DAA Experiment 5
Comps A (A Batch)

NAME	SHAAN AGARWAL
UID	2021300001
BATCH	SY BTECH COMPS A
EXPERIMENT NO.	5
DATE OF PERFORMANCE	30th March, 2023

Aim: To Solve Knapsack Problem using Greedy Approach

Theory:

1. **Main Idea:** The knapsack problem is an optimization problem used to illustrate both problem and solution. It derives its name from a scenario where one is constrained in the number of items that can be placed inside a fixed-size knapsack. Given a set of items with specific weights and values, the aim is to get as much value into the knapsack as possible given the weight constraint of the knapsack.

The knapsack problem is an example of a combinational optimization problem, a topic in mathematics and computer science about finding the optimal object among a set of objects. This is a problem that has been studied for more than a century and is a commonly used example problem in combinatorial optimization, where there is a need for an optimal object or finite solution where an exhaustive search is not possible. The problem can be found real-world scenarios like resource

allocation in financial constraints or even in selecting investments and portfolios. It also can be found in fields such as applied mathematics, complexity theory, cryptography, combinatorics and computer science. It is easily the most important problem in logistics.

In the knapsack problem, the given items have two attributes at minimum – an item's value, which affects its importance, and an item's weight or volume, which is its limitation aspect. Since an exhaustive search is not possible, one can break the problems into smaller sub-problems and run it recursively. This is called an optimal sub-structure.

This deals with only one item at a time and the current weight still available in the knapsack. The problem solver only needs to decide whether to take the item or not based on the weight that can still be accepted. However, if it is a program, re-computation is not independent and would cause problems. This is where dynamic programming techniques can be applied. Solutions to each sub-problem are stored so that the computation would only need to happen once.

ALGORITHM FOR SOLVING KNAPSACK PROBLEM:

1. Sort the given array of items according to weight / value (W/V) ration in the descending order.
2. Start adding the item with the maximum W/V ratio.
3. Add the whole item, if the current weight is less than the capacity, else, add a portion of the item to the knapsack.
4. Stop, when all the items have been considered and the total weight becomes equal to the weight of the given knapsack.

Pseucode:

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
```

CODE IMPLEMENTATION:

```cpp
#include<bits/stdc++.h>

using namespace std;

// A structure to represent items in the knapsack
struct Item {
    int value;
    int weight;
    double ratio; // value to weight ratio
};

// Function to compare items based on their ratio
bool compareItems(Item i1, Item i2) {
    return (i1.ratio > i2.ratio);
}

// Function to solve the fractional knapsack problem using greedy approach
double knapsack(int W, vector<Item>& items) {
    double max_value = 0.0;
    sort(items.begin(), items.end(), compareItems); // Sort items based on their ratio
```

```
    cout << left << setw(10) << "Item"
         << setw(10) << "Value"
         << setw(10) << "Weight"
         << setw(10) << "Ratio"
         << setw(10) << "Take"
         << setw(10) << "Value Taken"
         << endl;

    for (int i = 0; i < items.size(); i++) {

        int wt = items[i].weight;
        int val = items[i].value;
        double fraction = 0.0;

        if (W >= wt) {
            // Take the whole item
            W -= wt;
            max_value += val;
            cout << left << setw(10) << i+1
                 << setw(10) << val
                 << setw(10) << wt
                 << setw(10) << items[i].ratio
                 << setw(10) << "Yes"
                 << setw(10) << val
                 << endl;
        } else {
            // Take a fraction of the item
            fraction = (double) W / (double) wt;
            max_value += (val * fraction);
            W = 0;
            cout << left << setw(10) << i+1
                 << setw(10) << val
                 << setw(10) << wt
                 << setw(10) << items[i].ratio
                 << setw(10) << fixed << setprecision(2) << fraction
                 << setw(10) << fixed << setprecision(2) << val*fraction
                 << endl;
        }
    }

    return max_value;
}

// Main function
int main() {
    int n; // Number of items
    int W; // Total weight capacity of the knapsack

    cout << "Enter the number of items: ";
    cin >> n;

    cout << "Enter the total weight capacity of the knapsack: ";
    cin >> W;
```

```
    vector<Item> items(n);

    for (int i = 0; i < n; i++) {
        cout << "Enter the value and weight of item " << i+1 << ": ";
        cin >> items[i].value >> items[i].weight;
        items[i].ratio = (double) items[i].value / (double) items[i].weight;
    }

    double max_value = knapsack(W, items);

    cout << "The maximum value that can be obtained is: " << max_value << endl;

    return 0;
}
```

OUTPUT:

```
Enter the number of items: 3
Enter the total weight capacity of the knapsack: 50
Enter the value and weight of item 1: 60 20
Enter the value and weight of item 2: 100 50
Enter the value and weight of item 3: 120 30
Item      Value     Weight    Ratio     Take      Value Taken
1         120       30        4         Yes       120
2         60        20        3         Yes       60
3         100       50        2         0.00      0.00
The maximum value that can be obtained is: 180.00
```

```
Enter the number of items: 1
Enter the total weight capacity of the knapsack: 10
Enter the value and weight of item 1: 500 30
Item      Value     Weight    Ratio     Take      Value Taken
1         500       30        16.6667   0.33      166.67
The maximum value that can be obtained is: 166.67
```

SOLVED EXAMPLES ON PAPER

Shaan Agarwal 2021300001

classmate Date Page

Solved Examples - Fractional Knapsack Problem

Example - 1 : Profit = {60, 100, 120}
Weight = {20, 50, 30}.
Capacity = 50

Arranging items w.r.t. profit / weight ratio.

Item	Profit	weight	Ratio
I_3	120	30	4
I_1	60	20	3
I_2	100	50	2

First two items in the table can be considered to fill at max - capacity.

Max. Profit → 120 + 60 = 180
Total weight → 30 + 20 = 50

∴ Items selected are I_3, I_1.

2021300001

Example 2

Profit = {500}
Weight = {30}
Capacity = 10

Arranging the items w.r.t. profit/weight ratio.

Item	Profit	Weight	Ratio
I_1	500	30	16.67

I_1 can be partially selected.

∴ Max profit = $\frac{500}{3}$ = 166.67

Items selected → $\frac{I_1}{3}$.

CONCLUSION: Thus, after performing the above experiment, I have understood the concept of classical as well as Fractional Knapsack Algorithm. I have implemented the algorithm using a C++ Program and have even shown the pen and paper working. I have used greedy approach for the implementation purpose.