**Problem Statement**

Betty is organizing a charity event and needs to collect a specific amount of money using the least number of coins from given denominations. Help Betty by writing a program that calculates the minimum number of coins required to achieve the target amount. If it is not possible to reach the exact amount with the given coins, the program should indicate this.

**Note:** Assume that you have an infinite number of each kind of coin.

**Example 1**

**Input:**

1 2 5

11

**Output:**

3

**Explanation**:

11 = 5 + 5 + 1

**Example 2**

**Input:**

2

3

**Output:**

-1

**Input format :**

The first line of input consists of a string s, representing the space-separated denominations of coins.

The second line of input consists of an integer n, representing the target amount.

**Output format :**

The output displays the minimum number of coins required to make up the target amount.

If it is not possible, print -1.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ s.length ≤ 12

0 ≤ n ≤ 104

**Sample test cases :**

**Input 1 :**

1 2 5

11

**Output 1 :**

3

**Input 2 :**

1

0

**Output 2 :**

0

**Input 3 :**

2

3

**Output 3 :**

-1

// You are using Java

import java.util.\*; // Importing the utilities package which contains classes like Scanner and Arrays

class CoinChange {

    // Function to find the minimum number of coins needed to make the given amount

    public static int minCoins(int[] coins, int amount) {

        // Initialize the dp array with a large number

        int[] dp = new int[amount + 1];

        Arrays.fill(dp, amount + 1); // Use amount + 1 as infinity to signify initially impossible states

        dp[0] = 0; // Base case: zero coins are needed to make amount 0

        // Fill the dp array

        for (int i = 1; i <= amount; i++) {

            // Iterate through each coin denomination

            for (int coin : coins) {

                // Check if the current coin can be used to make up the amount i

                if (i - coin >= 0) {

                    // Update dp[i] with the minimum coins needed by comparing current value with new value

                    dp[i] = Math.min(dp[i], dp[i - coin] + 1);

                }

            }

        }

        // Return the result: if dp[amount] is still greater than amount, it means it's not possible to make that amount

        return dp[amount] > amount ? -1 : dp[amount];

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Read coin denominations from the input

        String[] coinStrings = scanner.nextLine().split(" ");

        int[] coins = new int[coinStrings.length];

        for (int i = 0; i < coinStrings.length; i++) {

            coins[i] = Integer.parseInt(coinStrings[i]);

        }

        // Read the target amount from the input

        int amount = scanner.nextInt();

        // Calculate and print the minimum number of coins needed

        System.out.println(minCoins(coins, amount));

    }

}

**Explanation:**

1. **Import Statement**:
   * import java.util.\*; imports necessary classes like Scanner and Arrays.
2. **Class Declaration**:
   * class CoinChange declares a class named CoinChange.
3. **minCoins Method**:
   * This method calculates the minimum number of coins required to make the given amount using dynamic programming.
   * int[] dp = new int[amount + 1]; initializes an array dp to store the minimum number of coins needed for each amount from 0 to amount.
   * Arrays.fill(dp, amount + 1); fills the dp array with amount + 1, a large number representing "infinity" since initially, we assume it's impossible to form those amounts.
   * dp[0] = 0; sets the base case where zero coins are needed to make the amount 0.
   * The nested loops iterate through each possible amount (i from 1 to amount) and each coin denomination (coin).
   * if (i - coin >= 0) checks if using the current coin is feasible.
   * dp[i] = Math.min(dp[i], dp[i - coin] + 1); updates the dp value for the current amount i by comparing the existing value with the new value obtained by using one more coin.
   * return dp[amount] > amount ? -1 : dp[amount]; returns -1 if it's not possible to form the amount, otherwise, it returns the minimum number of coins.
4. **main Method**:
   * Scanner scanner = new Scanner(System.in); creates a Scanner object to read input from the console.
   * String[] coinStrings = scanner.nextLine().split(" "); reads a line of input, splits it by spaces, and converts each substring into an integer array coins.
   * int amount = scanner.nextInt(); reads the target amount from the input.
   * System.out.println(minCoins(coins, amount)); calculates and prints the minimum number of coins needed to make the amount.

