



Experiment No. : 4

Title: Implement Fractional Knapsack using Greedy approach

Batch:SY-IT(B3)

Roll No.: 16010423076

Experiment No.:4

Aim: To Implement Fractional Knapsack using Greedy approach and analyse its time Complexity.

Algorithm of Fractional Knapsack:

Here's how you can solve this problem step by step:

1. Understand the Items:
 - Each item has a weight and a value.
 - Calculate the "value per unit weight" for each item. This is done by dividing the value of the item by its weight. For example, if an item is worth
 - If an item has a total value of 60 and weighs 10 kg, its value per unit weight is 6 per kg.
2. Sort the Items:
 - Arrange all the items in order of their value per unit weight, starting with the highest. This way, you prioritize the items that give you the most value for the least weight.
3. Fill the Knapsack:
 - Start adding items to your backpack, beginning with the one with the highest value per unit weight.
 - If an item fits entirely into the backpack, add it completely and subtract its weight from the total capacity of the backpack.
 - If an item doesn't fit entirely, take as much of it as you can (a fraction of it) to fill the remaining space in the backpack.
4. Stop When the Knapsack is Full:
 - Once the backpack is full, you're done! You've maximized the total value of the items in the backpack.

Explanation and Working of Fractional Knapsack:

Imagine you have a knapsack (backpack) with a limited capacity, say W kilograms, and you have n different items to choose from. Each item has:

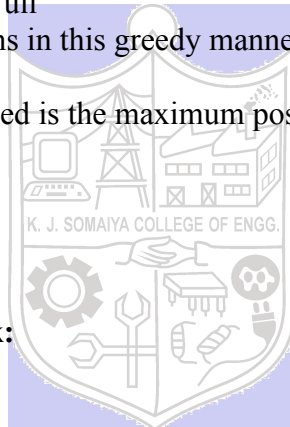
- A weight (w_1, w_2, \dots, w_n)
- A value (v_1, v_2, \dots, v_n)

Your goal is to maximize the total value of items in the knapsack without exceeding the weight limit.

The key difference in Fractional Knapsack (compared to 0/1 Knapsack) is that you are allowed to take fractions of an item, meaning you don't have to take an item entirely—you can break it into smaller portions and take only as much as needed to fill the knapsack.

Step-by-Step Working of Fractional Knapsack

1. Calculate Value-to-Weight Ratio
 - For each item, compute its value per unit weight using the formula:
ratio = value / weight
 - Example: If an item has a value of 60 and a weight of 10 kg, its value per unit weight is:
 $60/10 = 6$ (i.e. 6 per kg)
2. Sort Items by Value-to-Weight Ratio (Descending Order)
 - Arrange all items in decreasing order of their value-to-weight ratio.
 - This ensures that we pick the most valuable items first.
3. Start Filling the Knapsack
 - Begin with the item that has the highest value-to-weight ratio.
 - If the entire item fits, take it and subtract its weight from the remaining capacity.
 - If it does not fit entirely, take only a fraction of it to fill the remaining space.
4. Stop When the Knapsack is Full
 - Continue picking items in this greedy manner until the knapsack reaches its full capacity.
 - The total value obtained is the maximum possible value.



Derivation of Fractional Knapsack:

Time complexity Analysis

Sorting Step: The items are sorted based on their value-to-weight ratio, which takes $O(n \log n)$ time.

Selection Step: The algorithm iterates through the sorted list, picking items or fractions of them until the knapsack is full, which takes $O(n)$ time.

Overall Complexity: The total time complexity is $O(n \log n) + O(n) = O(n \log n)$.

Applicability: This approach works optimally for the Fractional Knapsack Problem, but not for the 0/1 Knapsack Problem, which requires dynamic programming or backtracking.

Program(s) of Fractional Knapsack:

