### 1) **Realistic Applications of Fractional Knapsack Problem**

Here are two realistic applications of the **fractional knapsack problem**:

#### a) **Resource Allocation in Business or Manufacturing:**

In many business scenarios, companies have a limited amount of resources, such as raw materials, funds, or time, and need to allocate them to various projects to maximize profit. Each project requires a certain amount of resources (weight), and it provides a certain profit (value). By using the **fractional knapsack problem**, businesses can determine how to allocate resources efficiently by selecting the most profitable projects with the best value-to-weight ratio, even if they need to partially allocate resources to projects (i.e., fractional amounts).

Example:

A company has a total of 100 units of raw material and a set of products with varying profit potential and material usage. The company can either produce a full amount of a product or a fractional part of it, so they select the products with the highest **profit-to-weight ratio** (profit per unit of material used).

#### b) **Cargo Loading in Shipping or Transport:**

When transporting goods, the capacity of a vehicle (like a truck, cargo plane, or ship) is limited, and different goods have different values and weights. The **fractional knapsack** problem can be applied to determine how to load the vehicle with goods to maximize total value while staying within the weight limit. The goods can be loaded as full units or fractional units (e.g., part of a container, or quantity of goods).

Example:

A cargo plane has a weight limit of 1,000 kg and a list of goods with varying values and weights. The goal is to maximize the total value of the cargo by selecting the goods with the highest **value-to-weight ratio**, including partial amounts if necessary.

### 2) **Explanation of the Knapsack Problem Using a Greedy Approach**

Let's explain the **fractional knapsack problem** with an example using the **greedy approach**.

#### Example Data:

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Capacity of Knapsack (W) = 50

#### **Step 1: Calculate the Value-to-Weight Ratio (V/W) for Each Item**

For each item, calculate the **value-to-weight ratio** \(\frac{Value}{Weight} \).

Item	Weight	Value	Value-to-Weight Ratio (V/W)
1	5	30	30/5 = 6
2	10	40	40/10 = 4
3	15	45	45/15 = 3
4	22	77	77/22 = 3.5
5	25	90	90/25 = 3.6

#### **Step 2: Sort Items in Decreasing Order of the Value-to-Weight Ratio**

Sort the items in descending order of the value-to-weight ratio:

Item	Weight	Value	Value-to-Weight Ratio (V/W)
1	5	30	6
5	25	90	3.6
4	22	77	3.5
2	10	40	4
3	15	45	3

#### **Step 3: Initialize Variables**

Set the **current capacity** of the knapsack to the total capacity \(W = 50 \), and initialize the total value \(res = 0 \).

#### **Step 4: Select Items**

Now, iterate through the sorted items and select them greedily:

- **Item 1**: Weight = 5, Value = 30, **Ratio = 6**.

  - Since the remaining capacity (50) is greater than or equal to the weight of Item 1, we add it entirely.

  - **Remaining capacity** = 50 - 5 = 45.

  - **Total value (res)** = 0 + 30 = 30.

- **Item 5**: Weight = 25, Value = 90, **Ratio = 3.6**.

  - Since the remaining capacity (45) is greater than or equal to the weight of Item 5, we add it entirely.

  - **Remaining capacity** = 45 - 25 = 20.

  - **Total value (res)** = 30 + 90 = 120.

- **Item 4**: Weight = 22, Value = 77, **Ratio = 3.5**.

  - The remaining capacity (20) is less than the weight of Item 4 (22), so we can only add a fraction of this item.

  - The fraction of Item 4 that we can add is \(\frac{20}{22} = 0.909 \).

  - **Value added** = 77 * 0.909 = 70.09.

  - **Remaining capacity** = 20 - 20 = 0.

  - **Total value (res)** = 120 + 70.09 = 190.09.

At this point, the knapsack is full.

#### **Step 5: Return the Total Value**

The total value of the items in the knapsack is approximately **190.09**.

### 3) **Time Complexity of the Knapsack Problem (Greedy Approach)**

- **Step 1 (Calculate the value-to-weight ratio for each item)**: This takes \(O(n) \) time where \(n \) is the number of items.

- **Step 2 (Sorting the items by ratio)**: The sorting operation takes \(O(n \log n) \) time.

- **Step 3 (Greedily picking items)**: This step requires iterating through the sorted list of items, which takes \(O(n) \) time.

Thus, the overall time complexity of the **fractional knapsack problem** using the **greedy approach** is:

\[

O(n \log n)

\]

This is primarily dominated by the sorting step. The greedy selection and value calculation are linear in complexity.

……………………code explanation…………………….

