**LAB SHEET - 5**

AIM🡪 Knapsack greedy approach

The Knapsack problem is a classic optimization problem that can be solved using different approaches, including greedy algorithms. The greedy approach is applicable specifically to the **Fractional Knapsack Problem**, where you can take fractions of items. In contrast, the **0/1 Knapsack Problem** requires you to either take the whole item or leave it.

**Fractional Knapsack Problem**

In the Fractional Knapsack Problem, you are given a set of items, each with a weight and a value, and a maximum weight capacity for the knapsack. The goal is to maximize the total value in the knapsack.

**Greedy Approach**

The greedy approach involves the following steps:

1. Calculate the value-to-weight ratio for each item.
2. Sort the items based on this ratio in descending order.
3. Start adding items to the knapsack, beginning with the item with the highest ratio.
4. If the item can fit entirely in the knapsack, take it all; if not, take the fraction that fits.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Structure to represent an item

struct Item {

int value;

int weight;

double ratio; // Value-to-weight ratio

// Constructor to initialize the item

Item(int v, int w) : value(v), weight(w) {

ratio = static\_cast<double>(v) / w; // Calculate the ratio

}

};

// Comparison function to sort items by their value-to-weight ratio

bool compare(Item a, Item b) {

return a.ratio > b.ratio; // Sort in descending order

}

// Function to solve the Fractional Knapsack Problem

double fractionalKnapsack(int capacity, vector<Item>& items) {

// Sort items based on value-to-weight ratio

sort(items.begin(), items.end(), compare);

double totalValue = 0.0; // Total value accumulated in the knapsack

for (const auto& item : items) {

if (capacity <= 0) { // If the knapsack is full, break out of the loop

break;

}

if (item.weight <= capacity) { // If the item can fit in the knapsack

totalValue += item.value; // Take the whole item

capacity -= item.weight; // Decrease the remaining capacity

} else { // If the item cannot fit, take the fraction that fits

totalValue += item.ratio \* capacity; // Take the fraction of the item

capacity = 0; // The knapsack is now full

}

}

return totalValue; // Return the maximum value

}

int main() {

vector<Item> items = { Item(60, 10), Item(100, 20), Item(120, 30) }; // (value, weight)

int capacity = 50; // Maximum capacity of the knapsack

double maxValue = fractionalKnapsack(capacity, items);

cout << "Maximum value in the knapsack: " << maxValue << endl;

return 0;

}

**Explanation of the Code**

1. **Item Structure**: This structure represents an item with its value, weight, and value-to-weight ratio. The constructor initializes these values and calculates the ratio.
2. **Comparison Function**: The **compare** function is used to sort the items based on their value-to-weight ratio in descending order.
3. **fractionalKnapsack Function**:
   * It takes the maximum capacity of the knapsack and a vector of items.
   * The items are sorted using the **sort** function and the **compare** function.
   * The function iterates through the sorted items:
     + If the item can fit entirely in the knapsack, it adds the entire value of the item to **totalValue** and reduces the capacity.
     + If the item cannot fit, it calculates the fraction of the item that can fit, adds the corresponding value to **totalValue**, and sets the capacity to zero (indicating the knapsack is full).
4. **Main Function**: The **main** function creates a vector of items and a knapsack capacity, then calls the **fractionalKnapsack** function to compute the maximum value that can be carried in the knapsack. Finally, it prints the result.

**Output 🡪**

