National Institute Of Technology Surathkal Mangalore Karnataka-575025

Department Of Information Technology

Lab Assignment :- 03

Name:- Chikkeri Chinmaya

Roll Number:- 211IT017

Branch:- Information Technology (B.Tech)

Section :- S13

Course :- Data Structure And Algorithm (IT251)

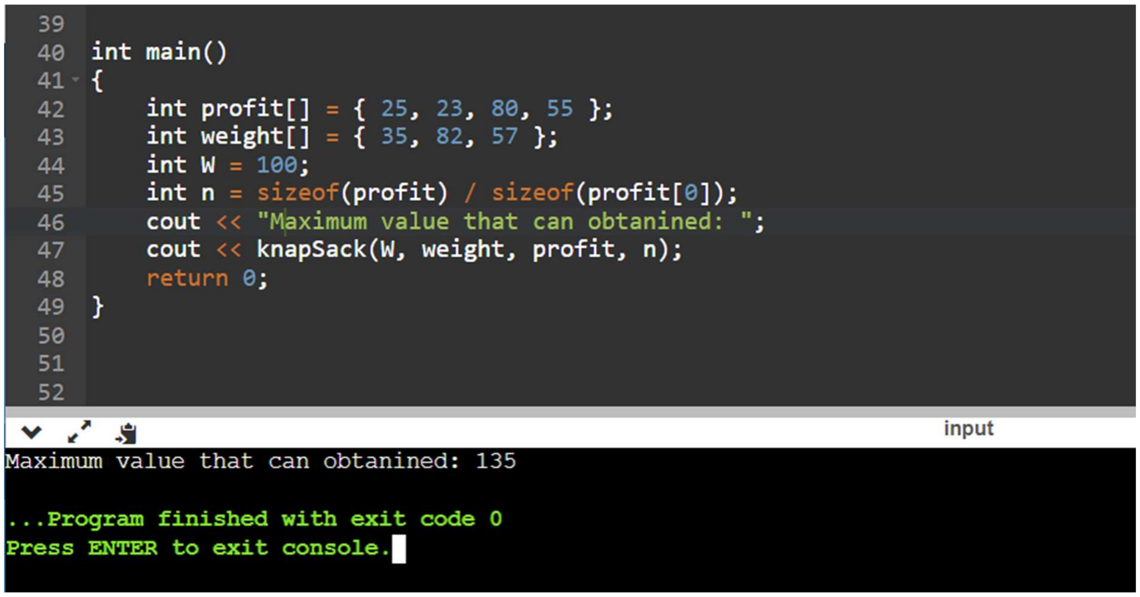
Submitted To:-
HariDas Pai Sir

1st Question :-

a

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


// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that
// can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{

    // Base Case
    if (n == 0 || W == 0)
        return 0;


    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);

    // Return the maximum of two cases:

    else
        return max(
            val[n - 1]
                + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}


int main()
{
    int profit[] = { 25, 23, 80, 55 };
    int weight[] = { 35, 82, 57 };
    int W = 100;
    int n = sizeof(profit) / sizeof(profit[0]);
    cout << "Maximum value that can obtanined: ";
    cout << knapSack(W, weight, profit, n);
    return 0;
}
```

Out Put:-



```
int main()
{
    int profit[] = { 25, 23, 80, 55 };
    int weight[] = { 35, 82, 57 };
    int W = 100;
    int n = sizeof(profit) / sizeof(profit[0]);
    cout << "Maximum value that can obtanined: ";
    cout << knapSack(W, weight, profit, n);
    return 0;
}
```