Let's go through the **Fractional Knapsack Problem** code step by step and explain each part in detail:

### 1. **Header Files and `Item` Struct**

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct Item {
    int weight;
    int value;
    float ratio;
};
```

- **`#include <iostream>`**: Includes the standard input-output stream library to use functions like `cout` and `cin`.

- **`#include <vector>`**: Includes the vector library to store the items in a dynamic array.

- **`#include <algorithm>`**: Includes the algorithm library to use functions like `sort()`.

The `Item` structure is used to store the **weight**, **value**, and **value-to-weight ratio** of each item.

### 2. **Comparison Function: `compare`**

```cpp
bool compare(Item a, Item b) {
    return a.ratio > b.ratio;
}
```

- This is a custom comparison function to sort the items based on the **value-to-weight ratio** (ratio).

- The function returns `true` if **Item `a`'s ratio is greater than Item `b`'s ratio**, which allows sorting items in **descending order** based on their ratio.

### 3. **Main Algorithm: `fractionalKnapsack`**

```cpp
float fractionalKnapsack(int W, vector<Item>& items) {
    sort(items.begin(), items.end(), compare);  // Sorting items by ratio


    int currWeight = 0;
    float totalValue = 0.0;
```

- **Sorting Items**: The `sort()` function sorts the items in descending order of their value-to-weight ratio using the custom `compare` function.

- **`currWeight`**: Initializes the current weight of the knapsack to 0. This will keep track of the weight added to the knapsack.

- **`totalValue`**: Initializes the total value of the knapsack to 0. This will keep track of the total value of items in the knapsack.

```cpp
    cout << "Item   Weight   Value   Ratio     Taken\n";
    cout << "----------------------------------------\n";
```

- Prints the header of the output table to display item details including its weight, value, ratio, and whether it was taken fully or partially.

#### Iterating over Items

```cpp
    for (int i = 0; i < items.size(); i++) {
        if (currWeight + items[i].weight <= W) {
```

- This loop iterates through all the items sorted by their value-to-weight ratio.
- **`currWeight + items[i].weight <= W`** checks if the item can be added fully to the knapsack without exceeding the capacity. If the item can be fully added:

```cpp
            currWeight += items[i].weight;  // Add the item's weight to the knapsack
            totalValue += items[i].value;   // Add the item's value to the total value
```

- **`currWeight += items[i].weight`** updates the current weight of the knapsack after adding the item.
- **`totalValue += items[i].value`** updates the total value by adding the full value of the item.

```cpp
            cout << i + 1 << "      " << items[i].weight << "        "
                 << items[i].value << "       "
                 << items[i].ratio << "      Fully\n";
```

- Prints the item details: item index, weight, value, ratio, and that it was taken fully.

#### If the Item Cannot Be Fully Added

```cpp
        } else {
            int remainingWeight = W - currWeight;  // Calculate the remaining capacity
            float fraction = (float)remainingWeight / items[i].weight;  // Calculate the fraction of the item that can be taken
            totalValue += items[i].value * fraction;  // Add the fraction of the item’s value to total value
```

- If the current item cannot be fully added to the knapsack (because adding it would exceed the capacity), we calculate how much of the item can be added:

  - **`remainingWeight = W - currWeight`** calculates the remaining capacity of the knapsack.

  - **`fraction = (float)remainingWeight / items[i].weight`** calculates the fraction of the item that can fit into the knapsack.

  - **`totalValue += items[i].value * fraction`** adds the fraction of the item’s value to the total value.

```cpp
        cout << i + 1 << "      " << items[i].weight << "        "
            << items[i].value << "       "
            << items[i].ratio << "      Partially (" << (fraction * 100) << "%)\n";
        break;  // Exit loop as we cannot add any more items
```

- The details of the partially taken item are printed, including the percentage of the item taken, and the loop is exited because the knapsack is full.

```cpp
    }

    return totalValue;  // Return the total value accumulated
}
```

- After processing all items or exiting the loop, the function returns the **total value** of the knapsack.

### 4. **`main()` Function**

```cpp
int main() {
    int n, W;

    cout << "Enter the number of items: ";
    cin >> n;  // Input the number of items
    cout << "Enter the capacity of the knapsack: ";
    cin >> W;  // Input the capacity of the knapsack
```
- The user is asked to input the number of items (`n`) and the capacity of the knapsack (`W`).

```cpp
    vector<Item> items(n);  // Create a vector to store items

    for (int i = 0; i < n; i++) {
        cout << "Enter weight and value for item " << i + 1 << ": ";
        cin >> items[i].weight >> items[i].value;

        items[i].ratio = (float)items[i].value / items[i].weight;  // Calculate the value-
to-weight ratio
    }
```
- A vector `items` of size `n` is created to store the items.
- For each item, the user inputs its weight and value, and the value-to-weight ratio is computed and stored in `items[i].ratio`.

