Box Stacking Problem - GeeksforGeeks

Source: https://www.geeksforgeeks.org/box-stacking-problem-dp-22/

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 Box Stacking Problem Last Updated : 23 Jul, 2025 Given three arrays height[], width[], and length[] of size n, where height[i], width[i], and length[i] represent the dimensions of a box . The task is to create a stack of boxes that is as tall as possible , but we can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Note: We can rotate a box so that any side functions as its base. It is also allowable to us e multiple instances of the same type of box. The base of the lower box should be strictly larger than that of the new box we're going to place. This is in terms of both length and width , not just in terms of area . So, two boxes with the same base cannot be placed one over the other. Example: Input: height[] = [4, 1, 4, 10], width[] = [6, 2, 5, 12], length[] = [7, 3, 6, 32] Output: 60 Explanation: One way of placing the boxes is as follows in the bottom to top manner: (Denoting the boxes in (l, w, h) manner) (12, 32, 10) (10, 12, 32) (6, 7, 4) (5, 6, 4) (4, 5, 6) (2, 3, 1) (1, 2, 3) Hence, the total height of this stack is 10 + 32 + 4 + 4 + 6 + 1 + 3 = 60. No other combination of boxes produces a height greater than this. Input: height[] = [1, 4, 3], width[] = [2, 5, 4], length[] = [3, 6, 1] Output: 15 Explanation: One way of placing the boxes is as follows in the bottom to top manner: (Denoting the boxes in (l, w, h) manner) (5, 6, 4) (4, 5, 6) (3, 4, 1), (2, 3, 1) (1, 2, 3). Hence, the total height of this stack is 4 + 6 + 1 + 1 + 3 = 15 No other combination of boxes produces a height greater than this. Try it on GfG Practice Main Idea The Box Stacking problem is a variation of LIS problem . The main idea is to maximize the height of the stack by considering all possible orientations of the boxes and find the optimal stacking order . For each box, we generate all six possible rotations by treating each dimension as the height once, and the remaining two dimensions as the base dimensions (width and depth) . By doing this, we account for all possible orientations and allow multiple instances of the same box type in different orientations. We then sort these box rotations by their length and breath in descending order , ensuring that smaller boxes are only placed on larger ones . We compute the maximum stack height for each box as the base by iterating through all prior boxes and checking if they can be stacked. This approach ensures that we evaluate all valid stacking combinations efficiently and find the maximum height achievable. Table of Content Using Recursion - O(n^n) Time and O(n) Space Using Top-Down DP (Memoization) - O(n^2) Time and O(n) Space Using Bottom-Up DP (Tabulation) - O(n^2) Time and O(n) Space Using Recursion - O(n^n) Time and O(n) Space The idea is to recursively compute the maximum stack height starting from each box in all its possible orientations. For a given box i with a particular orientation, the recursive relation is based on two conditions: We check if the current box i can be placed on top of any previously considered box j , meaning the base of box i must be strictly smaller than the base of box j. We compute the maximum stack height by choosing the best possible prior box to place under box i. The r ecurrence relation can be written as: maxHeight[i] = max■(height[i] + maxHeight[j] for all boxes j where base of box i > base of box j) Here, maxHeight[i] represents the maximum height of the stack starting with box i in its current orientation. The recurrence works by checking all possible box placements and taking the maximum value among them, ensuring that the box stack grows in height as much as possible. The base case is that a box by itself contributes a stack height of height[i], i.e., maxHeight[i] = height[i], when no boxes can be placed under it. C++ // C++ program to implement // box stacking problem #include <bits/stdc++.h> using namespace std ; // Function to find the maximum height // with box i as base. int maxHeightRecur (int i , vector < vector < int >> & boxes) { int ans = boxes [i][2]; // Check all the next boxes for (int j = i + 1 ; j < boxes . size (); j ++) { // If size of box j is

