|  |  |  |
| --- | --- | --- |
|  | Pimpri Chinchwad Education Trust’s  **Pimpri Chinchwad College of Engineering (PCCoE)**  (An Autonomous Institute)  Affiliated to Savitribai Phule Pune University (SPPU) ISO 21001:2018 Certified by TUV |  |
| **Department**: Information Technology **Academic Year**: 2025-26 **Semester:** V | | |
| **DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY** | | |

ASSIGNMENT NO. 3

**Aim:-** The aim of this study is to design and implement the Fractional Knapsack algorithm to maximize the total utility value of critical relief supplies transported during emergency flood relief operations. The solution ensures optimal use of limited boat capacity while prioritizing high-utility items such as medicine, food, and drinking water.

# Objective:

* To model the flood relief supply transport as an optimization problem.
* To implement the **Fractional Knapsack Algorithm** for maximizing utility value.
* To prioritize **high-utility items** under given weight capacity constraints.
* To allow **fractional selection of divisible items** (e.g., food, water).
* To evaluate efficiency and effectiveness of the greedy approach.

# Problem Statement:

Scenario: Emergency Relief Supply Distribution

A devastating flood has hit multiple villages in a remote area. The government, along with NGOs, is conducting an emergency relief operation. A rescue team must transport essential supplies from a relief center to affected villages using a boat with limited capacity W.

Each relief item has:

* Weight (wi) in kilograms.
* Utility value (vi) indicating its importance (e.g., medicine > food).
* Some items are divisible (food, water) and can be taken in fractions, while others are indivisible (medical kits, equipment).

As a logistics manager, your tasks are:

1. Implement the Fractional Knapsack algorithm.
2. Prioritize high-utility items while considering the boat’s weight limit.
3. Allow partial selection of divisible items.
4. Ensure the boat carries maximum possible critical supplies within W.

# Explanation:

The **Fractional Knapsack** problem is a classic **Greedy Algorithm** approach where:

* + Items are selected based on **utility-to-weight ratio (vi/wi)**.
  + Highest ratio items are taken first.
  + If an item cannot fit entirely, only a fraction of it is taken (if divisible).
  + Process continues until the capacity W is filled.

This ensures maximum utility value is achieved under limited capacity.

# Algorithm:

**Steps:**

1. Input the number of items n and boat capacity W.
2. For each item, read weight wi and utility vi.
3. Compute the ratio vi/wi for each item.
4. Sort items in descending order of ratio.
5. Initialize total value = 0.
6. For each item:
   * If the item can fit entirely → add full weight and value.
   * Else → take fraction of item that fits into remaining capacity.
7. Stop when weight limit W is reached.
8. Output maximum utility value and list of selected items.

# Pseudocode:

struct Item {

int weight; int value;

};

double fractionalKnapsack(int W, Item arr[], int n) {

// Compute ratio and sort

sort(arr, arr+n, by value/weight ratio descending);

int currentWeight = 0; double finalValue = 0.0;

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

if (currentWeight + arr[i].weight <= W) {

// Take whole item currentWeight += arr[i].weight; finalValue += arr[i].value;

}

else {

// Take fraction of item

int remain = W - currentWeight;

finalValue += arr[i].value \* ((double) remain / arr[i].weight); break;

}

}

return finalValue;

}

# Time Complexity:

* Sorting step: **O(n log n)**
* Iteration over items: **O(n)**
* **Total:** O(n log n)

# Space Complexity:

* O(1) (in-place, except for sorting overhead).

# Questions:

# Why is the Fractional Knapsack algorithm suitable for solving the flood relief supply distribution problem, and how does it differ from the 0/1 Knapsack approach?

# The Fractional Knapsack algorithm is highly suitable for this problem because the scenario explicitly involves divisible items like food and water. This aligns perfectly with the algorithm's core strength, which is the ability to take fractions of items to completely fill the capacity and maximize value.

# The key difference from the 0/1 Knapsack problem lies in this divisibility:

# Fractional Knapsack: Items can be split. You can take any fraction of an item. This allows for a greedy approach that always yields the optimal solution.

# 0/1 Knapsack: Items are indivisible. You must either take an entire item (1) or leave it (0). This constraint makes the problem more complex, and a simple greedy approach does not guarantee an optimal solution. It typically requires dynamic programming to solve.

# For emergency relief, where you want to maximize the utility of supplies like grain or water without being restricted to pre-packaged units, the fractional approach is more realistic and effective.

# Explain how the utility-to-weight ratio (vi/wi) guides the greedy selection process in the Fractional Knapsack algorithm, and why this ensures an optimal solution.

# The utility-to-weight ratio (vi/wi) represents the "value density" or "utility per kilogram" for each item. This ratio is the cornerstone of the greedy strategy.

