Name	Surti Keyur
Roll No.	25MCE026
Branch	M.Tech (CSE)

Experiment 6

Aim: *Implement fractional knapsack problem using Greedy approach.*

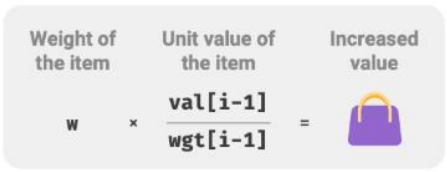
Introduction:

The Fractional Knapsack Problem is a classic optimization problem in computer science that demonstrates the power of the Greedy algorithmic approach. The objective is to maximize the total value that can be carried in a knapsack of fixed capacity, where items can be divided into fractions.

Each item has a specific value and weight, and the goal is to determine the most valuable combination of items (including fractions) that fit within the knapsack's weight limit. The greedy strategy solves this problem efficiently by always selecting items based on the highest value-to-weight ratio (value per unit weight) first.

This approach works optimally for the fractional version of the knapsack problem because local (greedy) choices — picking the item that gives the most value per unit weight — also lead to a globally optimal solution. However, it does not guarantee optimality in the 0/1 Knapsack problem, where items cannot be divided.

Index	Weight	Value	Unit value
i	wgt[i-1]	val[i-1]	$\frac{val[i-1]}{wgt[i-1]}$
1	10	50	5
2	20	120	6
3	30	150	5
4	40	210	5.25
5	50	240	4.8



Code:

```
#include<bits/stdc++.h>
using namespace std;

//creating the structure
struct Item {
    int v, w;
```

```
};

//cmp funtin
bool cmp(const Item a, const Item b) {
    double r1 = (double)a.v / a.w;
    double r2 = (double)b.v / b.w;
    return r1 > r2;
}

int main(){

    //entering the capacity
    double capacity;
    cout<<"Enter the capacity of the knapsack: ";
    cin>>capacity;
    if(capacity <= 0){
        cout<<"Capacity must be positive. Please enter a valid capacity."<<endl;
        return 1;
    }

    //entering the weights and values in the item array
    int n;
    cout<<"Enter the number of items: ";
    cin>>n;
    if(n <= 0){
        cout<<"Number of items must be positive. Please enter a valid number."<<endl;
        return 1;
    }

    //creating the item array
    vector<Item> arr(n);
    cout<<"Enter the weights and values of the items: "<<endl;
    for(int i=0; i<n; i++){
        cin>>arr[i].w>>arr[i].v;
        if(arr[i].w <= 0 || arr[i].v < 0){
            cout<<"Weights must be positive and values cannot be negative. Please enter valid
values."<<endl;
            return 1;
    }
    }

    //sorting the array based on the ratio of v to w
    sort(arr.begin(), arr.end(),cmp);

    // printing the sorted array
    cout<<"Sorted array based on v to w ratio: "<<endl;
    for(int i=0; i<n; i++){
        cout<<arr[i].w<<" "<<arr[i].v<<endl;
    }
```

```
    //calculating the maximum v
    double total = 0.0;
    for(int i=0; i<n; i++){
        if(capacity == 0){
            break;
        }
        if(arr[i].w <= capacity){
            total += arr[i].v;
            capacity -= arr[i].w;
        }
        else{
            total += arr[i].v * ((double)capacity / arr[i].w);
            capacity = 0;
        }
    }

    //printing the max v
    cout<<"Total maximum v in the knapsack: "<<total<<endl;

    return 0;
}
```

Output:

```
Enter the capacity of the knapsack: 50
Enter the number of items: 3
Enter the weights and values of the items:
10 60
20 100
30 120
Sorted array based on v to w ratio:
10 60
20 100
30 120
Total maximum v in the knapsack: 240
PS A:\Study\Nirma M.tech\Sem 1\DSA\Practical\Prac6>
```

Analysis:

The core idea of the Dynamic Programming (DP) approach in the Assembly Line Scheduling problem is to determine the minimum total time required to process a product through multiple assembly lines, considering both station times and transfer times between lines. DP is used because the problem exhibits overlapping subproblems and optimal substructure, meaning the optimal solution of the overall process can be built from optimal solutions of smaller subproblems.

General Logic:

- Calculate Value-to-Weight Ratio:
  For each item, compute

$$r_i = \frac{value_i}{weight_i}$$

This ratio determines which items provide the most value per unit of weight.

- Sort Items by Ratio (Descending):
  Arrange the items so that the item with the highest ratio comes first. This ensures that the most valuable items (per weight) are considered earlier.

- Pick Items Greedily:
  Start with the item having the highest ratio. If the entire item fits in the remaining knapsack capacity, take it completely. Otherwise, take the fractional part of it that fits and stop (since the knapsack is now full).
- Compute Total Value:
  Add the value of all fully and fractionally included items to get the maximum achievable value.

Conclusion:

The Fractional Knapsack problem highlights the efficiency and simplicity of the Greedy algorithm in optimization tasks. By always selecting the item with the highest value-to-weight ratio, the algorithm ensures that each choice contributes maximally to the total value. This local decision-making approach leads to a globally optimal solution for fractional cases. The method efficiently balances value maximization with capacity constraints, requiring only sorting and a single pass through the items. With a time complexity of O(n log n), the greedy approach provides a fast and reliable solution for resource allocation and load optimization problems where partial selections are allowed.