```
Maximum value that can obtanined: 135

...Program finished with exit code 0
Press ENTER to exit console.
```

1. **Recursive method:-** In this method, we define a recursive function that tries all possible combinations of items and capacities, and returns the maximum value that can be obtained. This method involves a lot of redundant computation and has an exponential time complexity. However, it can be useful for small input sizes.

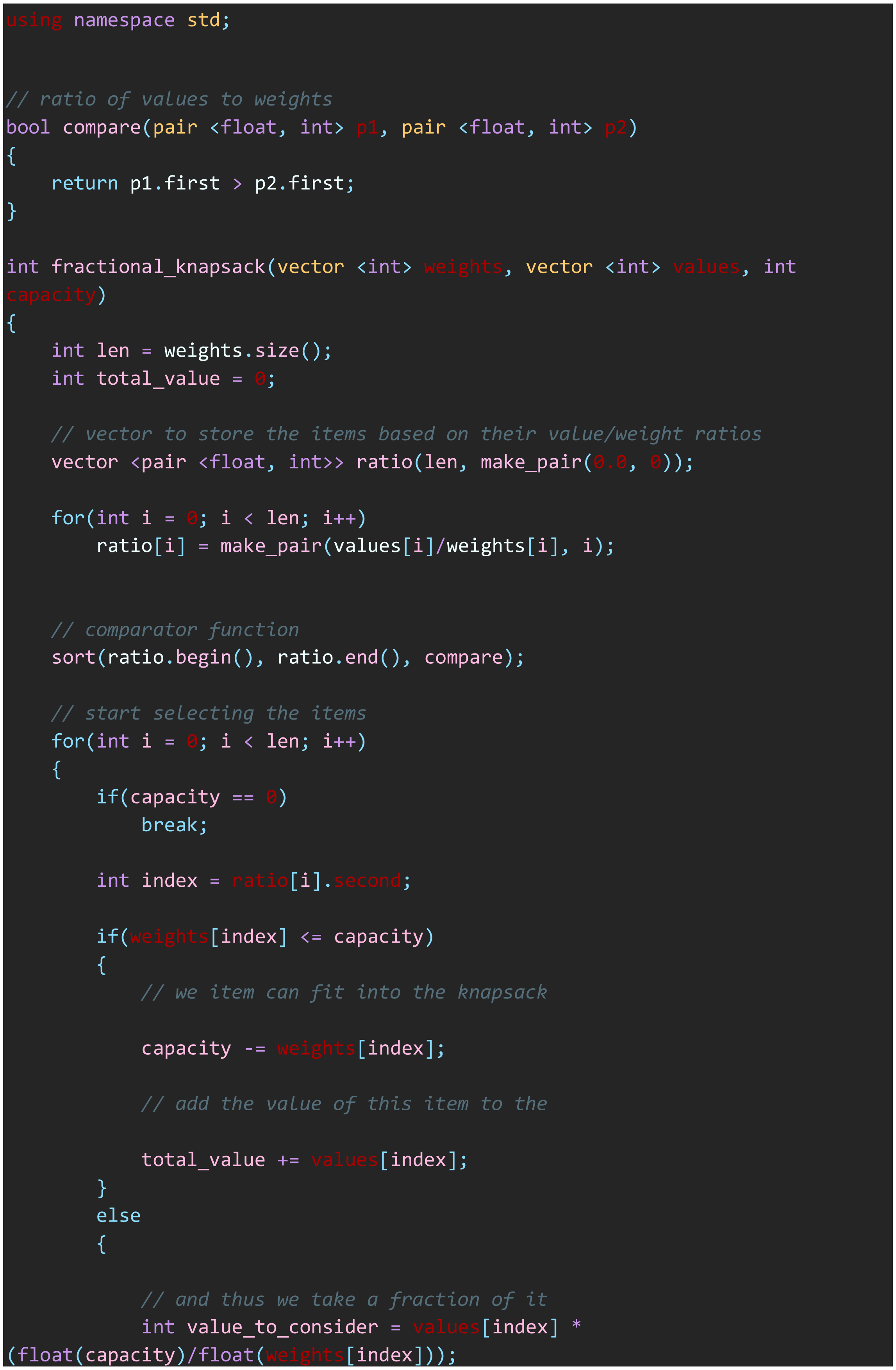
2. **Memoization method:-**In this method, we define a memo table to store the maximum value that can be obtained using the first i items and a knapsack of capacity w. Before making a recursive call, we check whether the required result is already present in the memo table. If it is, we simply return it; otherwise, we compute it recursively and store it in the memo table for future use

3. **Greedy method:-** in a greedy manner based on some criterion, such as the ratio of value to weight. In this method, we sort the items in decreasing order of their ratio and then add them to the knapsack in that order, as long as the knapsack capacity allows.

4. **Branch and bound method**:- In this method, we start by solving a relaxed version of the problem, where we can take fractional amounts of the items. Then, we divide the problem into two subproblems, one where we take the next item and another where we do not take it. We add each subproblem to a priority queue and keep expanding the one with the highest potential value until we find the optimal solution or prove that it is not possible

2nd Question:-

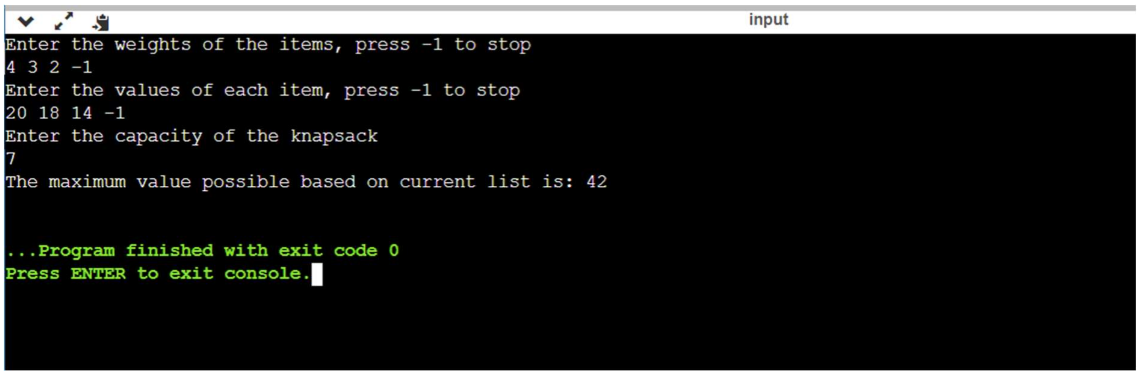
```
#include <iostream>
#include <algorithm>
#include <vector>
```