This approach ensures an efficient solution to the Coin Change problem using dynamic programming, with a time complexity of O(n \* m), where n is the amount and m is the number of coin denominations.

Sure! Let’s go through a detailed example to understand how the Coin Change algorithm works.

**Example**

**Coin Denominations:** 1, 2, 5

**Target Amount:** 11

We want to find the minimum number of coins needed to make the amount 11.

**Step-by-Step Execution**

1. **Initialization:**
   * Create an array dp where dp[i] represents the minimum number of coins needed to make the amount i.
   * Initialize dp with amount + 1 (which is 12 in this case) to represent infinity.
   * Set dp[0] to 0 because zero coins are needed to make amount 0.

java

Copy code

int[] dp = new int[12];

Arrays.fill(dp, 12);

dp[0] = 0;

So initially:

css

Copy code

dp = [0, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12]

1. **Filling the dp Array:**

Iterate through each amount i from 1 to 11, and for each coin, update the dp array.

* + For i = 1:
    - With coin 1: dp[1] = Math.min(dp[1], dp[1 - 1] + 1) = Math.min(12, dp[0] + 1) = 1

Updated dp:

css

Copy code

dp = [0, 1, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12]

* + For i = 2:
    - With coin 1: dp[2] = Math.min(dp[2], dp[2 - 1] + 1) = Math.min(12, dp[1] + 1) = 2
    - With coin 2: dp[2] = Math.min(dp[2], dp[2 - 2] + 1) = Math.min(2, dp[0] + 1) = 1

Updated dp:

css

Copy code

dp = [0, 1, 1, 12, 12, 12, 12, 12, 12, 12, 12, 12]

* + For i = 3:
    - With coin 1: dp[3] = Math.min(dp[3], dp[3 - 1] + 1) = Math.min(12, dp[2] + 1) = 2
    - With coin 2: dp[3] = Math.min(dp[3], dp[3 - 2] + 1) = Math.min(2, dp[1] + 1) = 2

Updated dp:

css

Copy code

dp = [0, 1, 1, 2, 12, 12, 12, 12, 12, 12, 12, 12]

* + For i = 4:
    - With coin 1: dp[4] = Math.min(dp[4], dp[4 - 1] + 1) = Math.min(12, dp[3] + 1) = 3
    - With coin 2: dp[4] = Math.min(dp[4], dp[4 - 2] + 1) = Math.min(3, dp[2] + 1) = 2

Updated dp:

css

Copy code

dp = [0, 1, 1, 2, 2, 12, 12, 12, 12, 12, 12, 12]

* + Continue this process up to i = 11:

After processing all amounts up to 11, the final dp array looks like this:

css

Copy code

dp = [0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 3]

1. **Result:**
   * dp[11] gives us the minimum number of coins needed to make the amount 11, which is 3 in this case.

Therefore, the minimum number of coins needed to make the amount 11 is 3. The coins that achieve this are 5, 5, 1 or 5, 2, 2, 2 (among other combinations).

**Summary**

* **Initialization:** Set all values in dp to "infinity" (i.e., amount + 1), except for dp[0] which is 0.
* **Dynamic Programming Table Update:** Update the dp array by considering each coin and each amount.
* **Result Extraction:** The value dp[amount] represents the minimum number of coins needed. If dp[amount] is still amount + 1, it means it's not possible to form that amount with the given coins.

This approach efficiently computes the solution using dynamic programming.

**Problem Statement**

You are given a knapsack with a maximum weight capacity of W and n objects. Each object i has a weight c[i] and a costv[i]. Your task is to fill the knapsack with objects in such a way that the total cost of the objects in the knapsack is maximized without exceeding the weight capacity.

Implement a program to solve this problem using a simple greedy algorithm known as the Fractional Knapsack Problem.

**Input format :**

The first line of input consists of an integer n, representing the number of objects.

The second line of input consists of n integers, separated by a space, representing the weight of the object.