less than // size of box i. if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { ans = max (ans , boxes [i][2] + maxHeightRecur (j , boxes)); } } return ans ; } int maxHeight (vector < int > & height , vector < int > & width , vector < int > & length) { int n = height . size (); // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. vector < vector < int >> boxes ; for (int i = 0 ; i < n ; i ++) { int a = height [i], b = width [i], c = length [i]; boxes . push_back ({ a , b , c }); boxes . push_back ({ a , c , b }); boxes . push_back ({ b , a , c }); boxes . push_back ({ b , c , a }); boxes . push_back ({ c , a , b }); boxes . push_back ({ c , b , a }); } // Sort the boxes in descending // order of length and width. sort (boxes . begin (), boxes . end (), greater < vector < int >> ()); int ans = 0 ; // Check for all boxes starting as base. for (int i = 0 ; i < boxes . size (); i ++) { ans = max (ans , maxHeightRecur (i , boxes)); } return ans ; } int main (){ vector < int > height = { 4 , 1 , 4 , 10 }; vector < int > width = { 6 , 2 , 5 , 12 }; vector < int > length = { 7 , 3 , 6 , 32 }; cout << maxHeight (height , width , length); return 0 ; } Java // Java program to implement // box stacking problem import java.util.* ; class GfG { // Function to find the maximum height // with box i as base. static int maxHeightRecur (int i , int [][] boxes) { int ans = boxes [i][2] ; // Check all the next boxes for (int j = i + 1 ; j < boxes . length ; j ++) { // If size of box j is less than // size of box i. if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { ans = Math . max (ans , boxes [i][2] + maxHeightRecur (j , boxes)); } } return ans ; } static int maxHeight (int [] height , int [] width , int [] length) { int n = height . length ; // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. int [][] boxes = new int [n * 6][3] ; int index = 0 ; for (int i = 0 ; i < n ; i ++) { int a = height [i] , b = width [i] , c = length [i] ; boxes [index ++] = new int [] { a , b , c }; boxes [index ++] = new int [] { a , c , b }; boxes [index ++] = new int [] { b , a , c }; boxes [index ++] = new int [] { b , c , a }; boxes [index ++] = new int [] { c , a , b }; boxes [index ++] = new int [] { c , b , a }; } // Sort the boxes in descending // order of length and width. Arrays . sort (boxes , (box1 , box2) -> { if (box1 [0] == box2 [0]) { return Integer . compare (box1 [1] , box2 [1]); } return Integer . compare (box2 [0] , box1 [0]); }); int ans = 0 ; // Check for all boxes starting as base. for (int i = 0 ; i < boxes . length ; i ++) { ans = Math . max (ans , maxHeightRecur (i , boxes)); } return ans ; } public static void main (String [] args) { int [] height = { 4 , 1 , 4 , 10 }; int [] width = { 6 , 2 , 5 , 12 }; int [] length = { 7 , 3 , 6 , 32 }; System . out . println (maxHeight (height , width , length)); } } Python # Python program to implement # box stacking problem # function to find the maximum height # with box i as base. def maxHeightRecur (i , boxes): ans = boxes [i][2] # Check all the next boxes for j in range (i + 1 , len (boxes)): # If size of box j is less than # size of box i. if boxes [i][0] > boxes [j][0] and boxes [i][1] > boxes [j][1]: ans = max (ans , boxes [i][2] + maxHeightRecur (j , boxes)) return ans def maxHeight (height , width , length): n = len (height) # Create a 2d array to store all # orientations of boxes in (l, b, h) # manner. boxes = [] for i in range (n): a , b , c = height [i], width [i], length [i] boxes . append ([a , b , c]) boxes . append ([a , c , b]) boxes . append ([b , a , c]) boxes . append ([b , c , a]) boxes . append ([c , a , b]) boxes . append ([c , b , a]) # Sort the boxes in descending # order of length and width. boxes . sort (key = lambda x : (x [0], x [1]), reverse = True) ans = 0 # Check for all boxes starting as base. for i in range (len (boxes)): ans = max (ans , maxHeightRecur (i , boxes)) return ans if __name__ == "__main__" : height = [4 , 1 , 4 , 10] width = [6 , 2 , 5 , 12] length = [7 , 3 , 6 , 32] print (maxHeight (height , width , length)) C# // C# program to implement // box stacking problem using System ; using System.Collections.Generic ; class GfG { // Function to find the maximum height // with box i as base. static int maxHeightRecur (int i , List < List < int >> boxes) { int ans = boxes [i][2]; // Check all the next boxes for (int j = i + 1 ; j < boxes . Count ; j ++) { // If size of box j is less than // size of box i. if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { ans = Math . Max (ans , boxes [i][2] + maxHeightRecur (j , boxes)); } } return ans ; } static int maxHeight (int [] height , int [] width , int [] length) { int n = height . Length ; // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. List < List < int >> boxes = new List < List < int >> (); for (int i = 0 ; i < n ; i ++) { int a = height [i], b = width [i], c = length [i]; boxes . Add (new List < int > { a , b , c }); boxes . Add (new List < int > { a , c , b }); boxes . Add (new List < int > { b , a , c }); boxes . Add (new List < int > { b , c , a }); boxes . Add (new List < int > { c , a , b }); boxes . Add (new List < int > { c , b , a }); } // Sort the boxes in descending // order of length and width. boxes . Sort ((x , y) => y [0] != x [0] ? y [0]. CompareTo (x [0]) : y [1]. CompareTo (x [1])); int ans = 0 ; // Check for all boxes starting as base. for (int i = 0 ; i < boxes . Count ; i ++) { ans = Math . Max (ans , maxHeightRecur (i , boxes)); } return ans ; } static void Main () { int [] height = { 4 , 1 , 4 , 10 }; int [] width = { 6 , 2 , 5 , 12 }; int [] length = { 7 , 3 , 6 , 32 }; Console . WriteLine (maxHeight (height , width , length)); } } JavaScript // JavaScript program to implement // box stacking problem // Function to find the maximum height // with box i as base. function maxHeightRecur (i , boxes) { let ans = boxes [i][2]; // Check all the next boxes for (let j = i + 1 ; j < boxes . length ; j ++) { // If size of box j is less than //

size of box i. if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { ans = Math . max (ans , boxes [i][2] + maxHeightRecur (j , boxes)); } } return ans ; } function maxHeight (height , width , length) { let n = height . length ; // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. let boxes = []; for (let i = 0 ; i < n ; i ++) { let a = height [i], b = width [i], c = length [i]; boxes . push ([a , b , c]); boxes . push ([a , c , b]); boxes . push ([b , a , c]); boxes . push ([b , c , a]); boxes . push ([c , a , b]); boxes . push ([c , b , a]); } // Sort the boxes in descending // order of length and width. boxes . sort ((x , y) => y [0] !== x [0] ? y [0] - x [0] : y [1] - x [1]); let ans = 0 ; // Check for all boxes starting as base. for (let i = 0 ; i < boxes . length ; i ++) { ans = Math . max (ans , maxHeightRecur (i , boxes)); } return ans ; } let height = [4 , 1 , 4 , 10]; let width = [6 , 2 , 5 , 12]; let length = [7 , 3 , 6 , 32]; console . log (maxHeight (height , width , length)); Output 60 Using Top-Down DP (Memoization) - O(n^2) Time and O(n) Space If we notice carefully, we can observe that the above recursive solution holds the following two properties of Dynamic Programming : 1. Optimal Substructure : Maximum height of box stack at index i , i.e., maxHeight(i) , depends on the optimal solutions of the subproblems maxHeight(j) for all j > i and base of j is smaller than i. By comparing these optimal substructures, we can efficiently calculate the maximum height of box stack at index i. 2. Overlapping Subproblems : While applying a recursive approach in this problem, we notice that certain subproblems are computed multiple times. There is only one parameter: i that changes in the recursive solution. So we create a 1D a rray of size 6*n for memorization (since there are 6 possible orientations for each box). Therefore, memo[i] stores the result for the i-th box in the sorted list of orientations. 