Name: Sapate Vaibhav Ramdas

Roll No: 307B055

Division: 2

Batch: C

Assignment No: 1

Problem Statement:

Write a program to implement the Fractional Knapsack problem using the following approaches:
a. Greedy algorithm.
b. 0/1 knapsack using dynamic programming.
c. Compare the results obtained by the greedy algorithm (a) and the dynamic programming approach (b) and demonstrate that the greedy strategy does not necessarily yield an optimal solution compared to the dynamic programming approach. Provide a scenario or set of test cases that highlight the difference in solutions between these two approaches.

Part A: Greedy algorithm

Program:

```
// Implement fractional Knapsack Problem using greedy approch

#include <iostream>

#include <algorithm>

using namespace std;

// Structure for an item which holds weight & profit of the Item

struct Item

{

    int profit, weight;

    // Constructor- creation of object having profit and weight

    Item(int profit, int weight) : profit(profit), weight(weight)

    {

    }
```

```
};
// Comparison function to sort Item according to profit/weight ratio
int compareByProfit(struct Item a, struct Item b)
{
    float r1 = (float)a.profit / a.weight;
    float r2 = (float)b.profit / b.weight;
    return r1 > r2;
}
int compareByWeight(struct Item a, struct Item b)
{
    /*int result = compareOperation(a.profit, b.profit);
    return result;*/
    float r1 = (float)a.profit;
    float r2 = (float)b.profit;
    return r1 > r2;
}
int compareByRatio(struct Item a, struct Item b)
{
    float r1 = (float)a.weight;
    float r2 = (float)b.weight;
    return r1 < r2;
}
// Main greedy function of case 1- maximum profit to solve problem
double fractionalKnapsack_1(struct Item arr[],
                            int knapsack_capacity, int size)
{
    sort(arr, arr + size, compareByWeight); // Sort Item on basis of maximum
profit
```

```
    int curWeight = 0;                  // Current weight in knapsack
    float Total_profit = 0.0;           // Total Profit Variable
    // Going through all Items
    for (int i = 0; i < size; i++)
    {
        // Add item if weight of given item is < knapsack capacity , add it
completely
        if (curWeight + arr[i].weight <= knapsack_capacity)
        {
            curWeight += arr[i].weight;
            Total_profit += arr[i].profit;
        }
        // If we can't add current Item add fractional part of it
        else
        {
            float remain = knapsack_capacity - curWeight;
            Total_profit += arr[i].profit * (remain / arr[i].weight);
            break;
        }
    }
    return Total_profit; // final profit
}
// Main greedy function of case 2-minimum weight to solve problem
// Returns the maximum profit that can be obtained by filling the knapsack
// with a fractional subset of the items, given the weights and profits of
// the items and the capacity of the knapsack.
double fractionalKnapsack_2(struct Item arr[],
                            int knapsack_capacity, int size)
```

```
{
   sort(arr, arr + size, compareByRatio); // Sort Item on basis of ratio
   int curWeight = 0;                    // Current weight in knapsack
   float Total_profit = 0.0;             // Total Profit Variable
   // Going through all Items
   for (int i = 0; i < size; i++)
   {
      // Add item if weight of given item is < knapsack capacity , add it
completely
      if (curWeight + arr[i].weight <= knapsack_capacity)
      {
         curWeight += arr[i].weight;
         Total_profit += arr[i].profit;
      }
      // If we can't add current Item add fractional part of it
      else
      {
         float remain = knapsack_capacity - curWeight;
         Total_profit += arr[i].profit * (remain / arr[i].weight);
         break;
      }
   }
   return Total_profit; // final profit
}
// Main greedy function of case 3-P/W ration to solve problem
double fractionalKnapsack(struct Item arr[],
                          int knapsack_capacity, int size)
{
```

```
    sort(arr, arr + size, compareByProfit); // Sort Item on basis of ratio
    int curWeight = 0;                     // Current weight in knapsack
    float Total_profit = 0.0;              // Total Profit Variable
    // Going through all Items
    for (int i = 0; i < size; i++)
    {
        // Add item if weight of given item is < knapsack capacity , add it
completely
        if (curWeight + arr[i].weight <= knapsack_capacity)
        {
            curWeight += arr[i].weight;
            Total_profit += arr[i].profit;
        }
        // If we can't add current Item add fractional part of it
        else
        {
            float remain = knapsack_capacity - curWeight;
            Total_profit += arr[i].profit * (remain / arr[i].weight);
            break;
        }
    }
    return Total_profit; // final profit
}
// main function
int main()
{
    // Weight of knapsack
    int knapsack_capacity = 25;
```

```
    // Given weights and profits as a pairs
    Item arr[] = {{24, 24},
                  {18, 10},
                  {18, 10},
                  {10, 7}};
    int size = sizeof(arr) / sizeof(arr[0]); // to find the size of array
    // Function Call
    cout << "Maximum profit earned in case 1 (Maximum profit)="
         << fractionalKnapsack_1(arr, knapsack_capacity, size) << endl;
    ; // calling a fractional knapsack function for case 1
    cout << "Maximum profit earned in case 2 (Minimum Weight)= "
         << fractionalKnapsack_2(arr, knapsack_capacity, size) << endl; // calling
a fractional knapsack function for case 2
    cout << "Maximum profit earned in case 3 (Profit/weight ratio)= "
         << fractionalKnapsack(arr, knapsack_capacity, size) << endl; // calling a
fractional knapsack function for case 3
    return 0;
}
```

