Reg.No: 24BCE0554
Name : Partha Pratim Gogoi
Topic : Fractional Knapsack

Algorithm:

```
1  Capacity = 15                                          [T.C: O(1)]
2  profit = [ … ]                                         [T.C: O(1)]
3  weight = [ … ]                                         [T.C: O(1)]
4  struct Item                                            [T.C: O(1)]
   4.1  int id
   4.2  int profit
   4.3  int weight
   4.4  double ratio
   4.5  Constructor(id, profit, weight)                   [T.C: O(1)]
      4.5.1  this→id = id
      4.5.2  this→profit = profit
      4.5.3  this→weight = weight
      4.5.4  this→ratio = profit/weight
5  vector<Item> items                                     [T.C: O(1)]
6  populateItemsIntoArray(profits, weight)                [T.C: O(n)]
   6.1  for i in profit                                   [T.C: O(n)]
   6.2  push_back(Item(i, profit[i], weight[i]))          [T.C: O(1)]
7  compare(a, b)                                          [T.C: O(1)]
   7.1  return a.profit > b.profit
8  fractionalKnapsack(capacity, items)           [T.C: O(n+nlog(n))]
   8.1  sort(items[0], items[-1], compare)          [T.C: O(nlog(n))]
   8.2  totalProfit = 0.0
   8.3  currentWeight = 0
   8.4  for item in items                                 [T.C: O(n)]
      8.4.1  if currentWeight + item.weight<=capacity     [T.C: O(1)]
         8.4.1.1  currentWeight += item.weight
         8.4.1.2  totalProfit += item.profit
      8.4.2  else                                         [T.C: O(1)]
         8.4.2.1  remaining = capacity-currentWeight
         8.4.2.2  if remaining>0                          [T.C: O(1)]
            8.4.2.2.1  fraction = remaining/item.weight
            8.4.2.2.2  fractionalProfit = item.profit*fraction
         8.4.2.3  break
   8.5  return totalProfit                                [T.C: O(1)]
```

Time Complexity:

1. sort(items[0], items[-1], compare)
This function has been implemented in "**algorithm**" library of C++ to have **O(nlogn)** time complexity

Total Time Complexity = O(1) + O(n) + O(n+log(n))
= O(n + nlog(n))
= O(nlog(n))

Source Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

///////////////////
/// Define item ///
///////////////////
struct Item {
    int id;
    int profit;
    int weight;
    double ratio;

    // ----------- //
    // Constructor //
    // ----------- //
    Item(int id, int profit, int weight) {
        this->id = id;
        this->profit = profit;
        this->weight = weight;
        ratio = (double) profit/weight;
    }
};

//////////////////
/// Sort Logic ///
//////////////////
bool compare(Item a, Item b) {
    // --------------------------------- //
    // Greedy Choice:- Highest P:W Ratio //
    // --------------------------------- //
    return a.ratio > b.ratio;
}

//////////////////
/// Main Logic ///
//////////////////
double fractionalKnapsack(int M, vector<Item>& items) {
    // -------------------------------- //
    // Sort highest to lowest P:W ratio //
    // -------------------------------- //
    cout << "Sorting 'items' [highest to lowest P:W]..." << endl;
    sort(items.begin(), items.end(), compare);
```

```
    double totalProfit = 0.0;
    int currentWeight = 0;

    // ------------------ //
    // Make Greedy Choices //
    // ------------------ //
    cout << "Making greedy choices..." << endl;
    for (auto& item: items) {
        if (currentWeight + item.weight <= M) {

          // Incrementing values
          currentWeight += item.weight;
          totalProfit += item.profit;
        } else {
          cout << "Taking current highest P:W item exceeds capacity. Taking fractional
part..." << endl;
          // This case is when we have to take a fractional
          // part an item to reach max profit
          int remaining = M - currentWeight;
          if (remaining > 0) {
            double fraction = (double)remaining / item.weight;
            double fractionalProfit = item.profit * fraction;

            totalProfit += fractionalProfit;
          }
          // Break since capacity is filled
          break;
      }
    }
    // ---------------------------- //
    // Return maximum possible profit //
    // ---------------------------- //
    return totalProfit;
}

///////////////////
/// Driver Code ///
///////////////////
int main() {
    cout << "Initializing starting conditions..." << endl;
    int n=10;      // No. of objects
    int M=15;      // Knapsack capacity
    int profit[] = {10, 5, 15, 7, 6, 18, 3, 12, 20, 8};
    int weight[] = { 2, 3,  5, 7, 1,  4, 1,  6,  5, 2};

    // ---------------------- //
    // Populating 'items' array //
    // ---------------------- //
    cout << "Populating 'items' array..." << endl;
    vector<Item> items;
    for (int i=0; i<n; i++) {
        items.push_back(Item(i, profit[i], weight[i]));
    }
    // ---------------------------- //
    // Apply fractional knapsack logic //
    // ---------------------------- //
    cout << "Applying Fractional Knapsack..." << endl;
    double maxProfit = fractionalKnapsack(M, items);
```

```
    cout << endl << "Maximum Profit: " << fixed << setprecision(2) << maxProfit <<
endl;

    return 0;
}
```

Sample Output:

```
rug-arch@Oxide [Fractional Knapsack]>> g++ fractionalKnapsack.cpp
rug-arch@Oxide [Fractional Knapsack]>> ./a.out
Initializing starting conditions...
Populating 'items' array...
Applying Fractional Knapsack...
Sorting 'items' [highest to lowest P:W]...
Making greedy choices...
Taking current highest P:W item exceeds capacity. Taking fractional part...

Maximum Profit: 65.00
```