```cpp
    cout << "\nTable of Items (sorted by value-to-weight ratio):\n";
    float maxValue = fractionalKnapsack(W, items);  // Call the fractional knapsack function
    cout << "\nMaximum value we can obtain = " << maxValue << endl;  // Output the total value of knapsack
}
```

- A table of items is printed, sorted by the value-to-weight ratio.

- The `fractionalKnapsack` function is called to compute the maximum value the knapsack can hold, and the result is displayed.

### Summary

- The program solves the **Fractional Knapsack Problem** using a **greedy strategy**.

- It first calculates the **value-to-weight ratio** of each item, sorts the items by this ratio in descending order, and then adds items to the knapsack, either fully or partially, to maximize the total value.

- The program prints the details of each item, including how much of it is taken (fully or partially) and the total value achieved.

……………………Applications……………………………….

Here are some **realistic applications** of the **Fractional Knapsack Problem** and how the greedy approach is used:

### 1. **Resource Allocation in Projects**

- **Problem**: Imagine a scenario where you have several tasks to be completed in a project. Each task has a certain time to complete (weight) and a certain value it brings to the project (value). You have limited resources (e.g., time or workforce) to allocate to the tasks.

- **Application**: The fractional knapsack algorithm can be used to allocate resources efficiently. You would compute the "value per unit of resource" for each task (similar to value/weight ratio) and prioritize tasks with higher value per unit resource. This way, you can maximize the total value of the project by taking as much as possible from the tasks with the highest ratio, and if needed, completing only part of the tasks that fit into the available resources.

### 2. **Cargo Loading and Shipping**

- **Problem**: Suppose you are managing a fleet of trucks or shipping containers, and you need to transport various items. Each item has a certain weight and value (such as in a shipping or warehouse context). The truck/container has a limited capacity (weight limit), and you need to maximize the value of the cargo you load into it.

- **Application**: By applying the fractional knapsack approach, you can determine the optimal items to pack in the truck/container by evaluating the **value-to-weight ratio** of each item. The items with the highest ratio will be prioritized for full inclusion, and if needed, items can be taken in fractions (for example, only part of an item’s weight can be loaded). This ensures maximum value within the capacity constraints.

### 3. **Investment Portfolio Optimization**

- **Problem**: In investment portfolio management, each investment option has a certain cost (weight) and an expected return (value). The goal is to maximize returns while staying within a fixed budget.

- **Application**: You can treat this as a fractional knapsack problem where:

- The **weights** represent the amount of money invested in each asset.

- The **values** represent the expected returns or profits from those investments.

- The **capacity** is the available investment budget.

- By calculating the **value-to-weight ratio** (return per unit of investment), you can sort the investments by this ratio and prioritize those with the highest return per dollar. If there is leftover budget after investing in high-value options, you can invest in fractions of remaining options that fit the budget.

### 4. **Advertisement Space Allocation**

- **Problem**: You are responsible for allocating advertising space on a website or a physical billboard. Each ad occupies a certain space (weight) and generates a specific revenue (value). You have a limited amount of space available and need to maximize the revenue from ads.

- **Application**: The fractional knapsack approach can help decide which ads to place and in what proportion. The **value-to-space ratio** for each ad can be computed and sorted, allowing you to maximize the total revenue by allocating full space to the ads with the highest ratio and partial space to others as needed.

### 5. **Production Line and Manufacturing Optimization**

- **Problem**: In a manufacturing setup, different parts or materials are required to manufacture products, and each material has its own cost (weight) and profit margin (value). The factory has a limited production capacity or material budget, and you want to maximize the profit.

- **Application**: By applying the fractional knapsack strategy, the materials with the highest **profit-to-cost ratio** can be prioritized. The factory can use a combination of full materials and fractional parts of materials to maximize profit while staying within the production capacity.

### 6. **Data Compression**

- **Problem**: In data compression techniques, data elements need to be stored in a way that minimizes space (e.g., disk or memory usage) while preserving data integrity. Some data elements are more valuable (or more frequent) than others and need to be prioritized.

- **Application**: A **fractional knapsack** model can help determine how to allocate storage space. For example, in the context of **Huffman coding**, where symbols are assigned binary codes based on their frequency, the problem is similar to choosing which data chunks to compress or store based on their relative "importance" or frequency.

### 7. **Emergency Resource Distribution**

- **Problem**: In disaster management or emergency response, there is often a need to allocate limited resources (such as food, medicine, and other supplies) across various affected areas. Each area has a different need (value) and a different demand (weight, i.e., quantity of supplies).

- **Application**: The fractional knapsack algorithm can help by calculating the **value per unit of resource** for each area or need. Resources would then be distributed in a way that maximizes overall relief, giving priority to areas with the highest need-to-resource ratio, and supplying them fully or partially as needed.

### Key Takeaways:

The fractional knapsack problem and greedy approach are widely applicable in fields where resources are limited, and the goal is to maximize total value or benefit. The core idea of **value-to-weight ratio** helps in selecting the most efficient combination of resources, items, or investments, even when fractional usage is allowed.