0/1 Knapsack Problem - GeeksforGeeks

Source: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

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 0/1 Knapsack Problem Last Updated : 29 Jan, 2026 Given two arrays, profit[] and weight[] , where each element represents the profit and weight of an item respectively, also given an integer W representing the maximum capacity of the knapsack (the total weight it can hold). Put the items into the knapsack such that the sum of profits associated with them is the maximum possible, without exceeding the capacity W. Note: We can either include an item completely or exclude it entirely - we cannot include a fraction of an item. Examples: Input: W = 4, profit[] = [1, 2, 3], weight[] = [4, 5, 1] Output: 3 Explanation: There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4. Input: W = 3, profit[] = [1, 2, 3], weight[] = [4, 5, 6] Output: 0 Explanation: All the item weights are greater than the knapsack capacity. Try it on GfG Practice Table of Content [Naive Approach] Using Recursion O(2^n) Time and O(n) Space [Better Approach 1] Using Top-Down DP (Memoization)- O(n x W) Time and Space [Better Approach 2] Using Bottom-Up DP (Tabulation) - O(n x W) Time and Space [Expected Approach] Using Bottom-Up DP (Space-Optimized) - O(n x W) Time and O(W) Space [Naive Approach] Using Recursion O(2^n) Time and O(n) Space The idea is to use recursion to explore all possible combinations of items. For each item, we have two choices :we can either pick it or not pick it - depending on whether its weight allows it to fit in the knapsack. Case 1: Pick the item-If the item’s weight is less than or equal to the remaining capacity of the knapsack, we include it. Case 2: Do not pick the item-We simply skip the current item and move to the next one, keeping the remaining capacity the same. We take the maximum of these two cases for every item to ensure that the total profit is maximized. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends // Returns the maximum value that // can be put in a knapsack of capacity W int knapsackRec (int W , vector < int > & val , vector < int > & wt , int n) { // Base Case if (n == 0 || W == 0) return 0 ; int pick = 0 ; // Pick nth item if it does not exceed the capacity of knapsack if (wt [n - 1] <= W) pick = val [n - 1] + knapsackRec (W - wt [n - 1], val , wt , n - 1); // Don't pick the nth item int notPick = knapsackRec (W , val , wt , n - 1); return max (pick , notPick); } int knapsack (int W , vector < int > & val , vector < int > & wt) { int n = val . size (); return knapsackRec (W , val , wt , n); } //Driver Code Starts int main () { vector < int > val = { 1 , 2 , 3 }; vector < int > wt = { 4 , 5 , 1 }; int W = 4 ; cout << knapsack (W , val , wt) << endl ; return 0 ; } //Driver Code Ends Java class GfG { // Returns the maximum value that // can be put in a knapsack of capacity W static int knapsackRec (int W , int [] val , int [] wt , int n) { // Base Case if (n == 0 || W == 0) return 0 ; int pick = 0 ; // Pick nth item if it does not exceed the capacity of knapsack if (wt [n - 1] <= W) pick = val [n - 1] + knapsackRec (W - wt [n - 1] , val , wt , n - 1); // Don't pick the nth item int notPick = knapsackRec (W , val , wt , n - 1); return Math . max (pick , notPick); } static int knapsack (int W , int [] val , int [] wt) { int n = val . length ; return knapsackRec (W , val , wt , n); } public static void main (String [] args) { //Driver Code Starts int [] val = { 1 , 2 , 3 }; int [] wt = { 4 , 5 , 1 }; int W = 4 ; System . out . println (knapsack (W , val , wt)); } } //Driver Code Ends Python # Returns the maximum value that # can be put in a knapsack of capacity W def knapsackRec (W , val , wt , n): # Base Case if n == 0 or W == 0 : return 0 pick = 0 # Pick nth item if it does not exceed the capacity of knapsack if wt [n - 1] <= W : pick = val [n - 1] + knapsackRec (W - wt [n - 1], val , wt , n - 1) # Don't pick the nth item notPick = knapsackRec (W , val , wt , n - 1) return max (pick , notPick) def knapsack (W , val , wt): n = len (val) return knapsackRec (W , val , wt , n) if