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++ // C++ program to implement // box stacking problem #include <bits/stdc++.h> using namespace std ; // Function to find the maximum height // with box i as base. int maxHeightRecur (int i , vector < vector < int >> & boxes , vector < int > & memo) { // If value is memoized if (memo [i] != -1) return memo [i]; int ans = boxes [i][2]; // Check all the next boxes for (int j = i + 1 ; j < boxes . size (); j ++) { // If size of box j is less than // size of box i. if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]){ ans = max (ans , boxes [i][2] + maxHeightRecur (j , boxes , memo)); } } return memo [i] = ans ; } int maxHeight (vector < int > & height , vector < int > & width , vector < int > & length) { int n = height . size (); // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. vector < vector < int >> boxes ; for (int i = 0 ; i < n ; i ++) { int a = height [i], b = width [i], c = length [i]; boxes . push_back ({ a , b , c }); boxes . push_back ({ a , c , b }); boxes . push_back ({ b , a , c }); boxes . push_back ({ b , c , a }); boxes . push_back ({ c , a , b }); boxes . push_back ({ c , b , a }); } // Sort the boxes in descending // order of length and width. sort (boxes . begin (), boxes . end (), greater < vector < int >> ()); vector < int > memo (boxes . size (), -1); int ans = 0 ; // Check for all boxes starting as base. for (int i = 0 ; i < boxes . size (); i ++) { ans = max (ans , maxHeightRecur (i , boxes , memo)); } return ans ; } int main () { vector < int > height = { 4 , 1 , 4 , 10 }; vector < int > width = { 6 , 2 , 5 , 12 }; vector < int > length = { 7 , 3 , 6 , 32 }; cout << maxHeight (height , width , length); return 0 ; } Java // Java program to implement // box stacking problem import java.util.* ; class GfG { // Function to find the maximum height // with box i as base. static int maxHeightRecur (int i , int [][] boxes , int [] memo) { // If value is memoized if (memo [i] != - 1) return memo [i] ; int ans = boxes [i][2] ; // Check all the next boxes for (int j = i + 1 ; j < boxes . length ; j ++) { // If size of box j is less than // size of box i. if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { ans = Math . max (ans , boxes [i][2] + maxHeightRecur (j , boxes , memo)); } } return memo [i] = ans ; } static int maxHeight (int [] height , int [] width , int [] length) { int n = height . length ; // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. List < int []> boxes = new ArrayList <> (); for (int i = 0 ; i < n ; i ++) { int a = height [i] , b = width [i] , c = length [i] ; boxes . add (new int [] { a , b , c }); boxes . add (new int [] { a , c , b }); boxes . add (new int [] { b , a , c }); boxes . add (new int [] { b , c , a }); boxes . add (new int [] { c , a , b }); boxes . add (new int [] { c , b , a }); } // Sort the boxes in descending // order of length and width. boxes . sort ((x , y) -> y [0] != x [0] ? y [0] - x [0] : y [1] - x [1]); int [] memo = new int [boxes . size ()] ; Arrays . fill (memo , - 1); int ans = 0 ; // Check for all boxes starting as base. for (int i = 0 ; i < boxes . size (); i ++) { ans = Math . max (ans , maxHeightRecur (i , boxes . toArray (new int [0][]), memo)); } return ans ; } public static void main (String [] args) { int [] height = { 4 , 1 , 4 , 10 }; int [] width = { 6 , 2 , 5 , 12 }; int [] length = { 7 , 3 , 6 , 32 }; System . out . println (maxHeight (height , width , length)); } } Python # Python program to implement # box stacking problem def maxHeightRecur (i , boxes , memo): # If value is memoized if memo [i] != - 1 : return memo [i] ans = boxes [i][2] # Check all the next boxes for j in range (i + 1 , len (boxes)): # If size of box j is less than # size of box i. if boxes [i][0] > boxes [j][0] and boxes [i][1] > boxes [j

