Fractional Knapsack - GeeksforGeeks

Source: https://www.geeksforgeeks.org/fractional-knapsack-problem/

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 Fractional Knapsack Last Updated : 2 Sep, 2025 Given two arrays, val[] and wt[] , representing the values and weights of item respectively, and an integer capacity representing the maximum weight a knapsack can hold, we have to determine the maximum total value that can be achieved by putting the items in the knapsack without exceeding its capacity. Items can also be taken in fractional parts if required. Examples: Input: val[] = [60, 100, 120], wt[] = [10, 20, 30], capacity = 50 Output: 240 Explanation: We will take the items of weight 10kg and 20kg and 2/3 fraction of 30kg. Hence total value will be 60 + 100 + (2/3) * 120 = 240. Input: val[] = [500], wt[] = [30], capacity = 10 Output: 166.667 Try it on GfG Practice Why naive approach will fail? Case 1: Picking the items with smaller weights first Example : val[] = [10, 10, 10, 100], wt[] = [10, 10, 10, 30], capacity = 30 If we start picking smaller weights, we can pick the three items of weight 10 $\rightarrow$ total value = 10+10+10 = 30. But the optimal choice is to take the last item (value 100, weight 30) $\rightarrow$ total value = 100. So, choosing smaller weights first fails here. Case 2: Picking items with larger value first Example : val[]= [10, 10, 10, 20], wt[] = [10, 10, 10, 30], capacity = 30 If we start picking higher values, we might choose the last item (value 20, weight 30) $\rightarrow$ total value = 20. But the better choice is to take the three items of weight 10 each $\rightarrow$ total value = 10+10+10 = 30. So, choosing higher values first also fails. [Approach] Selecting Items by value/weight Ratio - O(nlogn) Time and O(n) Space The idea is to always pick items greedily based on their value-to-weight ratio . Take the item with the highest ratio first, then the next highest, and so on, until the knapsack is full. If any item doesn't fully fit, then take its fractional part according to the remaining capacity. Steps to solve the problem: Calculate the ratio (value/weight) for each item. Sort all the items in decreasing order of the ratio. Iterate through items: if the current item fully fits, add its full value and decrease capacity otherwise, take the fractional part that fits and add proportional value. Stop once the capacity becomes zero. C++ #include <iostream> #include <vector> #include <algorithm> using namespace std ; // Comparison function to sort items based on value/weight ratio bool compare (vector < int >& a , vector < int >& b) { double a1 = (1.0 * a [0]) / a [1]; double b1 = (1.0 * b [0]) / b [1]; return a1 > b1 ; } double fractionalKnapsack (vector < int >& val , vector < int >& wt , int capacity) { int n = val . size (); // Create 2D vector to store value and weight // items[i][0] = value, items[i][1] = weight vector < vector < int >> items (n , vector < int > (2)); for (int i = 0 ; i < n ; i ++) { items [i][0] = val [i]; items [i][1] = wt [i]; } // Sort items based on value-to-weight ratio in descending order sort (items . begin (), items . end (), compare); double res = 0.0 ; int currentCapacity = capacity ; // Process items in sorted order for (int i = 0 ; i < n ; i ++) { // If we can take the entire item if (items [i][1] <= currentCapacity) { res += items [i][0]; currentCapacity -= items [i][1]; } // Otherwise take a fraction of the item else { res += (1.0 * items [i][0] / items [i][1]) * currentCapacity ; // Knapsack is full break ; } } return res ; } int main () { vector < int > val = { 60 , 100 , 120 }; vector < int > wt = { 10 , 20 , 30 }; int capacity = 50 ; cout << fractionalKnapsack (val , wt , capacity) << endl ; return 0 ; } Java import java.util.* ; class GfG { // Comparison function to sort items based on value/weight ratio static class ItemComparator implements Comparator < int []> { public int compare (int [] a , int [] b) { double a1 = (1.0 * a [0]) / a [1] ; double b1 = (1.0 * b [0]) / b [1] ; return Double . compare (b1 , a1); } } static double fractionalKnapsack (int [] val , int [] wt , int capacity) { int n = val . length ; // Create 2D array to store value and weight // items[i][0] = value, items[i][1] = weight int [][] items = new int [n][2] ; for (int i = 0 ; i < n ; i ++) { items [i][0] = val [i] ; items [i][1] = wt [i] ; } // Sort items based on value-to-weight ratio in descending order Arrays . sort (items , new