__name__ == "__main__" : // Driver Code Starts val = [1 , 2 , 3] wt = [4 , 5 , 1] W = 4 print (knapsack (W , val , wt)) // Driver Code Ends C# //Driver Code Starts using System ; class GfG { //Driver Code Ends // Returns the maximum value that // can be put in a knapsack of capacity W static int knapsackRec (int W , int [] val , int [] wt , int n) { // Base Case if (n == 0 || W == 0) return 0 ; int pick = 0 ; // Pick nth item if it does not exceed the capacity of knapsack if (wt [n - 1] <= W) pick = val [n - 1] + knapsackRec (W - wt [n - 1], val , wt , n - 1); // Don't pick the nth item int notPick = knapsackRec (W , val , wt , n - 1); return Math . Max (pick , notPick); } static int knapsack (int W , int [] val , int [] wt) { int n = val . Length ; return knapsackRec (W , val , wt , n); } //Driver Code Starts static void Main () { int [] val = { 1 , 2 , 3 }; int [] wt = { 4 , 5 , 1 }; int W = 4 ; Console . WriteLine (knapsack (W , val , wt)); } } //Driver Code Ends JavaScript // Returns the maximum value that // can be put in a knapsack of capacity W function knapsackRec (W , val , wt , n) { // Base Case if (n === 0 || W === 0) return 0 ; let pick = 0 ; // Pick nth item if it does not exceed the capacity of knapsack if (wt [n - 1] <= W) pick = val [n - 1] + knapsackRec (W - wt [n - 1], val , wt , n - 1); // Don't pick the nth item let notPick = knapsackRec (W , val , wt , n - 1); return Math . max (pick , notPick); } function knapsack (W , val , wt) { let n = val . length ; return knapsackRec (W , val , wt , n); } // Driver Code //Driver Code Starts let val = [1 , 2 , 3]; let wt = [4 , 5 , 1]; let W = 4 ; console . log (knapsack (W , val , wt)); //Driver Code Ends Output 3 [Better Approach - 1] Using Top-Down DP (Memoization)- O(n x W) Time and O(n x W) Space Note: The above function using recursion computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. As there are repetitions of the same subproblem again and again we can implement the following idea to solve the problem. If we get a subproblem the first time, we can solve this problem by creating a 2-D array that can store a particular state. Now if we come across the same state again instead of calculating it i again we can directly return its result stored in the table in constant time. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends // Returns the maximum value that // can be put in a knapsack of capacity W int knapsackRec (int W , vector < int > & val , vector < int > & wt , int n , vector < vector < int >> & memo) { // Base Case if (n == 0 || W == 0) return 0 ; // Check if we have previously calculated the same subproblem if (memo [n][W] != -1) return memo [n][W]; int pick = 0 ; // Pick nth item if it does not exceed the capacity of knapsack if (wt [n - 1] <= W) pick = val [n - 1] + knapsackRec (W - wt [n - 1], val , wt , n - 1 , memo); // Don't pick the nth item int notPick = knapsackRec (W , val , wt , n - 1 , memo); // Store the result in memo[n][W] and return it return memo [n][W] = max (pick , notPick); } int knapsack (int W , vector < int > & val , vector < int > & wt) { int n = val . size (); // Memoization table to store the results vector < vector < int >> memo (n + 1 , vector < int > (W + 1 , -1)); return knapsackRec (W , val , wt , n , memo); } //Driver Code Starts int main () { vector < int > val = { 1 , 2 , 3 }; vector < int > wt = { 4 , 5 , 1 }; int W = 4 ; cout << knapsack (W , val , wt) << endl ; return 0 ; } //Driver Code Ends Java class GfG { // Returns the maximum value that // can be put in a knapsack of capacity W static int knapsackRec (int W , int [] val , int [] wt , int n , int [][] memo) { // Base Case if (n == 0 || W == 0) return 0 ; // Check if we have previously calculated the same subproblem if (memo [n][W] != - 1) return memo [n][W] ; int pick = 0 ; // Pick nth item if it does not exceed the capacity of knapsack if (wt [n - 1] <= W) pick = val [n - 1] + knapsackRec (W - wt [n - 1] , val , wt , n - 1 , memo); // Don't pick the nth item int notPick = knapsackRec (W , val , wt , n - 1 , memo); // Store