7. Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.

Objectives: This Program enable students to :

Learn the **0/1 Knapsack** problem using Greedy method to implement using c/c++

ALGORITHM:

Knapsack (i, j)

```
//Input: A nonnegative integer i indicating the number of the first items being considered and
       a non negative  integer  j indicating the Knapsack's capacity.

//Output:The value ofan optimal feasible subset of the first i items.

 //Note:Uses as global variables input arrays Weights [1...n], Values [ 1… n] and
        table V [0…n, 0…W ] whose entries are initialized with -1's except for
        row 0 and column 0  initialized with 0's.

ifV [i, j] < 0

        if j< Weights[i]
then value = Knapsack
( i-1, j )

else
    value = max ( Knapsack (  i-1,  j),
        values[I] + Knapsack ( i-1, j-
           Weights[i] ) ) V[i j] = value

returnV [  i,  j ]
```

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Structure to represent an item
struct Item {
    int weight; int
    value;
};
// Function to solve discrete knapsack using greedy approach
int discreteKnapsack(vector<Item>& items, int capacity) {
    // Sort items based on their value per unit weight
     sort(items.begin(), items.end(), [](const Item& a, const Item& b) {
        return (double)a.value / a.weight > (double)b.value / b.weight;
    });
    int totalValue = 0;
    int currentWeight = 0;
    // Fill the knapsack with items for
    (const Item& item : items) {
        if (currentWeight + item.weight <= capacity) {
            currentWeight += item.weight;
            totalValue += item.value;
        }
    }
    return totalValue;
}
// Function to solve continuous knapsack using greedy approach
double continuousKnapsack(vector<Item>& items, int capacity) {
    // Sort items based on their value per unit weight sort(items.begin(),
    items.end(), [](const Item& a, const Item& b) {
        return (double)a.value / a.weight > (double)b.value / b.weight;
    });
    double totalValue = 0.0; int
    currentWeight = 0;
    // Fill the knapsack with items fractionally for
    (const Item& item : items) {
        if (currentWeight + item.weight <= capacity) {
            currentWeight += item.weight;
            totalValue += item.value;
        } else {
            int remainingCapacity = capacity - currentWeight;
            totalValue += (double)item.value / item.weight * remainingCapacity;
            break;

        } }
    return totalValue;
}
int main() { vector<Item>
    items; int n, capacity;
    // Input number of items and capacity of knapsack
    cout << "Enter the number of items: ";
    cin >> n;
    cout << "Enter the capacity of knapsack: "; cin
    >> capacity;
    // Input the weight and value of each item
    cout << "Enter the weight and value of each item:" << endl; for
    (int i = 0; i < n; i++) {
        Item item;
        cout << "Item " << i + 1 << ": "; cin
        >> item.weight >> item.value;
        items.push_back(item);
    }
    // Solve discrete knapsack problem
    int discreteResult = discreteKnapsack(items, capacity);
    cout << "Maximum value for discrete knapsack: " << discreteResult << endl;
    // Solve continuous knapsack problem
    double continuousResult = continuousKnapsack(items, capacity);
    cout << "Maximum value for continuous knapsack: " << continuousResult << endl; return
    0;
}
```