```cpp
using namespace std;


// ratio of values to weights
bool compare(pair <float, int> p1, pair <float, int> p2)
{
    return p1.first > p2.first;
}

int fractional_knapsack(vector <int> weights, vector <int> values, int
capacity)
{
    int len = weights.size();
    int total_value = 0;

    // vector to store the items based on their value/weight ratios
    vector <pair <float, int>> ratio(len, make_pair(0.0, 0));

    for(int i = 0; i < len; i++)
        ratio[i] = make_pair(values[i]/weights[i], i);


    // comparator function
    sort(ratio.begin(), ratio.end(), compare);

    // start selecting the items
    for(int i = 0; i < len; i++)
    {
        if(capacity == 0)
            break;

        int index = ratio[i].second;

        if(weights[index] <= capacity)
        {
            // we item can fit into the knapsack

            capacity -= weights[index];

            // add the value of this item to the

            total_value += values[index];
        }
        else
        {

            // and thus we take a fraction of it
            int value_to_consider = values[index] *
(float(capacity)/float(weights[index]));
```

```cpp
            total_value += value_to_consider;
            capacity = 0;
        }
    }

    return total_value;
}

int main()
{
    cout << "Enter the weights of the items, press -1 to stop" << endl;

    vector <int> weights;

    while(true)
    {
        int weight;
        cin >> weight;

        if(weight == -1)
            break;

        weights.push_back(weight);
    }

    cout << "Enter the values of each item, press -1 to stop" << endl;

    vector <int> values;

    while(true)
    {
        int value;
        cin >> value;

        if(value == -1)
            break;

        values.push_back(value);
    }

    cout << "Enter the capacity of the knapsack" << endl;

    int capacity;
    cin >> capacity;

    cout << "The maximum value possible based on current list is: " <<
fractional_knapsack(weights, values, capacity) << endl;
}
```

Out Put:-



```
Enter the weights of the items, press -1 to stop
4 3 2 -1
Enter the values of each item, press -1 to stop
20 18 14 -1
Enter the capacity of the knapsack
7
The maximum value possible based on current list is: 42


...Program finished with exit code 0
Press ENTER to exit console.
```