the result in memo[n][W] and return it return memo [n][W] = Math . max (pick , notPick); } static int knapsack (int W , int [] val , int [] wt) { int n = val . length ; // Memoization table to store the results int [][] memo = new int [n + 1][W + 1] ; // Initialize memoization table with -1 for (int i = 0 ; i <= n ; i ++) { for (int j = 0 ; j <= W ; j ++) memo [i][j] = - 1 ; } return knapsackRec (W , val , wt , n , memo); } public static void main (String [] args) { //Driver Code Starts int [] val = { 1 , 2 , 3 }; int [] wt = { 4 , 5 , 1 }; int W = 4 ; System . out . println (knapsack (W , val , wt)); } } //Driver Code Ends Python # Returns the maximum value that # can be put in a knapsack of capacity W def knapsackRec (W , val , wt , n , memo): # Base Case if n == 0 or W == 0 : return 0 # Check if we have previously calculated the same subproblem if memo [n][W] != - 1 : return memo [n][W] pick = 0 # Pick nth item if it does not exceed the capacity of knapsack if wt [n - 1] <= W : pick = val [n - 1] + knapsackRec (W - wt [n - 1], val , wt , n - 1 , memo) # Don't pick the nth item notPick = knapsackRec (W , val , wt , n - 1 , memo) # Store the result in memo[n][W] and return it memo [n][W] = max (pick , notPick) return memo [n][W] def knapsack (W , val , wt): n = len (val) # Memoization table to store the results memo = [[- 1] * (W + 1) for _ in range (n + 1)] return knapsackRec (W , val , wt , n , memo) if __name__ == "__main__" : #Driver Code Starts val = [1 , 2 , 3] wt = [4 , 5 , 1] W = 4 print (knapsack (W , val , wt)) #Driver Code Ends C# //Driver Code Starts using System ; class GfG { //Driver Code Ends // Returns the maximum value that // can be put in a knapsack of capacity W static int knapsackRec (int W , int []

val , int [] wt , int n , ref int [,] memo) { // Base Case if (n == 0 || W == 0) return 0 ; // Check if we have previously calculated the same subproblem if (memo [n , W] != - 1) return memo [n , W]; int pick = 0 ; // Pick nth item if it does not exceed the capacity of knapsack if (wt [n - 1] <= W) pick = val [n - 1] + knapsackRec (W - wt [n - 1], val , wt , n - 1 , ref memo); // Don't pick the nth item int notPick = knapsackRec (W , val , wt , n - 1 , ref memo); // Store the result in memo[n, W] and return it return memo [n , W] = Math . Max (pick , notPick); } static int knapsack (int W , int [] val , int [] wt) { int n = val . Length ; // Memoization table to store the results int [,] memo = new int [n + 1 , W + 1]; // Initialize memo table with -1 for (int i = 0 ; i <= n ; i ++) { for (int j = 0 ; j <= W ; j ++) { memo [i , j] = - 1 ; } } return knapsackRec (W , val , wt , n , ref memo); } //Driver Code Starts static void Main () { int [] val = { 1 , 2 , 3 }; int [] wt = { 4 , 5 , 1 }; int W = 4 ; Console . WriteLine (knapsack (W , val , wt)); } } //Driver Code Ends JavaScript // Returns the maximum value that // can be put in a knapsack of capacity W function knapsackRec (W , val , wt , n , memo) { // Base Case if (n === 0 || W === 0) return 0 ; // Check if we have previously calculated the same subproblem if (memo [n][W] !== - 1) return memo [n][W]; let pick = 0 ; // Pick nth item if it does not exceed the capacity of knapsack if (wt [n - 1] <= W) pick = val [n - 1] + knapsackRec (W - wt [n - 1], val , wt , n - 1 , memo); // Don't pick the nth item let notPick = knapsackRec (W , val , wt , n - 1 , memo); // Store the result in memo[n][W] and return it memo [n][W] = Math . max (pick , notPick); return memo [n][W]; } function knapsack (W , val , wt) { const n = val . length ; // Memoization table to store the results const memo = Array . from ({ length : n + 1 }, () => Array (W + 1). fill (- 1)); return knapsackRec (W , val , wt , n , memo); } // Driver Code //Driver Code Starts const val = [1 , 2 , 3]; const wt = [4 , 5 , 1]; const W = 4 ; console . log (knapsack (W , val , wt)); //Driver Code Ends Output 3 [Better Approach - 2] Using Bottom-Up DP (Tabulation) - O(n x W) Time and O(n x W) Space There are two parameters that change in the recursive solution and these parameters go from 0 to