Output:

Maximum profit earned in case 1 (Maximum profit)=25.8

Maximum profit earned in case 2 (Minimum Weight)= 42.4

Maximum profit earned in case 3 (Profit/weight ratio)= 43.1429

Part B: 0/1 knapsack using dynamic programming

Program:

```
// Implementation of 0/1 knapsack using dynamic programming
#include <iostream>
#include <iomanip>
using namespace std;
// max function to find out maximum values from the given 2 values
int max(int x, int y)
{
   return (x > y) ? x : y; // max function to find max function among 2 differnt
values
}
// 0/1 Knapsack function definition
int knapSack(int k_capacity, int w[], int p[], int n)
{
   int i, j;
   int A[n + 1][k_capacity + 1]; // 2D array which will store the values(Matrix)
   // scan for every object
   for (i = 0; i <= n; i++)
   {
      for (j = 0; j <= k_capacity; j++)
      { //intialize the matrix
         if (i == 0 || j == 0)
            A[i][j] = 0;                              //initialize the first row and
first column of the matrix
         else if (w[i - 1] <= j)                        //till the weight is grater
than w[i-1] execute the else if
```

```
            A[i][j] = max(A[i - 1][j], p[i - 1] + A[i - 1][j - w[i - 1]]); //calculating
max value  for every entry
          else
            A[i][j] = A[i - 1][j]; //copy the values from above row
      }
   }
   cout << "Matrix generated for Dynamic programming:"
        << "\n"
        << "\n";
   for (i = 0; i <= n; i++)
   {
      for (j = 0; j <= k_capacity; j++)
      {
         cout << setw(2) << A[i][j] << " ";
      }
      cout << "\n";
   }
   cout << "\n";
   int profit = A[n][k_capacity];
   // cout<<"profit is:"<<profit<<"\n";
   // return A[n][k_capacity]; //last box of matrix holds a maximum profit
   int wt = k_capacity;
   for (i = n; i > 0 && profit > 0; i--)
   {
      // either the result comes from the top (A[i-1][w]) or from (p[i-1] + A[i-1]
[w-wt[i-1]]) as in Knapsack table. If it comes from the latter one/ it means the
item is included.cout<<A[i-1][j];
      if (profit == A[i - 1][j])
```

```
            cout << "This item is not included" << i << "->0"
                 << "\n";
        else
        {
            // This item is included.
            cout << "This item is included" << i << "->1"
                 << "\n";
            // cout<<" "<<w[i - 1]<<" " ;
            // Since this weight is included its
            // value is deducted
            profit = profit - p[i - 1];
            wt = wt - w[i - 1];
        }
    }
    cout << "Maximum Profit for a 0/1 knapsack is: ";
    return A[n][k_capacity];
}
int main(){
    cout << "Enter the number of objects for a Knapsack: "; // accept the number
of objects from user
    int n, K_capacity;
    cin >> n;
    int p[n], w[n];
    for (int i = 0; i < n; i++)
    {
        cout << "Enter Profit and weight for item using space " << i << ": "; //
accept the profit and weight values
        cin >> p[i];
```

```
        cin >> w[i];
    }
    cout << "Enter the capacity of knapsack: "; // enter the knapsack capacity
    cin >> K_capacity;
    cout << knapSack(K_capacity, w, p, n); // fuction call for knapsack
    return 0;
}
```

Output:

```
Enter the number of objects for a Knapsack: 5
Enter Profit and weight for item using space 0: 10 2
Enter Profit and weight for item using space 1: 5 3
Enter Profit and weight for item using space 2: 15 5
Enter Profit and weight for item using space 3: 7 7
Enter Profit and weight for item using space 4: 6 1
Enter the capacity of knapsack: 7
Matrix generated for Dynamic programming:
 0  0   0   0   0   0   0   0
 0  0 10 10 10 10 10 10
 0  0 10 10 10 15 15 15
 0  0 10 10 10 15 15 25
 0  0 10 10 10 15 15 25
 0  6 10 16 16 16 21 25

This item is included5->1
This item is included4->1
This item is included3->1
Maximum Profit for a 0/1 knapsack is: 25⏎
```