Name:

Roll No:

Division:

Batch:

**Assignment No: 1**

**Problem Statement:**

Write a program to implement Fractional knapsack using

a. Greedy algorithm and

b. 0/1 knapsack using dynamic programming.

c. Show that Greedy strategy does not necessarily yield an optimal solution

over a dynamic programming approach.

**Program: (With proper comments):**

#include <iostream>

#include <algorithm>

using namespace std;

// Define a struct to represent items with weight and profit.

struct Item

{

int weight;

int profit;

};

// Function to solve Fractional Knapsack using Greedy Algorithm

double fractionalKnapsack(Item items[], int numItems, int capacity)

{

// Sort items in descending order of profit-to-weight ratio using a lambda function.

sort(items, items + numItems, [](const Item &a, const Item &b)

{ return static\_cast<double>(a.profit) / a.weight > static\_cast<double>(b.profit) / b.weight; });

double totalProfit = 0.0;

int currentWeight = 0;

for (int i = 0; i < numItems; ++i)

{

if (currentWeight + items[i].weight <= capacity)

{

// Add the entire item to the knapsack if it fits.

totalProfit += items[i].profit;

currentWeight += items[i].weight;

}

else

{

// Add a fraction of the item to fill the knapsack to its capacity.

double remainingCapacity = capacity - currentWeight;

totalProfit += (remainingCapacity / items[i].weight) \* items[i].profit;

break;

}

}

return totalProfit;

}

// Function to solve 0/1 Knapsack using Dynamic Programming

int knapsack01(Item items[], int numItems, int capacity)

{

// Create a 2D array dp to store the maximum profit for each item and capacity combination.

int dp[numItems + 1][capacity + 1];

for (int i = 0; i <= numItems; i++)

{

for (int w = 0; w <= capacity; w++)

{

if (i == 0 || w == 0)

{

// Base case: no items or no capacity, profit is zero.

dp[i][w] = 0;

}

else if (items[i - 1].weight <= w)

{

// If the current item can fit in the knapsack, choose the maximum of including or excluding it.

dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - items[i - 1].weight] + items[i - 1].profit);

}

else

{

// If the current item is too heavy, exclude it.

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

}

}

}

return dp[numItems][capacity];

}

int main()

{

int capacity;

cout << "Enter the capacity of the knapsack: ";

cin >> capacity; // Initialize bag capacity here

const int numItems = 3; // Initialize the number of items here

Item items[numItems] = {

// Initialize all items and their respective weights and profits here

{10, 60},

{20, 100},

{30, 120}};

// Calculate profit using Greedy Fractional Knapsack

double greedyProfit = fractionalKnapsack(items, numItems, capacity);

// Calculate profit using 0/1 Knapsack using Dynamic Programming

int dpProfit = knapsack01(items, numItems, capacity);

cout << "Greedy Fractional Knapsack Profit: " << greedyProfit << endl;

cout << "0/1 Knapsack Profit (DP): " << dpProfit << endl;

if (greedyProfit != dpProfit)

{

cout << "Greedy strategy does not yield the optimal solution." << endl;

}

else

{

cout << "Greedy strategy yields the optimal solution." << endl;

}

return 0;

}

**Output:**

Enter the capacity of the knapsack: 50

Greedy Fractional Knapsack Profit: 240

0/1 Knapsack Profit (DP): 220

Greedy strategy does not yield the optimal solution.

Enter the capacity of the knapsack: 70

Greedy Fractional Knapsack Profit: 280

0/1 Knapsack Profit (DP): 280

Greedy strategy yields the optimal solution.

Enter the capacity of the knapsack: 40

Greedy Fractional Knapsack Profit: 200

0/1 Knapsack Profit (DP): 180

Greedy strategy does not yield the optimal solution.