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

```
// Structure to represent an item with weight, value, and value-to-weight ratio
typedef struct {
```

```

    int weight;
    int value;
    double ratio; // value-to-weight ratio
} Item;

// Comparator function for sorting items in descending order of ratio
int compare(const void *a, const void *b) {
    double r1 = ((Item *)a)->ratio;
    double r2 = ((Item *)b)->ratio;
    return (r2 > r1) - (r2 < r1); // Sort in descending order
}

// Function to calculate the maximum value in the fractional knapsack
double fractionalKnapsack(Item items[], int n, int capacity) {
    // Sort items based on their value-to-weight ratio
    qsort(items, n, sizeof(Item), compare);

    double maxVal = 0.0; // Maximum value obtained
    int currentWeight = 0; // Current weight in knapsack

    // Iterate through sorted items
    for (int i = 0; i < n; i++) {
        if (currentWeight + items[i].weight <= capacity) {
            // Take the full item
            currentWeight += items[i].weight;
            maxVal += items[i].value;
        } else {
            // Take a fraction of the item
            int remainingCapacity = capacity - currentWeight;
            maxVal += items[i].ratio * remainingCapacity;
            break; // Knapsack is full
        }
    }
    return maxVal;
}

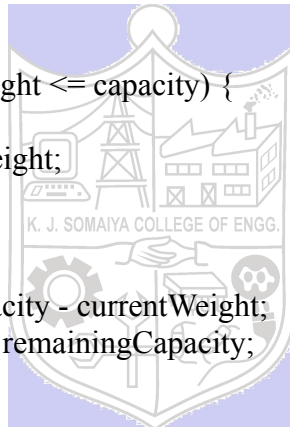
int main() {
    int n, capacity;

    // Input: Number of items and knapsack capacity
    printf("Enter the number of items: ");
    scanf("%d", &n);
    printf("Enter the knapsack capacity: ");
    scanf("%d", &capacity);

    Item items[n];

    // Input: Weights and values of items
    for (int i = 0; i < n; i++) {
        printf("Enter weight and value for item %d: ", i + 1);

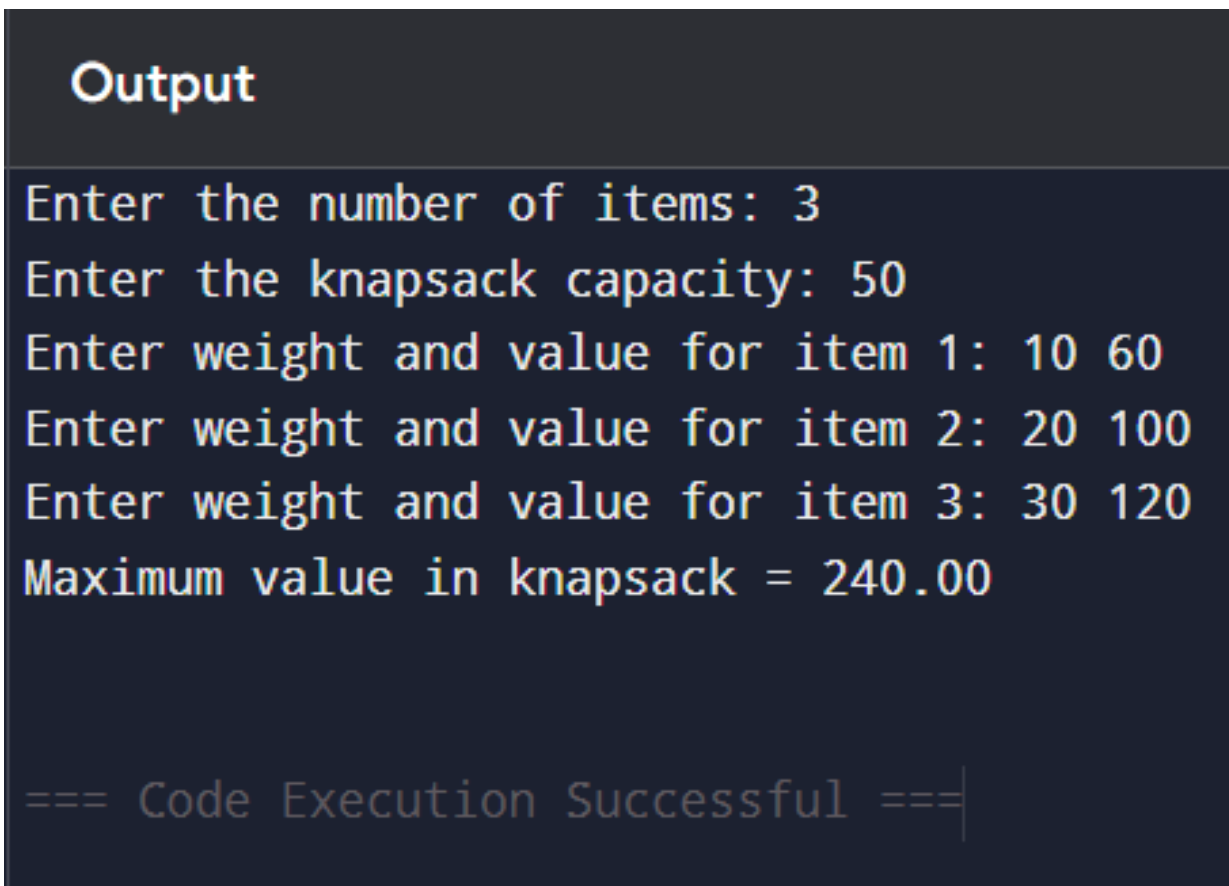
```



```
scanf("%d %d", &items[i].weight, &items[i].value);
items[i].ratio = (double)items[i].value / items[i].weight; // Calculate ratio
}

// Calculate and print the maximum value
double maxValue = fractionalKnapsack(items, n, capacity);
printf("Maximum value in knapsack = %.2f\n", maxValue);

return 0;
}
```