][1]: ans = max (ans , boxes [i][2] + maxHeightRecur (j , boxes , memo)) memo [i] = ans return ans def maxHeight (height , width , length): n = len (height) # Create a 2d array to store all # orientations of boxes in (l, b, h) # manner. boxes = [] for i in range (n): a , b , c = height [i], width [i], length [i] boxes . append ([a , b , c]) boxes . append ([a , c , b]) boxes . append ([b , a , c]) boxes . append ([b , c , a]) boxes . append ([c , a , b]) boxes . append ([c , b , a]) # Sort the boxes in descending # order of length and width. boxes . sort (key = lambda x : (x [0], x [1]), reverse = True) memo = [- 1] * len (boxes) ans = 0 # Check for all boxes starting as base. for i in range (len (boxes)): ans = max (ans , maxHeightRecur (i , boxes , memo)) return ans if __name__ == "__main__" : height = [4 , 1 , 4 , 10] width = [6 , 2 , 5 , 12] length = [7 , 3 , 6 , 32] print (maxHeight (height , width , length)) C# // C# program to implement // box stacking problem using System ; using System.Collections.Generic ; class GfG { // Function to find the maximum height // with box i as base. static int maxHeightRecur (int i , List < List < int >> boxes , int [] memo) { // If value is memoized if (memo [i] != - 1) return memo [i]; int ans = boxes [i][2]; // Check all the next boxes for (int j = i + 1 ; j < boxes . Count ; j ++) { // If size of box j is less than // size of box i. if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { ans = Math . Max (ans , boxes [i][2] + maxHeightRecur (j , boxes , memo)); } } return memo [i] = ans ; } static int maxHeight (int [] height , int [] width , int [] length) { int n = height . Length ; // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. List < List < int >> boxes = new List < List < int >> (); for (int i = 0 ; i < n ; i ++) { int a = height [i], b = width [i], c = length [i]; boxes . Add (new List < int > { a , b , c }); boxes . Add (new List < int > { a , c , b }); boxes . Add (new List < int > { b , a , c }); boxes . Add (new List < int > { b , c , a }); boxes . Add (new List < int > { c , a , b }); boxes . Add (new List < int > { c , b , a }); } // Sort the boxes in descending // order of length and width. boxes . Sort ((x , y) => y [0] != x [0] ? y [0]. CompareTo (x [0]) : y [1]. CompareTo (x [1])); int [] memo = new int [boxes . Count]; Array . Fill (memo , - 1); int ans = 0 ; // Check for all boxes starting as base. for (int i = 0 ; i < boxes . Count ; i ++) { ans = Math . Max (ans , maxHeightRecur (i , boxes , memo)); } return ans ; } static void Main () { int [] height = { 4 , 1 , 4 , 10 }; int [] width = { 6 , 2 , 5 , 12 }; int [] length = { 7 , 3 , 6 , 32 }; Console . WriteLine (maxHeight (height , width , length)); } } JavaScript // JavaScript program to implement // box stacking problem // Function to find the maximum height // with box i as base. function maxHeightRecur (i , boxes , memo) { // If value is memoized if (memo [i] !== - 1) return memo [i]; let ans = boxes [i][2]; // Check all the next boxes for (let j = i + 1 ; j < boxes . length ; j ++) { // If size of box j is less than // size of box i. if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { ans = Math . max (ans , boxes [i][2] + maxHeightRecur (j , boxes , memo)); } } memo [i] = ans ; return ans ; } function maxHeight (height , width , length) { let n = height . length ; // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. let boxes = []; for (let i = 0 ; i < n ; i ++) { let a = height [i], b = width [i], c = length [i]; boxes . push ([a , b , c]); boxes . push ([a , c , b]); boxes . push ([b , a , c]); boxes . push ([b , c , a]); boxes . push ([c , a , b]); boxes . push ([c , b , a]); } // Sort the boxes in descending // order of length and width. boxes . sort ((x , y) => y [0] !