Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058, India
(Autonomous College Affiliated to University of Mumbai)

Experiment No.	5
Aim	Implement fractional knapsack problem
Name	Aarush Kinhikar
UID No.	2021300063
Class & Division	SE Comps A, Batch D

Theory:

The knapsack problem is a classic optimization problem in computer science and mathematics. It can be stated as follows: Given a set of items, each with a weight and a value, and a knapsack with a limited capacity, the goal is to select a subset of items to maximize the total value of items in the knapsack, subject to the constraint that the total weight of items in the knapsack does not exceed the knapsack's capacity.

The knapsack problem is typically classified into two main types:

1. 0/1 Knapsack Problem: In this variation, each item can be either included in the knapsack or excluded from it. In other words, items cannot be divided or split into fractions, and only whole items can be selected or not selected.

2. Fractional Knapsack Problem: In this variation, items can be divided or split into fractions to fill the knapsack. Each item has a weight and a value, and a fraction of an item can be included in the knapsack based on its value-to-weight ratio.

The knapsack problem has a wide range of real-world applications, such as resource allocation, project selection, scheduling, and portfolio optimization, among others. It is also used as a benchmark problem in computer algorithms and optimization research due to its combinatorial nature and various solution techniques that can be applied to solve it.

Algorithm:

1. Read input: number of items, weight and value of each item, and knapsack capacity.
2. Compute the value-to-weight ratio for each item by dividing value by weight.
3. Sort the items based on their value-to-weight ratio in descending order.
4. Initialize total value and current weight in the knapsack to 0.
5. Loop through each item from highest value-to-weight ratio to lowest:
    a. If the current item can be fully included in the knapsack, add its value to total value and increment current weight by its weight.
    b. If the current item cannot be fully included, calculate the fraction that can be included based on the remaining capacity of the knapsack, add the fraction's value to total value, and break out of the loop.
6. Print the total value as the maximum value of items in the knapsack.

Program:

```c
#include <stdio.h>
#include <stdlib.h>

// Structure to represent an item in the knapsack
struct Item {
    int weight;     // weight of the item
    int value;      // value of the item
};

// Function to compare items based on their value-to-weight ratio
int compareItems(const void *a, const void *b) {
    struct Item *itemA = (struct Item *)a;
    struct Item *itemB = (struct Item *)b;
    double ratioA = (double)itemA->value / itemA->weight;
    double ratioB = (double)itemB->value / itemB->weight;
    if (ratioA < ratioB) {
        return 1;
    } else if (ratioA > ratioB) {
        return -1;
    } else {
        return 0;
    }
}

// Function to solve the fractional knapsack problem
```

