Vidyavardhini's College of Engineering and Technology

Department of Artificial Intelligence & Data Science

AY: 2024-25

Class:	**SE**	**Semester:**	**IV**
Course Code:	**CSL401**	**Course Name:**	**Analysis of Algorithm Lab**

Name of Student:	**Shravani Sandeep Raut**
Roll No. :	**48**
Experiment No.:	**5**
Title of the Experiment:	**Fractional Knapsack using Greedy Method**
Date of Performance:	**06/02/2025**
Date of Submission:	**13/02/2025**

Evaluation

Performance Indicator	Max. Marks	Marks Obtained
Performance	5	
Understanding	5	
Journal work and timely submission	10	
Total	20	

Performance Indicator	Exceed Expectations (EE)	Meet Expectations (ME)	Below Expectations (BE)
Performance	4-5	2-3	1
Understanding	4-5	2-3	1
Journal work and timely submission	8-10	5-8	1-4

Checked by

Name of Faculty : Mrs. Sneha Yadav

Signature :

Date:

Experiment No. 5

Title: Fraction Knapsack

Aim: To study and implement Fractional Knapsack Algorithm

Objective: To introduce Greedy based algorithms

Theory:

Greedy method or technique is used to solve Optimization problems. A solution that can be maximized or minimized is called Optimal Solution.

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed size knapsack and must fill it with the most valuable items. The most common problem being solved is the 0-1 knapsack problem, which restricts the number xi of copies of each kind of item to zero or one.

In Knapsack problem we are given:1) n objects 2) Knapsack with capacity m, 3) An object i is associated with profit Wi , 4) An object i is associated with profit Pi , 5) when an object i is placed in knapsack we get profit Pi Xi .

Here objects can be broken into pieces (Xi Values) The Objective of Knapsack problem is to maximize the profit.

Example:

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.

$$0 \leqslant xi \leqslant 1$$

The i^{th} item contributes the weight xi.wi to the total weight in the knapsack and profit xi.pi to the total profit**.**

```
greedy_fractional_knapsack (w[1...n], p[1...n], W)

for i=1 to n
    do x[i] = 0
weight = 0
for i=1 to n
    if weight + w[i] ≤ W then
        x[i] = 1
        weight = weight + w[i]
    else
        x[i] = (W-weight) / w[i]
        weight = W
        break
return x
```

i=1 → B
0+10 ≤ 60
x[i] = 1
wt = 10

i=2 → A
10+40
50 ≤ 60
x[i] = 2
10+40
wt = 50

i=3 → C
(60-50)/20
10/20 = 1/2
x[i] = 1/2
wt = 60

x = [A, B, $\frac{1}{2}$C]

~~x[i] = 0~~
~~wt = 0~~

Total profit is
100+280+120*(10/20)
380+60 = 440

Total wt
10+40+20*(10/20)
= 60

Ex: W = 60

Item	A	B	C	D
profit	280	100	120	120
weight	40	10	20	24
Ratio $\left(\frac{p_i}{w_i}\right)$	7	10	6	5

provided items are not sorted based on $\frac{p_i}{w_i}$.

sorted

Item	B	A	C	D
profit	100	280	120	120
weight	10	40	20	24
Ratio $\left(\frac{p_i}{w_i}\right)$	10	7	6	5

Algorithm:

Hence, the objective of this algorithm is to

$$maximize \sum_{n=1}^{n}(x_i . pi)$$

subject to constraint,

$$\sum_{n=1}^{n}(x_i . wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\boxed{\sum_{n=1}^{n}(x_i . wi) = W}$$

In this context, first we need to sort those items according to the value of $\frac{p_i}{w_i}$, so that $\frac{p_i+1}{w_i+1}$ ≤ $\frac{p_i}{w_i}$. Here, ***x*** is an array to store the fraction of items.

```
Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)
for i = 1 to n
   do x[i] = 0
weight = 0
for i = 1 to n
   if weight + w[i] ≤ W then
      x[i] = 1
      weight = weight + w[i]
   else
      x[i] = (W - weight) / w[i]
      weight = W
      break
return x
```

Implementation:

```
#include <stdio.h>

// Structure to store item details
struct Item {
    int weight;
    int value;
    double ratio;
};

void mergeSort(struct Item arr[], int left, int right);
void merge(struct Item arr[], int left, int mid, int right);
void fractionalKnapsack(const struct Item items[], int n, int capacity);

int main() {
    int n, capacity;

    printf("Enter number of items: ");
    scanf("%d", &n);

    struct Item items[n];  // Dynamic size based on input

    printf("Enter weight and value of each item:\n");
    for(int i = 0; i < n; i++) {
        printf("Enter weight and value: ");
        scanf("%d %d", &items[i].weight, &items[i].value);
        items[i].ratio = (double)items[i].value / items[i].weight;
    }

    printf("Enter knapsack capacity: ");
    scanf("%d", &capacity);

    fractionalKnapsack(items, n, capacity);

    return 0;
}

// Function to solve Fractional Knapsack
```

```
void fractionalKnapsack(const struct Item items[], int n, int capacity) {
    struct Item sortedItems[n];

    // Copy original array to avoid modifying input
    for (int i = 0; i < n; i++) {
        sortedItems[i] = items[i];
    }

    mergeSort(sortedItems, 0, n - 1);

    double totalValue = 0.0;

    printf("\nItems taken in the knapsack:\n");

    for(int i = 0; i < n; i++) {
        if(capacity >= sortedItems[i].weight) {
            printf("Item with weight %d and value %d taken fully.\n",
sortedItems[i].weight, sortedItems[i].value);
            capacity -= sortedItems[i].weight;
            totalValue += sortedItems[i].value;
        } else {
            double fraction = (double)capacity / sortedItems[i].weight;
            printf("Item with weight %d and value %d taken %.2f fraction.\n",
sortedItems[i].weight, sortedItems[i].value, fraction);
            totalValue += sortedItems[i].value * fraction;
            break;
        }
    }

    printf("Maximum value in knapsack: %.2f\n", totalValue);
}

// Merge Sort function
void mergeSort(struct Item arr[], int left, int right) {
    if(left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
```

```
    }
}

// Merge function for sorting by value/weight ratio
void merge(struct Item arr[], int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;

    struct Item L[n1], R[n2];

    for(int i = 0; i < n1; i++)
        L[i] = arr[left + i];

    for(int i = 0; i < n2; i++)
        R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;

    while(i < n1 && j < n2) {
        if(L[i].ratio >= R[j].ratio)
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }

    while(i < n1)
        arr[k++] = L[i++];

    while(j < n2)
        arr[k++] = R[j++];
}
```

```
Enter number of items: 4
Enter weight and value of each item:
Enter weight and value: 40 280
Enter weight and value: 10 100
Enter weight and value: 20 120
Enter weight and value: 24 120
Enter knapsack capacity: 60

Items taken in the knapsack:
Item with weight 10 and value 100 taken fully.
Item with weight 40 and value 280 taken fully.
Item with weight 20 and value 120 taken 0.50 fraction.
Maximum value in knapsack: 440.00
```

Conclusion:

The greedy method is a powerful approach in solving optimization problems like the fractional knapsack problem. In this variation, items can be divided, allowing a thief to take fractions of the most valuable goods. The goal is to maximize profit without exceeding the knapsack's weight capacity. By calculating the profit-to-weight ratio for each item and selecting the highest ratios first, the greedy algorithm ensures an optimal solution. This approach is both efficient and intuitive, making it a popular technique in computer science for problems requiring quick, near-optimal decisions. The fractional knapsack exemplifies how strategic thinking leads to the best outcomes.