The third line of input consists of n integers, separated by a space, representing the cost of the object.

The fourth line of input consists of an integer W, representing the maximum weight capacity of the knapsack.

**Output format :**

The output displays the "Objects worth Rs. V." where Rs. V represents the total cost of the objects in the knapsack as the float value with the two decimal points.

**Refer to the sample output for the formatting specifications.**

**Code constraints :**

1 ≤ n ≤ 10

1 ≤ c[i], v[i] ≤ 1000

1 ≤ W ≤ 100

**Sample test cases :**

**Input 1 :**

3

10 20 30

60 100 120

50

**Output 1 :**

Objects worth Rs.240.00

**Input 2 :**

4

10 20 30 40

60 100 120 200

90

**Output 2 :**

Objects worth Rs.440.00

import java.util.\*; // Import the utilities package for Scanner and Arrays

class FractionalKnapsack {

    // Nested class to represent an item with weight, cost, and value per weight

    static class Item {

        int weight; // Weight of the item

        int cost; // Cost (value) of the item

        double valuePerWeight; // Value per unit weight of the item

        // Constructor to initialize the item

        Item(int weight, int cost) {

            this.weight = weight;

            this.cost = cost;

            // Calculate value per weight as cost divided by weight

            this.valuePerWeight = (double) cost / weight;

        }

    }

    // Method to calculate the maximum value that can be achieved with the given weight capacity

    public static double getMaxValue(int n, int[] weights, int[] costs, int W) {

        // Create an array to hold Item objects

        Item[] items = new Item[n];

        for (int i = 0; i < n; i++) {

            // Initialize each item with weight and cost

            items[i] = new Item(weights[i], costs[i]);

        }

        // Sort items by value per weight in descending order

        Arrays.sort(items, (a, b) -> Double.compare(b.valuePerWeight, a.valuePerWeight));

        double totalValue = 0.0; // Variable to accumulate the total value

        int remainingCapacity = W; // Remaining capacity of the knapsack

        // Iterate through the sorted items

        for (Item item : items) {

            if (remainingCapacity == 0) break; // If the knapsack is full, exit the loop

            // Determine the amount of the current item to take

            int weightToTake = Math.min(item.weight, remainingCapacity);

            // Add the value of the taken weight to the total value

            totalValue += weightToTake \* item.valuePerWeight;

            // Decrease the remaining capacity by the taken weight

            remainingCapacity -= weightToTake;

        }

        // Return the total value that can be achieved

        return totalValue;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in); // Create a Scanner object for input

        // Read the number of items

        int n = scanner.nextInt();

        // Create arrays to hold weights and costs

        int[] weights = new int[n];

        int[] costs = new int[n];

        // Read the weights of the items

        for (int i = 0; i < n; i++) {

            weights[i] = scanner.nextInt();

        }

        // Read the costs (values) of the items

        for (int i = 0; i < n; i++) {

            costs[i] = scanner.nextInt();

        }

        // Read the maximum weight capacity of the knapsack

        int W = scanner.nextInt();

        // Calculate the maximum value that can be obtained

        double maxValue = getMaxValue(n, weights, costs, W);

        // Print the result formatted to two decimal places

        System.out.printf("Objects worth Rs.%.2f%n", maxValue);

    }

}

Certainly! Let's break down the FractionalKnapsack problem using the provided Java code. This problem involves maximizing the total value of items that can be placed in a knapsack with a given weight capacity, where you can take fractions of items.

**Code Breakdown**

**Imports and Class Declaration**

java

Copy code

import java.util.\*;

class FractionalKnapsack {

* Import the java.util package for utility classes like Scanner and Arrays.
* Define the FractionalKnapsack class.

**Item Class**

java

Copy code

static class Item {

int weight;

int cost;

double valuePerWeight;

Item(int weight, int cost) {

this.weight = weight;

this.cost = cost;

this.valuePerWeight = (double) cost / weight;

}

}

* Define a nested static class Item to represent each item.
* Each item has:
  + weight: The weight of the item.
  + cost: The cost (or value) of the item.
  + valuePerWeight: The value per unit weight, calculated as cost / weight.