ItemComparator ()); double res = 0.0 ; int currentCapacity = capacity ; // Process items in sorted order for (int i = 0 ; i < n ; i ++) { // If we can take the entire item if (items [i][1] <= currentCapacity) { res += items [i][0] ; currentCapacity -= items [i][1] ; } // Otherwise take a fraction of the item else { res += (1.0 * items [i][0] / items [i][1]) * currentCapacity ; // Knapsack is full break ; } } return res ; } public static void main (String [] args) { int [] val = { 60 , 100 , 120 }; int [] wt = { 10 , 20 , 30 }; int capacity = 50 ; System . out . println (fractionalKnapsack (val , wt , capacity)); } } Python def compare (a , b): a1 = (1.0 * a [0]) / a [1] b1 = (1.0 * b [0]) / b [1] return b1 - a1 def fractionalKnapsack (val , wt , capacity): n = len (val) # Create list to store value and weight # items[i][0] = value, items[i][1] = weight items = [[val [i], wt [i]] for i in range (n)] # Sort items based on value-to-weight ratio in descending order items . sort (key = lambda x : x [0] / x [1], reverse = True) res = 0.0 currentCapacity = capacity # Process items in sorted order for i in range (n): # If we can take the entire item if items [i][1] <= currentCapacity : res += items [i][0] currentCapacity -= items [i][1] # Otherwise take a fraction of the item else : res += (1.0 * items [i][0] / items [i][1]) * currentCapacity # Knapsack is full break return res if __name__ == "__main__" : val = [60 , 100 , 120] wt = [10 , 20 , 30] capacity = 50 print (fractionalKnapsack (val , wt , capacity)) C# using System ; using System.Collections.Generic ; class GfG { // Comparison function to sort items based on value/weight ratio class ItemComparer : IComparer < int [] > { public int Compare (int [] a , int [] b) { double a1 = (1.0 * a [0]) / a [1]; double b1 = (1.0 * b [0]) / b [1]; return b1 . CompareTo (a1); } } static double fractionalKnapsack (int [] val , int [] wt , int capacity) { int n = val . Length ; // Create 2D array to store value and weight // items[i][0] = value, items[i][1] = weight int [][] items = new int [n][]; for (int i = 0 ; i < n ; i ++) { items [i] = new int [2]; items [i][0] = val [i]; items [i][1] = wt [i]; } // Sort items based on value-to-weight ratio in descending order Array . Sort (items , new ItemComparer ()); double res = 0.0 ; int currentCapacity = capacity ; // Process items in sorted order for (int i = 0 ; i < n ; i ++) { // If we can take the entire item if (items [i][1] <= currentCapacity) { res += items [i][0]; currentCapacity -= items [i][1]; } // Otherwise take a fraction of the item else { res += (1.0 * items [i][0] / items [i][1]) * currentCapacity ; // Knapsack is full break ; } } return res ; } static void Main () { int [] val = { 60 , 100 , 120 }; int [] wt = { 10 , 20 , 30 }; int capacity = 50 ; Console . WriteLine (fractionalKnapsack (val , wt , capacity)); } } JavaScript function compare (a , b) { let a1 = (1.0 * a [0]) / a [1]; let b1 = (1.0 * b [0]) / b [1]; return b1 - a1 ; } function fractionalKnapsack (val , wt , capacity) { let n = val . length ; // Create array to store value and weight // items[i][0] = value, items[i][1] = weight let items = []; for (let i = 0 ; i < n ; i ++) { items . push ([val [i], wt [i]]); } // Sort items based on value-to-weight ratio in descending order items . sort (compare); let res = 0.0 ; let currentCapacity = capacity ; // Process items in sorted order for (let i = 0 ; i < n ; i ++) { // If we can take the entire item if (items [i][1] <= currentCapacity) { res += items [i][0]; currentCapacity -= items [i][1]; } // Otherwise take a fraction of the item else { res += (1.0 * items [i][0] / items [i][1]) * currentCapacity ; // Knapsack is full break ; } } return res ; } // Driver Code let val = [60 , 100 , 120]; let wt = [10 , 20 , 30]; let capacity = 50 ; console . log (fractionalKnapsack (val , wt , capacity)); Output 240 Comment Article Tags: Article Tags: Greedy DSA knapsack Fraction