonicSequence (arr): n = len (arr) maxLength = 0 # Initialize memoization tables leftMemo = [[- 1 for i in range (n)] for i in range (n)] rightMemo = [[- 1 for i in range (n)] for i in range (n)] # Iterate over potential peaks in the array for i in range (1 , n - 1): # Find the longest decreasing subsequences # on both sides of arr[i] leftLen = left (i , i - 1 , arr , leftMemo) rightLen = right (i , i + 1 , arr , rightMemo) # Ensure both left and right subsequences are valid if leftLen == 0 or rightLen == 0 : leftLen = 0 rightLen = 0 # Update maximum bitonic sequence length maxLength = max (maxLength , leftLen + rightLen + 1) # If no valid bitonic sequence, return 0 return 0 if maxLength < 3 else maxLength if __name__ == "__main__" : arr = [12 , 11 , 40 , 5 , 3 , 1] print (LongestBitonicSequence (arr)) C# // C# implementation to find longest Bitonic // subsequence using Recursion and Memoization using System ; using System.Collections.Generic ; class GfG { // Function to find the longest decreasing subsequence // to the left with memoization static int Left (int prev , int idx , List < int > arr , int [,] leftMemo) { if (idx < 0) { return 0 ; } if (leftMemo [prev , idx] != - 1) { return leftMemo [prev , idx]; } // Check if arr[idx] can be included in // decreasing subsequence int include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + Left (idx , idx - 1 , arr , leftMemo); } // Store and return the result leftMemo [prev , idx] = Math . Max (include , Left (prev , idx - 1 , arr , leftMemo)); return leftMemo [prev , idx]; } // Function to find the longest decreasing subsequence // to the right with memoization static int Right (int prev , int idx , List < int > arr , int [,] rightMemo) { if (idx >= arr . Count) { return 0 ; } if (rightMemo [prev , idx] != - 1) { return rightMemo [prev , idx]; } // Check if arr[idx] can be included in // decreasing subsequence int include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + Right (idx , idx + 1 , arr , rightMemo); } // Store and return the result rightMemo [prev , idx] = Math . Max (include , Right (prev , idx + 1 , arr , rightMemo)); return rightMemo [prev , idx]; } // Function to find the longest bitonic sequence static int LongestBitonicSequence (List < int > arr) { int n = arr . Count ; int maxLength = 0 ; // Initialize memoization tables int [,] leftMemo = new int [n , n]; int [,] rightMemo = new int [n , n]; // Fill memoization tables with -1 for (int i = 0 ; i < n ; i ++) { for (int j = 0 ; j < n ; j ++) { leftMemo [i , j] = - 1 ; rightMemo [i , j] = - 1 ; } } // Iterate over potential peaks in the array for (int i = 1 ; i < n - 1 ; i ++) { // Find the longest decreasing subsequences // on both sides of arr[i] int leftLen = Left (i , i - 1 , arr , leftMemo); int rightLen = Right (i , i + 1 , arr , rightMemo); // Ensure both left and right subsequences are valid if (leftLen == 0 || rightLen == 0) { leftLen = 0 ; rightLen = 0 ; } // Update maximum bitonic sequence length maxLength = Math . Max (maxLength , leftLen + rightLen + 1); } return (maxLength < 3) ? 0 : maxLength ; } static void Main (string [] args) { List < int > arr = new List < int > { 12 , 11 , 40 , 5 , 3 , 1 }; Console . WriteLine (LongestBitonicSequence (arr)); } } JavaScript // Javascript implementation to find longest Bitonic // subsequence using Recursion and Memoization // Function to find the longest decreasing subsequence // to the left with memoization function left (prev , idx , arr , leftMemo) { if (idx < 0) { return 0 ; } if (leftMemo [prev][idx] !