**getMaxValue Method**

java

Copy code

public static double getMaxValue(int n, int[] weights, int[] costs, int W) {

// Create an array of items

Item[] items = new Item[n];

for (int i = 0; i < n; i++) {

items[i] = new Item(weights[i], costs[i]);

}

* **Create Items Array:** Initialize an array items of Item objects.
* Populate it with items based on weights and costs.

java

Copy code

// Sort items by value per weight in descending order

Arrays.sort(items, (a, b) -> Double.compare(b.valuePerWeight, a.valuePerWeight));

* **Sort Items:** Sort the items array based on valuePerWeight in descending order using a lambda function for comparison.

java

Copy code

double totalValue = 0.0;

int remainingCapacity = W;

* **Initialize Variables:** totalValue to accumulate the maximum value and remainingCapacity to keep track of the remaining knapsack capacity.

java

Copy code

// Iterate through the sorted items

for (Item item : items) {

if (remainingCapacity == 0) break;

// Take as much as possible of the item

int weightToTake = Math.min(item.weight, remainingCapacity);

totalValue += weightToTake \* item.valuePerWeight;

remainingCapacity -= weightToTake;

}

return totalValue;

}

* **Iterate Items:** For each item (starting from the highest value per weight):
  + If the remaining capacity is 0, exit the loop.
  + **Determine Weight to Take:** Take the minimum of the item’s weight or the remaining capacity.
  + **Update Total Value:** Add the value of the taken weight to totalValue.
  + **Update Remaining Capacity:** Subtract the taken weight from remainingCapacity.

**main Method**

java

Copy code

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Read number of objects

int n = scanner.nextInt();

// Read weights of the objects

int[] weights = new int[n];

for (int i = 0; i < n; i++) {

weights[i] = scanner.nextInt();

}

// Read costs of the objects

int[] costs = new int[n];

for (int i = 0; i < n; i++) {

costs[i] = scanner.nextInt();

}

// Read maximum weight capacity

int W = scanner.nextInt();

// Calculate the maximum value

double maxValue = getMaxValue(n, weights, costs, W);

// Print the result formatted to two decimal places

System.out.printf("Objects worth Rs.%.2f%n", maxValue);

}

}

* **Input Reading:**
  + **Number of Items (n)**: Read the number of items.
  + **Weights Array:** Read the weights for each item.
  + **Costs Array:** Read the costs for each item.
  + **Maximum Weight Capacity (W)**: Read the maximum weight capacity of the knapsack.
* **Calculate and Output:** Call getMaxValue to compute the maximum value and print it formatted to two decimal places.

**Example Execution**

Let’s say we have the following inputs:

* **Weights:** 10 20 30
* **Costs:** 60 100 120
* **Maximum Weight Capacity:** 50

**Steps:**

1. Create items with weights and costs.
2. Compute value per weight for each item:
   * Item 1: 60 / 10 = 6.0
   * Item 2: 100 / 20 = 5.0
   * Item 3: 120 / 30 = 4.0
3. Sort items by value per weight: [Item 1, Item 2, Item 3].
4. Compute maximum value:
   * Take 10 units of Item 1 (full item), total value = 60, remaining capacity = 40.
   * Take 20 units of Item 2 (full item), total value = 100 + 60 = 160, remaining capacity = 20.
   * Take 20 units of Item 3, total value = 160 + 80 = 240, remaining capacity = 0.

**Output:** Objects worth Rs.240.00

This approach ensures you get the maximum value for the given knapsack capacity by always picking items with the highest value-to-weight ratio first, and considering fractional parts of items as needed.

**Problem Statement**

Sarah loves challenges and has recently taken up a new programming problem. She is fascinated by the classic **"0/1 Knapsack Problem"** and wants to create a program that can efficiently solve it.

The 0/1 Knapsack Problem involves selecting a combination of items with given weights and values to maximize the total value, while not exceeding a given weight capacity.