n and 0 to W. So we create a 2D dp[][] array of size (n+1) x (W+1), such that dp[i][j] stores the maximum value we can get using i items such that the knapsack capacity is j. We first fill the known entries when m is 0 or n is 0. Then we fill the remaining entries using the recursive formula. For each item i and knapsack capacity j, we decide whether to pick the item or not. If we don't pick the item: dp[i][j] remains same as the previous item, that is dp[i - 1][j]. If we pick the item: dp[i][j] is updated to val[i] + dp[i - 1][j - wt[i]] . C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int knapsack (int W , vector < int > & val , vector < int > & wt) { int n = wt . size (); vector < vector < int >> dp (n + 1 , vector < int > (W + 1)); // Build table dp[][] in bottom-up manner for (int i = 0 ; i <= n ; i ++) { for (int j = 0 ; j <= W ; j ++) { // If there is no item or the knapsack's capacity is 0 if (i == 0 || j == 0) dp [i][j] = 0 ; else { int pick = 0 ; // Pick ith item if it does not exceed the capacity of knapsack if (wt [i - 1] <= j) pick = val [i - 1] + dp [i - 1][j - wt [i - 1]]; // Don't pick the ith item int notPick = dp [i - 1][j]; dp [i][j] = max (pick , notPick); } } } return dp [n][W]; } //Driver Code Starts int main () { vector < int > val = { 1 , 2 , 3 }; vector < int > wt = { 4 , 5 , 1 }; int W = 4 ; cout << knapsack (W , val , wt) << endl ; return 0 ; } //Driver Code Ends Java //Driver Code Starts class GfG { //Driver Code Ends static int knapsack (int W , int [] val , int [] wt) { int n = wt . length ; int [][] dp = new int [n + 1][W + 1] ; // Build table dp[][] in bottom-up manner for (int i = 0 ; i <= n ; i ++) { for (int j = 0 ; j <= W ; j ++) { // If there is no item or the knapsack's capacity is 0 if (i == 0 || j == 0) dp [i][j] = 0 ; else { int pick = 0 ; // Pick ith item if it does not exceed the capacity of knapsack if (wt [i - 1] <= j) pick = val [i - 1] + dp [i - 1][j - wt [i - 1]] ; // Don't pick the ith item int notPick = dp [i - 1][j] ; dp [i][j] = Math . max (pick , notPick); } } } return dp [n][W] ; } //Driver Code Starts public static void main (String [] args) { int [] val = { 1 , 2 , 3 }; int [] wt = { 4 , 5 , 1 }; int W = 4 ; System . out . println (knapsack (W , val , wt)); } } //Driver Code Ends Python def knapsack (W , val , wt): n = len (wt) dp = [[0 for _ in range (W + 1)] for _ in range (n + 1)] # Build table dp[][] in bottom-up manner for i in range (n + 1): for j in range (W + 1): # If there is no item or the knapsack's capacity is 0 if i == 0 or j == 0 : dp [i][j] = 0 else : pick = 0 # Pick ith item if it does not exceed the capacity of knapsack if wt [i - 1] <= j : pick = val [i - 1] + dp [i - 1][j - wt [i - 1]] # Don't pick the ith item notPick = dp [i - 1][j] dp [i][j] = max (pick , notPick) return dp [n][W] if __name__ == "__main__" : #Driver Code Starts val = [1 , 2 , 3] wt = [4 , 5 , 1] W = 4 print (knapsack (W , val , wt)) #Driver Code Ends C# //Driver Code Starts using System ; using System.Linq ; class GfG { //Driver Code Ends static int knapsack (int W , int [] val , int [] wt) { int n = wt . Length ; int [,] dp = new int [n + 1 , W + 1]; // Build table dp[][] in bottom-up manner for (int i = 0 ; i <= n ; i ++) { for (int j = 0 ; j <= W ; j ++) { // If there is no item or the knapsack's capacity is 0 if (i == 0 || j == 0) dp [i , j] = 0 ; else { int pick = 0 ; // Pick ith item if it does not exceed the capacity of knapsack if (wt [i - 1] <= j) pick = val [i - 1] + dp [i - 1 , j - wt [i - 1]]; // Don't pick the ith item int notPick = dp [i - 1 , j]; dp [i , j] = Math . Max (pick , notPick); } } } return dp [n , W]; } //Driver Code Starts static void Main () { int [] val = { 1 , 2 , 3 }; int [] wt = { 4 , 5 , 1 }; int W = 4 ; Console . WriteLine (