Output(o) of Fractional Knapsack:

```
Output

Enter the number of items: 3
Enter the knapsack capacity: 50
Enter weight and value for item 1: 10 60
Enter weight and value for item 2: 20 100
Enter weight and value for item 3: 30 120
Maximum value in knapsack = 240.00

=== Code Execution Successful ===
```

Post Lab Questions:-

Differentiate between 0/1 and Fractional Knapsack with suitable examples.

1. Item Selection

- 0/1 Knapsack: Items can either be fully taken (1) or not taken at all (0).
- Fractional Knapsack: Items can be divided, allowing fractional selection.

2. Optimal Approach

- 0/1 Knapsack: Solved using Dynamic Programming ($O(nW)$) or Backtracking.
- Fractional Knapsack: Solved using a Greedy Algorithm ($O(n \log n)$).

3. Solution Guarantee

- 0/1 Knapsack: The greedy approach does not guarantee an optimal solution.
- Fractional Knapsack: The greedy approach always provides an optimal solution.

4. Time Complexity

- 0/1 Knapsack: $O(nW)$ (where W is the knapsack capacity) using Dynamic Programming.
- Fractional Knapsack: $O(n \log n)$ (due to sorting items by value-to-weight ratio).

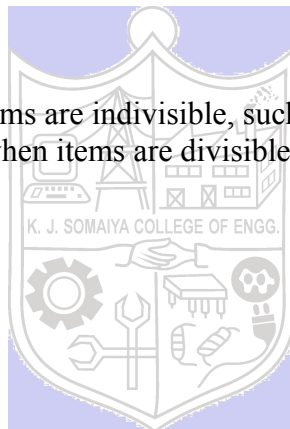
5. Use Case

- 0/1 Knapsack: Used when items are indivisible, such as choosing laptops or books.
- Fractional Knapsack: Used when items are divisible, such as selecting gold or liquid fuel.

Example 1: 0/1 Knapsack Problem

Given Data:

- Knapsack Capacity = 50
- Items:
 - Item 1: Weight = 10, Value = 60
 - Item 2: Weight = 20, Value = 100
 - Item 3: Weight = 30, Value = 120



Optimal Selection (0/1 Knapsack)

- We cannot take fractions of items.
- The best way is to pick Item 2 (20 kg) and Item 3 (30 kg).
- Total Value = $100 + 120 = 220$.

Example 2: Fractional Knapsack Problem

Given Data: (Same as above)

Optimal Selection (Fractional Knapsack)

1. Sort items by value/weight ratio:
 - Item 1: 6
 - Item 2: 5
 - Item 3: 4
2. Pick Item 1 fully (10 kg, value = 60).
3. Pick Item 2 fully (20 kg, value = 100).
4. Take 2/3rd of Item 3 (20 kg out of 30 kg, value = $120 \times (2/3) = 80$).
5. Total Value = $60 + 100 + 80 = 240$.

Conclusion: (Based on the observations):

From this experiment, I learned how the Fractional Knapsack problem can be efficiently solved using the Greedy approach to maximize value within a given weight limit. By sorting items based on their value-to-weight ratio, we can prioritize the most valuable items and even take fractions when necessary. The experiment also highlighted the difference between 0/1 Knapsack, which requires dynamic programming, and Fractional Knapsack, which is optimally solved using greedy methods. Through implementation and time complexity analysis, I understood why Fractional Knapsack runs in $O(n \log n)$ time, making it a faster and more efficient solution for problems where items can be divided.

Outcome:

CO1 : Implement Divide & Conquer algorithms & derive its time and space complexity.

References:

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Cormen ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.