Write a program that takes input for the 0/1 Knapsack Problem and outputs the maximum value that can be achieved, as well as the selected weights that contribute to this maximum value.

**Input format :**

The first line contains an integer n, representing the number of items available.

The second line contains n integers separated by space, representing the values of the items. The ith integer corresponds to the value of the ith item.

The third line contains n integers separated by space, representing the weights of the items. The ith integer corresponds to the weight of the ith item.

The fourth line contains an integer W, representing the maximum weight capacity of the knapsack.

**Output format :**

The first line output displays "Maximum value: " followed by an integer, representing the maximum value that can be achieved.

The second line output displays "Selected weights: followed by space-separated integers, representing the selected weights contributing to the maximum value.

**Refer to the sample outputs for the exact format.**

**Code constraints :**

In this scenario, the test cases fall under the following constraints:

1 ≤ n ≤ 10

1 ≤ val[i] ≤ 100

1 ≤ wt[i] ≤ 100

1 ≤ W ≤ 1000

**Sample test cases :**

**Input 1 :**

3

60 100 120

10 20 30

50

**Output 1 :**

Maximum value: 220

Selected weights: 30 20

**Input 2 :**

4

10 20 30 40

1 2 3 4

8

**Output 2 :**

Maximum value: 80

Selected weights: 4 3 1

import java.util.\*; // Import utility classes for Scanner, ArrayList, and Collections

class KnapsackProblem {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in); // Create a Scanner object for input

        // Read number of items

        int n = scanner.nextInt();

        // Create an array to hold the values of the items

        int[] values = new int[n];

        for (int i = 0; i < n; i++) {

            // Read each value of the items

            values[i] = scanner.nextInt();

        }

        // Create an array to hold the weights of the items

        int[] weights = new int[n];

        for (int i = 0; i < n; i++) {

            // Read each weight of the items

            weights[i] = scanner.nextInt();

        }

        // Read the maximum weight capacity of the knapsack

        int W = scanner.nextInt();

        // Call the knapsack function and get the result

        Result result = knapsack(n, values, weights, W);

        // Print the maximum value that can be obtained

        System.out.println("Maximum value: " + result.maxValue);

        // Print the selected weights

        System.out.print("Selected weights: ");

        for (int weight : result.selectedWeights) {

            // Print each selected weight

            System.out.print(weight + " ");

        }

        System.out.println(); // Newline for better readability

    }

    // Inner class to hold the result of the knapsack problem

    static class Result {

        int maxValue; // Maximum value that can be obtained

        List<Integer> selectedWeights; // List of selected weights

        // Constructor to initialize Result object

        Result(int maxValue, List<Integer> selectedWeights) {

            this.maxValue = maxValue;

            this.selectedWeights = selectedWeights;

        }

    }

    // Function to solve the knapsack problem using dynamic programming

    public static Result knapsack(int n, int[] values, int[] weights, int W) {

        // Create a DP table with dimensions (n+1) x (W+1)

        int[][] dp = new int[n + 1][W + 1];

        // Build the DP table

        for (int i = 1; i <= n; i++) { // Iterate over each item

            for (int w = 0; w <= W; w++) { // Iterate over each capacity from 0 to W

                if (weights[i - 1] <= w) { // If the item can be included in the knapsack

                    // Choose the maximum value between including and not including the current item

                    dp[i][w] = Math.max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1]);

                } else { // If the item cannot be included

                    dp[i][w] = dp[i - 1][w];

                }

            }

        }

        // Maximum value that can be obtained with full capacity W

        int maxValue = dp[n][W];

        // Traceback to find the selected weights

        List<Integer> selectedWeights = new ArrayList<>();

        int remainingCapacity = W;

        for (int i = n; i > 0 && remainingCapacity > 0; i--) {

            // If the value with the current item is different from the value without it

            if (dp[i][remainingCapacity] != dp[i - 1][remainingCapacity]) {

                // Item i-1 is included

                selectedWeights.add(weights[i - 1]);

                remainingCapacity -= weights[i - 1]; // Reduce the remaining capacity

            }

        }

        // Sort the selected weights in descending order for better readability

        Collections.sort(selectedWeights, Collections.reverseOrder());

        // Return the result with maximum value and selected weights

        return new Result(maxValue, selectedWeights);

    }

}

Let's walk through a detailed example of the knapsack problem with iterations to understand how the dynamic programming approach works in the provided Java code.