== - 1) { return leftMemo [prev][idx]; } // Check if arr[idx] can be included in decreasing subsequence let include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + left (idx , idx - 1 , arr , leftMemo); } // Store and return the result return leftMemo [prev][idx] = Math . max (include , left (prev , idx - 1 , arr , leftMemo)); } // Function to find the longest decreasing // subsequence to the right with memoization function right (prev , idx , arr , rightMemo) { if (idx >= arr . length) { return 0 ; } if (rightMemo [prev][idx] !== - 1) { return rightMemo [prev][idx]; } // Check if arr[idx] can be included in decreasing subsequence let include = 0 ; if (arr [idx] < arr [prev]) { include = 1 + right (idx , idx + 1 , arr , rightMemo); } // Store and return the result return rightMemo [prev][idx] = Math . max (include , right (prev , idx + 1 , arr , rightMemo)); } // Function to find the longest bitonic sequence function LongestBitonicSequence (arr) { const n = arr . length ; let maxLength = 0 ; // Initialize memoization tables const leftMemo = Array . from ({ length : n }, () => Array (n). fill (- 1)); const rightMemo = Array . from ({ length : n }, () => Array (n). fill (- 1)); // Iterate over potential peaks in the array for (let i = 1 ; i < n - 1 ; i ++) { // Find the longest decreasing subsequences // on both sides of arr[i] let leftLen = left (i , i - 1 , arr , leftMemo); let rightLen = right (i , i + 1 , arr , rightMemo); // Ensure both left and right subsequences are valid if (leftLen === 0 || rightLen === 0) { leftLen = 0 ; rightLen = 0 ; } // Update maximum bitonic sequence length maxLength = Math . max (maxLength , leftLen + rightLen + 1); } // If no valid bitonic sequence, return 0 return maxLength < 3 ? 0 : maxLength ; } const arr = [12 , 11 , 40 , 5 , 3 , 1]; console . log (LongestBitonicSequence (arr)); Output 5 Using Bottom-Up DP (Tabulation) - O(n^2) Time and O(n) Space This approach iteratively builds the solution from smaller subproblems in a bottom-up manner, avoiding recursion. This problem is a variation of standard Longest Increasing Subsequence (LIS) problem . We create two 1D arrays , left and right, of size n. left[i] stores the length of the Longest Increasing Subsequence (LIS) ending at index i. right[i] stores the length of the Longest Decreasing Subsequence (LDS) starting at index i. The dynamic programming relations are as follows: For LIS: If arr[i] > arr[j], then left[i] = max(left[i], left[j] + 1) This means the LIS at i can be extended by the LIS ending at j. For LDS: If arr[i] > arr[j], then right[i] = max(right[i], right[j] + 1) This means the LDS at i can be extended by the LDS starting at j. The final bitonic length for a peak at index i is calculated as: maxLength = max(maxLength, left[i] + right[i] - 1) The subtraction of 1 accounts for the peak element being counted in both LIS and LDS. C++ // C++ implementation to find the longest // bitonic subsequence using tabulation #include <bits/stdc++.h> using namespace std ; int LongestBitonicSequence (vector < int >& arr) { int n = arr . size (); // If there are less than 3 elements, // no bitonic subsequence exists if (n < 3) return 0 ; // Create tables for longest increasing subsequence // (LIS) and longest decreasing subsequence (LDS) vector < int > left (n , 1), right (n , 1); // Fill left table for LIS for (int i = 1 ; i < n ; i ++) { for (int j = 0 ; j < i ; j ++) { // If arr[i] is greater than arr[j], // update LIS value at i if (arr [i] > arr [j]) { left [i] = max (left [i], left [j] + 1); } } } // Fill right table for LDS for (int i = n - 2 ; i >= 0 ; i --) { // Compare each element with subsequent // ones to build LDS for (int j = n - 1 ; j > i ; j --) { // If