== x [0] ? y [0] - x [0] : y [1] - x [1]); let memo = new Array (boxes . length). fill (- 1); let ans = 0 ; // Check for all boxes starting as base. for (let i = 0 ; i < boxes . length ; i ++) { ans = Math . max (ans , maxHeightRecur (i , boxes , memo)); } return ans ; } let height = [4 , 1 , 4 , 10]; let width = [6 , 2 , 5 , 12]; let length = [7 , 3 , 6 , 32]; console . log (maxHeight (height , width , length)); Output 60 Using Bottom-Up DP (Tabulation) - O(n^2) Time and O(n) Space The idea is to fill the DP table from bottom to up. The table is filled in an iterative manner from i = n-1 to i = 0 . For each box i, The dynamic programming relation is as follows: set dp[i] = height[i] For j > i and base of j is smaller than base of i, set dp[i] = max(dp[i], height[i] + dp[j]). C++ // C++ program to implement // box stacking problem #include <bits/stdc++.h> using namespace std ; int maxHeight (vector < int > & height , vector < int > & width , vector < int > & length) { int n = height . size (); // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. vector < vector < int >> boxes ; for (int i = 0 ; i < n ; i ++) { int a = height [i], b = width [i], c = length [i]; boxes . push_back ({ a , b , c }); boxes . push_back ({ a , c , b }); boxes . push_back ({ b , a , c }); boxes . push_back ({ b , c , a }); boxes . push_back ({ c , a , b }); boxes . push_back ({ c , b , a }); } // Sort the boxes in descending // order of length and width. sort (boxes . begin (), boxes . end (), greater < vector < int >> ()); vector < int > dp (boxes . size ()); int ans = 0 ; // Check for all boxes starting as base. for (int i = boxes . size () - 1 ; i >= 0 ; i --) { dp [i] = boxes [i][2]; for (int j = i + 1 ; j < boxes . size (); j ++) { if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { dp [i] = max (dp [i], boxes [i][2] + dp [j]); } } ans = max (ans , dp [i]); } return ans ; } int main () { vector < int > height = { 4 , 1 , 4 , 10 }; vector < int > width = { 6 , 2 , 5 , 12 }; vector < int > length = { 7 , 3 , 6 , 32 }; cout << maxHeight (height , width , length); return 0 ; } Java // Java program to implement // box stacking problem import java.util.* ; class

GfG { // Function to find the maximum height // with box i as base. static int maxHeight (int [] height , int [] width , int [] length) { int n = height . length ; // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. List < int []> boxes = new ArrayList <> (); for (int i = 0 ; i < n ; i ++) { int a = height [i] , b = width [i] , c = length [i] ; boxes . add (new int [] { a , b , c }); boxes . add (new int [] { a , c , b }); boxes . add (new int [] { b , a , c }); boxes . add (new int [] { b , c , a }); boxes . add (new int [] { c , a , b }); boxes . add (new int [] { c , b , a }); } // Sort the boxes in descending // order of length and width. boxes . sort ((x , y) -> y [0] != x [0] ? y [0] - x [0] : y [1] - x [1]); int [] dp = new int [boxes . size ()] ; int ans = 0 ; // Check for all boxes starting as base. for (int i = boxes . size () - 1 ; i >= 0 ; i --) { dp [i] = boxes . get (i) [2] ; for (int j = i + 1 ; j < boxes . size (); j ++) { if (boxes . get (i) [0] > boxes . get (j) [0] && boxes . get (i) [1] > boxes . get (j) [1]) { dp [i] = Math . max (dp [i] , boxes . get (i) [2] + dp [j]); } } ans = Math . max (ans , dp [i]); } return ans ; } public static void main (String [] args) { int [] height = { 4 , 1 , 4 , 10 }; int [] width = { 6 , 2 , 5 , 12 }; int [] length = { 7 , 3 , 6 , 32 }; System . out . println (maxHeight (height , width , length)); } } Python # Python program to implement # box stacking problem def maxHeight (height , width , length): n = len (height) # Create a 2d array to store all # orientations of boxes in (l, b, h) # manner. boxes = [] for i in range (n): a , b , c = height [i], width [i], length [i] boxes . append ([a , b , c]) boxes . append ([a , c , b]) boxes . append ([b , a , c]) boxes . append ([b , c , a]) boxes . append ([c , a , b]) boxes . append ([c , b , a]) # Sort the boxes in descending # order of length and width. boxes . sort (key = lambda x : (x [0], x [1]), reverse = True) dp = [0] * len (boxes) ans = 0 # Check for all boxes starting as base. for i in range (len (boxes) - 1 , - 1 , - 1): dp [i] = boxes [i][2] for j in range (i + 1 , len (boxes)): if boxes [i][0] > boxes [j][0] and boxes [i][1] > boxes [j][1]: dp [i] = max (dp [i], boxes [i][2] + dp [j]) ans = max (ans , dp [i]) return ans if __name__ == "__main__" : height = [4 , 1 , 4 , 10] width = [6 , 2 , 5 , 12] length = [7 , 3 , 6 , 32] print (maxHeight (height , width , length)) C# // C# program to implement // box stacking problem using System ; using System.Collections.Generic ; class GfG { // Function to find the maximum height // with box i as base. static int maxHeight (int [] height , int [] width , int [] length) { int n = height . Length ; // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. List < int [] > boxes = new List < int [] > (); for (int i = 0 ; i < n ; i ++) { int a = height [i], b = width [i], c = length [i]; boxes . Add (new int [] { a , b , c }); boxes . Add (new int [] { a , c , b }); boxes . Add (new int [] { b , a , c }); boxes . Add (new int [] { b , c , a }); boxes . Add (new int [] { c , a , b }); boxes . Add (new int [] { c , b , a }); } // Sort the boxes in descending // order of length and width. boxes . Sort ((x , y) => y [0] != x [0] ? y [0]. CompareTo (x [0]) : y [1]. CompareTo (x [1])); int [] dp = new int [boxes . Count]; int ans = 0 ; // Check for all boxes starting as base. for (int i = boxes . Count - 1 ; i >= 0 ; i --) { dp [i] = boxes [i][2]; for (int j = i + 1 ; j < boxes . Count ; j ++) { if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { dp [i] = Math . Max (dp [i], boxes [i][2] + dp [j]); } } ans = Math . Max (ans , dp [i]); } return ans ; } static void Main () { int [] height = { 4 , 1 , 4 , 10 }; int [] width = { 6 , 2 , 5 , 12 }; int [] length = { 7 , 3 , 6 , 32 }; Console . WriteLine (maxHeight (height , width , length)); } } JavaScript // JavaScript program to implement // box stacking problem // Function to find the maximum height // with box i as base. function maxHeight (height , width , length) { let n = height . length ; // Create a 2d array to store all // orientations of boxes in (l, b, h) // manner. let boxes = []; for (let i = 0 ; i < n ; i ++) { let a = height [i], b = width [i], c = length [i]; boxes . push ([a , b , c]); boxes . push ([a , c , b]); boxes . push ([b , a , c]); boxes . push ([b , c , a]); boxes . push ([c , a , b]); boxes . push ([c , b , a]); } // Sort the boxes in descending // order of length and width. boxes . sort ((x , y) => y [0] !== x [0] ? y [0] - x [0] : y [1] - x [1]); let dp = new Array (boxes . length). fill (0); let ans = 0 ; // Check for all boxes starting as base. for (let i = boxes . length - 1 ; i >= 0 ; i --) { dp [i] = boxes [i][2]; for (let j = i + 1 ; j < boxes . length ; j ++) { if (boxes [i][0] > boxes [j][0] && boxes [i][1] > boxes [j][1]) { dp [i] = Math . max (dp [i], boxes [i][2] + dp [j]); } } ans = Math . max (ans , dp [i]); } return ans ; } let height = [4 , 1 , 4 , 10]; let width = [6 , 2 , 5 , 12]; let length = [7 , 3 , 6 , 32]; console . log (maxHeight (height , width , length)); Output 60 Comment Article Tags: Article Tags: Dynamic Programming DSA Microsoft Amazon Codenation + 1 More