knapsack (W , val , wt)); } } //Driver Code Ends JavaScript function knapsack (W , val , wt) { let n = wt . length ; let dp = Array . from ({ length : n + 1 }, () => Array (W + 1). fill (0)); // Build table dp[][] in bottom-up manner for (let i = 0 ; i <= n ; i ++) { for (let j = 0 ; j <= W ; j ++) { // If there is no item or the knapsack's capacity is 0 if (i === 0 || j === 0) dp [i][j] = 0 ; else { let pick = 0 ; // Pick ith item if it does not exceed the capacity of knapsack if (wt [i - 1] <= j) pick = val [i - 1] + dp [i - 1][j - wt [i - 1]]; // Don't pick the ith item let notPick = dp [i - 1][j]; dp [i][j] = Math . max (pick , notPick); } } } return dp [n][W]; } // Driver code //Driver Code Starts let val = [1 , 2 , 3]; let wt = [4 , 5 , 1]; let W = 4 ; console . log (knapsack (W , val , wt)); //Driver Code Ends Output 3 [Expected Approach] Using Bottom-Up DP (Space-Optimized) - O(n x W) Time and O(W) Space To compute the current row of the dp[] array, we only need values from the previous row. Therefore, instead of maintaining the entire 2D dp table, we can optimize space by using just a single 1D array. By traversing the array from right to left, we ensure that previously computed values are not overwritten before they are used. C++ //Driver Code Starts #include <iostream> #include <vector> using namespace std ; //Driver Code Ends int knapsack (int W , vector < int > & val , vector < int > & wt) { // Initializing dp vector vector < int > dp (W + 1 , 0); // Taking first i elements for (int i = 1 ; i <= wt . size (); i ++) { // Starting from back, so that we also have data of // previous computation of i-1 items for (int j = W ; j >= wt [i - 1]; j --) { dp [j] = max (dp [j], dp [j - wt [i - 1]] + val [i - 1]); } } return dp [W]; } //Driver Code Starts int main () { vector < int > val = { 1 , 2 , 3 }; vector < int > wt = { 4 , 5 , 1 }; int W = 4 ; cout << knapsack (W , val , wt) << endl ; return 0 ; } //Driver Code Ends Java //Driver Code Starts class GfG { //Driver Code Ends static int knapsack (int W , int [] val , int [] wt) { // Initializing dp array int [] dp = new int [W + 1] ; // Taking first i elements for (int i = 1 ; i <= wt . length ; i ++) { // Starting from back, so that we also have data of // previous computation of i-1 items for (int j = W ; j >= wt [i - 1] ; j --) { dp [j] = Math . max (dp [j] , dp [j - wt [i - 1]] + val [i - 1]); } } return dp [W] ; } //Driver Code Starts public static void main (String [] args) { int [] val = { 1 , 2 , 3 }; int [] wt = { 4 , 5 , 1 }; int W = 4 ; System . out . println (knapsack (W , val , wt)); } } //Driver Code Ends Python def knapsack (W , val , wt): # Initializing dp list dp = [0] * (W + 1) # Taking first i elements for i in range (1 , len (wt) + 1): # Starting from back, so that we also have data of # previous computation of i-1 items for j in range (W , wt [i - 1] - 1 , - 1): dp [j] = max (dp [j], dp [j - wt [i - 1]] + val [i - 1]) return dp [W] if __name__ == "__main__" : #Driver Code Starts val = [1 , 2 , 3] wt = [4 , 5 , 1] W = 4 print (knapsack (W , val , wt)) #Driver Code Ends C# //Driver Code Starts using System ; class GfG { //Driver Code Ends static int knapsack (int W , int [] val , int [] wt) { // Initializing dp array int [] dp = new int [W + 1]; // Taking first i elements for (int i = 1 ; i <= wt . Length ; i ++) { // Starting from back, so that we also have data of // previous computation of i-1 items for (int j = W ; j >= wt [i - 1]; j --) { dp [j] = Math . Max (dp [j], dp [j - wt [i - 1]] + val [i - 1]); } } return dp [W]; } //Driver Code Starts static void Main () { int [] val = { 1 , 2 , 3 }; int [] wt = { 4 , 5 , 1 }; int W = 4 ; Console . WriteLine (knapsack (W , val , wt)); } } //Driver Code Ends JavaScript function knapsack (W , val , wt) { // Initializing dp array let dp = new Array (W + 1). fill (0); // Taking first i elements for (let i = 1 ; i <= wt . length ; i ++) { // Starting from back, so that we also have data of // previous computation of i-1 items for (let j = W ; j >= wt [i - 1]; j --) { dp [j] = Math . max (dp [j], dp [j - wt [i - 1]] + val [i - 1]); } } return dp [W]; } // Driver Code //Driver Code Starts let val = [1 , 2 , 3]; let wt = [4 , 5 , 1]; let W = 4 ; console . log (knapsack (W , val , wt)); //Driver Code Ends Output 3 Comment Article Tags: Article Tags: Dynamic Programming DSA Snapdeal Zoho MakeMyTrip Visa knapsack + 3 More