arr[i] is greater than arr[j], // update LDS value at i if (arr [i] > arr [j]) { right [i] = max (right [i], right [j] + 1); } } } // Calculate the maximum length of bitonic subsequence int maxLength = 0 ; for (int i = 0 ; i < n ; i ++) { // Check if both LIS and LDS are valid // for the current index if (left [i] > 1 && right [i] > 1) { // Update maxLength considering both LIS // and LDS, subtracting 1 for the peak element maxLength = max (maxLength , left [i] + right [i] - 1); } } // If no valid bitonic sequence, return 0 return maxLength < 3 ? 0 : maxLength ; } int main () { vector < int > arr = { 12 , 11 , 40 , 5 , 3 , 1 }; cout << LongestBitonicSequence (arr) << endl ; return 0 ; } Java // Java implementation to find the longest // bitonic subsequence using tabulation import java.util.* ; class GfG { // Function to find the longest bitonic subsequence static int LongestBitonicSequence (ArrayList < Integer > arr) { int n = arr . size (); // If there are less than 3 elements, // no bitonic subsequence exists if (n < 3) return 0 ; // Create ArrayLists for longest increasing subsequence // (LIS) and longest decreasing subsequence (LDS) ArrayList < Integer > left = new ArrayList <> (Collections . nCopies (n , 1)); ArrayList < Integer > right = new ArrayList <> (Collections . nCopies (n , 1)); // Fill left ArrayList for LIS for (int i = 1 ; i < n ; i ++) { for (int j = 0 ; j < i ; j ++) { // If arr[i] is greater than arr[j], // update LIS value at i if (arr . get (i) > arr . get (j)) { left . set (i , Math . max (left . get (i), left . get (j) + 1)); } } } // Fill right ArrayList for LDS for (int i = n - 2 ; i >= 0 ; i --) { // Compare each element with subsequent // ones to build LDS for (int j = n - 1 ; j > i ; j --) { // If arr[i] is greater than arr[j], // update LDS value at i if (arr . get (i) > arr . get (j)) { right . set (i , Math . max (right . get (i), right . get (j) + 1)); } } } // Calculate the maximum length of bitonic subsequence int maxLength = 0 ; for (int i = 0 ; i < n ; i ++) { // Check if both LIS and LDS are valid // for the current index if (left . get (i) > 1 && right . get (i) > 1) { // Update maxLength considering both LIS // and LDS, subtracting 1 for the peak element maxLength = Math . max (maxLength , left . get (i) + right . get (i) - 1); } } // If no valid bitonic sequence, return 0 return maxLength < 3 ? 0 : maxLength ; } public static void main (String [] args) { ArrayList < Integer > arr = new ArrayList <> (Arrays . asList (12 , 11 , 40 , 5 , 3 , 1)); System . out . println (LongestBitonicSequence (arr)); } } Python # Python implementation to find the longest # bitonic subsequence using tabulation def LongestBitonicSequence (arr): n = len (arr) # If there are less than 3 elements, # no bitonic subsequence exists if n < 3 : return 0 # Create lists for longest increasing subsequence # (LIS) and longest decreasing subsequence (LDS) left = [1] * n right = [1] * n # Fill left list for LIS for i in range (1 , n): for j in range (i): # If arr[i] is greater than arr[j], # update LIS value at i if arr [i] > arr [j]: left [i] = max (left [i], left [j] + 1) # Fill right list for LDS for i in range (n - 2 , - 1 , - 1): # Compare each element with subsequent # ones to build LDS for j in range (n - 1 , i , - 1): # If arr[i] is greater than arr[j], # update LDS value at i if arr [i] > arr [j]: right [i] = max (right [i], right [j] + 1)