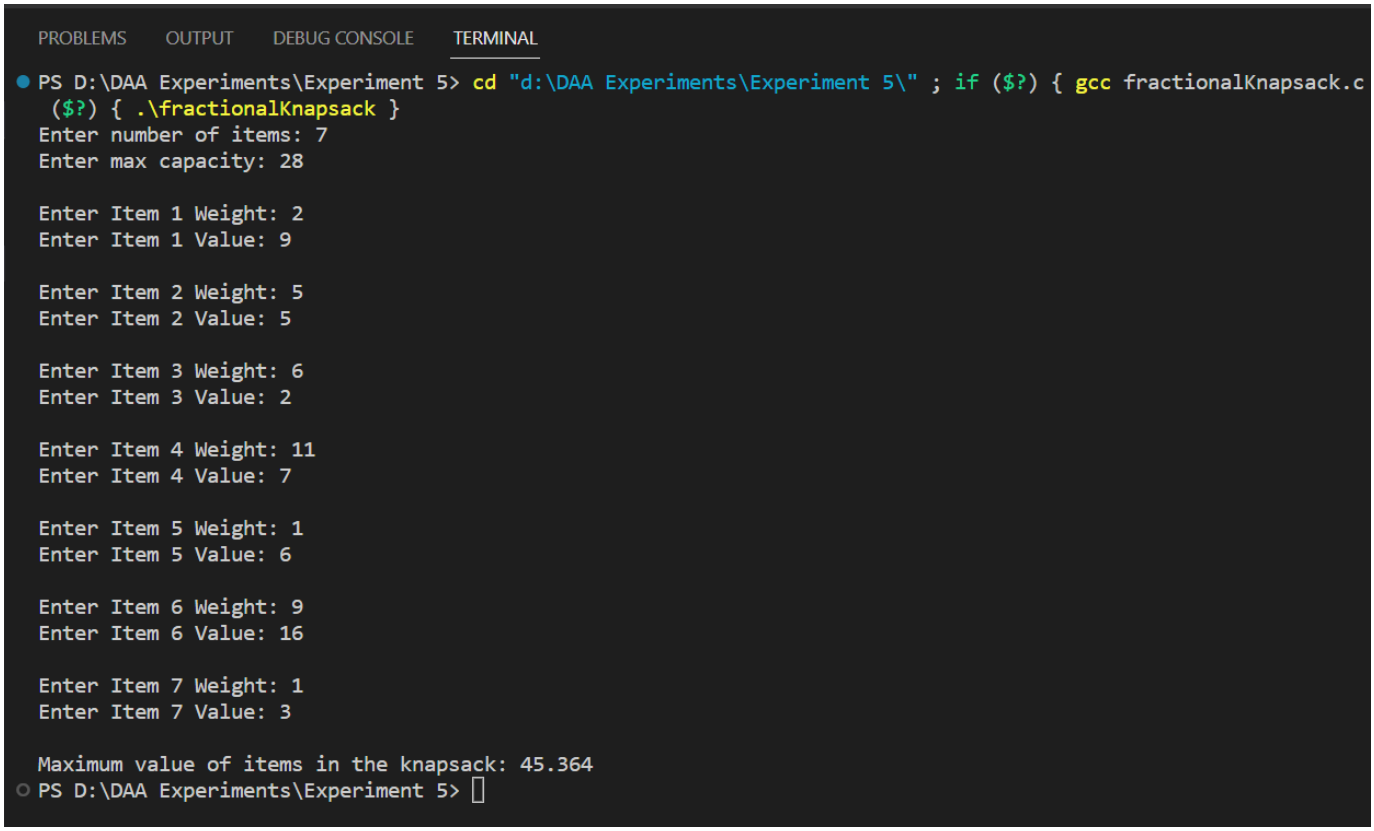
```c
double fractionalKnapsack(struct Item items[], int n, int capacity) {
    // Sort items based on their value-to-weight ratio in descending order
    qsort(items, n, sizeof(struct Item), compareItems);

    double totalValue = 0;    // Total value of items in the knapsack
    int currentWeight = 0;    // Current weight of items in the knapsack

    for (int i = 0; i < n; i++) {
        // If the item can be fully included in the knapsack
        if (currentWeight + items[i].weight <= capacity) {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        } else {
            // If the item cannot be fully included, calculate the fraction
            double fraction = (double)(capacity - currentWeight) /
items[i].weight;
            totalValue += fraction * items[i].value;
            break;
        }
    }
    return totalValue;
}

//main function
int main() {
    int i,n,capacity;
    printf("Enter number of items: ");
    scanf("%d",&n);
    printf("Enter max capacity: ");
    scanf("%d",&capacity);
    printf("\n");
    struct Item items[n];
    for(i=0;i<n;i++){
        printf("Enter Item %d Weight: ",(i+1));
        scanf("%d",&items[i].weight);
        printf("Enter Item %d Value: ",(i+1));
        scanf("%d",&items[i].value);
        printf("\n");
    }
    double maxValue = fractionalKnapsack(items, n, capacity);
    printf("Maximum value of items in the knapsack: %.3lf\n", maxValue);
    return 0;
}
```

Results:



Conclusion:

The greedy algorithm presented above is a simple and commonly used approach for solving the Fractional Knapsack Problem. By selecting items based on their value-to-weight ratio in descending order, the algorithm ensures that the most valuable items are included in the knapsack first, leading to an approximate optimal solution.

The knapsack problem is NP-complete, meaning that there is no known efficient algorithm to solve it optimally for large problem sizes. However, greedy algorithms, dynamic programming, and other techniques can be used to find approximate solutions or solve smaller instances of the problem.