**Example**

Suppose we have the following items and constraints:

* **Number of Items (n):** 4
* **Values:** [60, 100, 120, 200]
* **Weights:** [10, 20, 30, 40]
* **Maximum Weight Capacity (W):** 50

**Step-by-Step Iteration**

**1. Initialize the DP Table**

We create a 2D DP table where dp[i][w] represents the maximum value that can be obtained with the first i items and a knapsack capacity of w. The dimensions of the table are (n+1) x (W+1) which is 5 x 51.

**DP Table Initialization:**

markdown

Copy code

0 1 2 ... 50

0 0 0 0 ... 0

1 0

2 0

3 0

4 0

**2. Fill the DP Table**

**Iteration 1 (Item 1, Value: 60, Weight: 10):**

* For each capacity w from 0 to 50:
  + If w >= 10, update dp[1][w] as Math.max(dp[0][w], dp[0][w - 10] + 60).

**DP Table after Item 1:**

markdown

Copy code

0 1 2 ... 10 11 12 ... 50

0 0 0 0 ... 60 60 60 ... 60

1 0

2 0

3 0

4 0

**Iteration 2 (Item 2, Value: 100, Weight: 20):**

* For each capacity w from 0 to 50:
  + If w >= 20, update dp[2][w] as Math.max(dp[1][w], dp[1][w - 20] + 100).

**DP Table after Item 2:**

markdown

Copy code

0 1 2 ... 20 21 22 ... 50

0 0 0 0 ... 100 100 100 ... 160

1 0

2 0

3 0

4 0

**Iteration 3 (Item 3, Value: 120, Weight: 30):**

* For each capacity w from 0 to 50:
  + If w >= 30, update dp[3][w] as Math.max(dp[2][w], dp[2][w - 30] + 120).

**DP Table after Item 3:**

markdown

Copy code

0 1 2 ... 30 31 32 ... 50

0 0 0 0 ... 120 120 120 ... 220

1 0

2 0

3 0

4 0

**Iteration 4 (Item 4, Value: 200, Weight: 40):**

* For each capacity w from 0 to 50:
  + If w >= 40, update dp[4][w] as Math.max(dp[3][w], dp[3][w - 40] + 200).

**DP Table after Item 4:**

markdown

Copy code

0 1 2 ... 40 41 42 ... 50

0 0 0 0 ... 200 200 200 ... 320

1 0

2 0

3 0

4 0

**Final Result**

**Maximum Value:**

The maximum value that can be obtained with a knapsack capacity of 50 is dp[4][50], which is 320.

**Traceback to Find Selected Weights**

To find out which weights contribute to this maximum value, we perform a traceback starting from dp[4][50].

1. Start from i = 4 and remainingCapacity = 50:
   * Check if dp[4][50] is different from dp[3][50]. It is, so item 4 is included.
   * Add weight 40 to selectedWeights.
   * Update remainingCapacity to 10 (50 - 40).
2. Move to i = 3 and remainingCapacity = 10:
   * Check if dp[3][10] is different from dp[2][10]. It is not, so item 3 is not included.
3. Move to i = 2 and remainingCapacity = 10:
   * Check if dp[2][10] is different from dp[1][10]. It is not, so item 2 is not included.
4. Move to i = 1 and remainingCapacity = 10:
   * Check if dp[1][10] is different from dp[0][10]. It is, so item 1 is included.
   * Add weight 10 to selectedWeights.
   * Update remainingCapacity to 0 (10 - 10).

**Selected Weights: [40, 10]**

**Summary**

* **Maximum Value:** 320
* **Selected Weights:** [40, 10]

This step-by-step example shows how the dynamic programming table is built and how we traceback to determine which items contribute to the optimal solution.