# Calculate the maximum length of bitonic subsequence maxLength = 0 for i in range (n): # Check if both LIS and LDS are valid # for the current index if left [i] > 1 and right [i] > 1 : # Update maxLength considering both LIS # and LDS, subtracting 1 for the peak element maxLength = max (maxLength , left [i] + right [i] - 1) # If no valid bitonic sequence, return 0 return maxLength if maxLength >= 3 else 0 if __name__ == "__main__" : arr = [12 , 11 , 40 , 5 , 3 , 1] print (LongestBitonicSequence (arr)) C# // C# implementation to find the longest // bitonic subsequence using tabulation using System ; using System.Collections.Generic ; class GfG { static int LongestBitonicSequence (List < int > arr) { int n = arr . Count ; // If there are less than 3 elements, // no bitonic subsequence exists if (n < 3) return 0 ; // Create lists for longest increasing subsequence // (LIS) and longest decreasing subsequence (LDS) List < int > left = new List < int > (new int [n]); List < int > right = new List < int > (new int [n]); // Initialize lists with 1 for (int i = 0 ; i < n ; i ++) { left [i] = 1 ; right [i] = 1 ; } // Fill left list for LIS for (int i = 1 ; i < n ; i ++) { for (int j = 0 ; j < i ; j ++) { // If arr[i] is greater than arr[j], // update LIS value at i if (arr [i] > arr [j]) { left [i] = Math . Max (left [i], left [j] + 1); } } } // Fill right list for LDS for (int i = n - 2 ; i >= 0 ; i --) { // Compare each element with subsequent // ones to build LDS for (int j = n - 1 ; j > i ; j --) { // If arr[i] is greater than arr[j], // update LDS value at i if (arr [i] > arr [j]) { right [i] = Math . Max (right [i], right [j] + 1); } } } // Calculate the maximum length of bitonic subsequence int maxLength = 0 ; for (int i = 0 ; i < n ; i ++) { // Check if both LIS and LDS are valid // for the current index if (left [i] > 1 && right [i] > 1) { // Update maxLength considering both LIS // and LDS, subtracting 1 for the peak element maxLength = Math . Max (maxLength , left [i] + right [i] - 1); } } // If no valid bitonic sequence, return 0 return maxLength < 3 ? 0 : maxLength ; } static void Main () { List < int > arr = new List < int > { 12 , 11 , 40 , 5 , 3 , 1 }; Console . WriteLine (LongestBitonicSequence (arr)); } } JavaScript // Javascript implementation to find the longest // bitonic subsequence using tabulation function LongestBitonicSequence (arr) { const n = arr . length ; // If there are less than 3 elements, // no bitonic subsequence exists if (n < 3) return 0 ; // Create arrays for longest increasing subsequence // (LIS) and longest decreasing subsequence (LDS) const left = Array (n). fill (1); const right = Array (n). fill (1); // Fill left array for LIS for (let i = 1 ; i < n ; i ++) { for (let j = 0 ; j < i ; j ++) { // If arr[i] is greater than arr[j], // update LIS value at i if (arr [i] > arr [j]) { left [i] = Math . max (left [i], left [j] + 1); } } } // Fill right array for LDS for (let i = n - 2 ; i >= 0 ; i --) { // Compare each element with subsequent // ones to build LDS for (let j = n - 1 ; j > i ; j --) { // If arr[i] is greater than arr[j], // update LDS value at i if (arr [i] > arr [j]) { right [i] = Math . max (right [i], right [j] + 1); } } } // Calculate the maximum length of bitonic subsequence let maxLength = 0 ; for (let i = 0 ; i < n ; i ++) { // Check if both LIS and LDS are valid // for the current index if (left [i] > 1 && right [i] > 1) { // Update maxLength considering both LIS // and LDS, subtracting 1 for the peak element maxLength = Math . max (maxLength , left [i] + right [i] - 1); } } // If no valid bitonic sequence, return 0 return maxLength < 3 ? 0 : maxLength ; } const arr = [12 , 11 , 40 , 5 , 3 , 1]; console . log (LongestBitonicSequence (arr)); Output 5 Comment Article Tags: Article Tags: Dynamic Programming DSA Microsoft subsequence LIS + 1 More