| |
|---|
| Experiment No. 5 |
| Fractional Knapsack using Greedy Method |
| Date of Performance: |
| Date of Submission: |

## Experiment No. 5

**Title:** Fraction Knapsack

**Aim:** To study and implement Fraction 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
x[i] = 1/2    10/20 = 1/2
wt = 60

x = [A, B, 1/2 C]

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

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

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

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 represent items
struct Item {
    int value;
    int weight;
};


// Function to compare items based on their value per unit weight
int compare(const void *a, const void *b) {
    double ratio1 = (double)(((struct Item*)a)->value) / (((struct Item*)a)->weight);
    double ratio2 = (double)(((struct Item*)b)->value) / (((struct Item*)b)->weight);
```

```
    if (ratio1 < ratio2)
        return 1;
    else if (ratio1 > ratio2)
        return -1;
    else
        return 0;
}

// Function to solve fractional knapsack problem
void fractionalKnapsack(struct Item arr[], int n, int capacity) {
    // Sort items based on value per unit weight
    qsort(arr, n, sizeof(arr[0]), compare);

    int currentWeight = 0; // Current weight in knapsack
    double finalValue = 0.0; // Final value of items selected

    for (int i = 0; i < n; i++) {
        // If adding the current item won't overflow the knapsack
        if (currentWeight + arr[i].weight <= capacity) {
            currentWeight += arr[i].weight;
            finalValue += arr[i].value;
        }
        else {
```

```
        // Otherwise, add a fraction of the item to fill the knapsack
        int remainingWeight = capacity - currentWeight;
        finalValue += arr[i].value * ((double)remainingWeight / arr[i].weight);
        break; // Knapsack is full
      }
    }

    printf("Maximum value in the knapsack: %.2lf\n", finalValue);
}

int main() {
    struct Item arr[] = {{60, 10}, {100, 20}, {120, 30}};
    int n = sizeof(arr) / sizeof(arr[0]);
    int capacity = 50;
    fractionalKnapsack(arr, n, capacity);
    return 0;
}
```

**Conclusion:** Fractional Knapsack algorithm has been successfully implemented.