# How it guides selection: The algorithm calculates this ratio for every item and sorts them in descending order. It then iterates through the sorted list, picking the item with the highest ratio first. This means it always prioritizes the item that provides the most utility for each unit of weight it occupies. It continues selecting the next-highest ratio item until the boat's capacity is full.

# Why it ensures optimality: This greedy strategy is guaranteed to be optimal for the *Fractional* Knapsack problem. By always filling the remaining capacity with the most value-dense item available, you ensure that no other combination of items could yield a higher total utility. If you were to replace a fraction of a high-ratio item with a fraction of a lower-ratio item, the total utility value would inevitably decrease, proving that the greedy choice at each step leads to the global optimum.

**Conclusion:** The Fractional Knapsack algorithm provides an efficient greedy solution for the emergency relief supply distribution problem. By prioritizing items with the highest utility-to-weight ratio and allowing fractional selection for divisible goods, the algorithm ensures maximum utility within the boat’s limited capacity. This makes it highly effective for disaster management scenarios where both time and resources are critical.

**Code:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;

struct Item {

string name;

double weight;

double value;

bool divisible;

int priority;

Item(string n, double w, double v, bool d, int p)

: name(n), weight(w), value(v), divisible(d), priority(p) {}

double valuePerWeight() const {

return value / weight;

}

};

// Sort by priority first, then value per weight

bool compare(const Item& a, const Item& b) {

if (a.priority == b.priority)

return a.valuePerWeight() > b.valuePerWeight();

return a.priority < b.priority;

}

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

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

cout << "\nSorted Items (by Priority, then Value/Weight):\n";

cout << left << setw(20) << "Item"

<< setw(10) << "Weight"

<< setw(10) << "Value"

<< setw(12) << "Priority"

<< setw(15) << "Value/Weight"

<< setw(15) << "Type" << "\n";

for (const auto& item : items) {

cout << left << setw(20) << item.name

<< setw(10) << item.weight

<< setw(10) << item.value

<< setw(12) << item.priority

<< setw(15) << fixed << setprecision(2) << item.valuePerWeight()

<< setw(15) << (item.divisible ? "Divisible" : "Indivisible")

<< "\n";

}

double totalValue = 0.0;

totalWeightCarried = 0.0;

cout << "\nItems selected for transport:\n";

for (const auto& item : items) {

if (capacity <= 0) break;

if (item.divisible) {

double takenWeight = min(item.weight, capacity);

double takenValue = item.valuePerWeight() \* takenWeight;

totalValue += takenValue;

capacity -= takenWeight;

totalWeightCarried += takenWeight;

cout << " - " << item.name << ": " << takenWeight << " kg, Utility = " << takenValue

<< ", Priority = " << item.priority << ", Type = Divisible\n";

} else {

if (item.weight <= capacity) {

totalValue += item.value;

capacity -= item.weight;

totalWeightCarried += item.weight;

cout << " - " << item.name << ": " << item.weight << " kg, Utility = " << item.value

<< ", Priority = " << item.priority << ", Type = Indivisible\n";

}

}

}

return totalValue;

}

int main() {

vector<Item> items = {

Item("Medical Kits", 10, 100, false, 1),

Item("Food Packets", 20, 60, true, 3),

Item("Drinking Water", 30, 90, true, 2),

Item("Blankets", 15, 45, false, 3),

Item("Infant Formula", 5, 50, false, 1)

};

double capacity;

cout << "Enter maximum weight capacity of the boat (in kg): ";

cin >> capacity;

double totalWeightCarried;

double maxValue = fractionalKnapsack(items, capacity, totalWeightCarried);

cout << "\n===== Final Report =====\n";

cout << "Total weight carried: " << fixed << setprecision(2) << totalWeightCarried << " kg\n";

cout << "Total utility value carried: " << fixed << setprecision(2) << maxValue << " units\n";

return 0;

}

**Output:**

Enter maximum weight capacity of the boat (in kg): 40

Sorted Items (by Priority, then Value/Weight):

Item Weight Value Priority Value/Weight Type

Medical Kits 10 100 1 10.00 Indivisible

Infant Formula 5 50 1 10.00 Indivisible

Drinking Water 30 90 2 3.00 Divisible

Food Packets 20 60 3 3.00 Divisible

Blankets 15 45 3 3.00 Indivisible

Items selected for transport:

- Medical Kits: 10 kg, Utility = 100, Priority = 1, Type = Indivisible

- Infant Formula: 5 kg, Utility = 50, Priority = 1, Type = Indivisible

- Drinking Water: 25 kg, Utility = 75, Priority = 2, Type = Divisible

===== Final Report =====

Total weight carried: 40.00 kg

Total utility value carried: 225.00 units