**“Experiment 2.3”**

Student Name: **SUMIT KUMAR** UID: **20BCS8226**

Branch: **CSE** Section/Group: **808-A**

Semester: **5** Date of Submission: **12-11-22**

Subject Name: **Design and Analysis of Algorithms Lab** Subject Code: **20CSP-312**

1. **Aim/Overview of the practical:**

Code to implement 0-1 Knapsack using Dynamic Programming.

1. **Task to be done/ Which logistics used:**

Dynamic-0-1-knapsack Problem.

1. **Algorithm/Steps:**

## Calculate the profit-weight ratio for each item or product.

* 1. Arrange the items on the basis of ratio in descending order.
  2. Take the product having the highest ratio and put it in the sack.
  3. Reduce the sack capacity by the weight of that product.
  4. Add the profit value of that product to the total profit.
  5. Repeat the above three steps till the capacity of sack becomes 0 i.e. until the sack is full.

for w = 0 to W do

c[0, w] = 0 for i =1 to n do c[i, 0] = 0 for w = 1 to W do if wi ≤ w then if vi + c[i-1, w- wi] then c[i, w] = vi + c[i-1, w-wi]

else c[i, w] = c[i-1, w]

else c[i, w] = c[i-1, w]

# Steps for experiment/practical/Code:

#include<iostream> #define MAX 10 using namespace std; struct product

{ int product\_num; int profit; int weight; float ratio; float take\_quantity;

};

Int main() {

product P[MAX],temp;

int i,j,total\_product,capacity; float value=0;

cout<<"ENTER NUMBER OF ITEMS : ";

cin>>total\_product;

cout<<"ENTER CAPACITY OF SACK : ";

cin>>capacity; cout<<"\n"; for(i=0;i<total\_product;++i)

{

P[i].product\_num=i+1;

cout<<"ENTER PROFIT AND WEIGHT OF PRODUCT "<<i+1<<" : ";

cin>>P[i].profit>>P[i].weight;

P[i].ratio=(float)P[i].profit/P[i].weight; P[i].take\_quantity=0;

}

//HIGHEST RATIO BASED SORTING

for(i=0;i<total\_product;++i)

{

for(j=i+1;j<total\_product;++j)

{

if(P[i].ratio<P[j].ratio)

{

temp=P[i];

P[i]=P[j];

P[j]=temp;

}

}

}

for(i=0;i<total\_product;++i)

{

if(capacity==0) break;

else if(P[i].weight<capacity)

{

P[i].take\_quantity=1; capacity-=P[i].weight;

}

else if(P[i].weight>capacity)

{

P[i].take\_quantity=(float)capacity/P[i].weight; capacity=0;

}

}

cout<<"\n\nPRODUCTS TO BE TAKEN -";

for(i=0;i<total\_product;++i)

{

cout<<"\nTAKE PRODUCT "<<P[i].product\_num<<" : "<<P[i].take\_quantity\*P[i].weight<<" UNITS";

value+=P[i].profit\*P[i].take\_quantity;

}

cout<<"\nTHE KNAPSACK VALUE IS : "<<value;

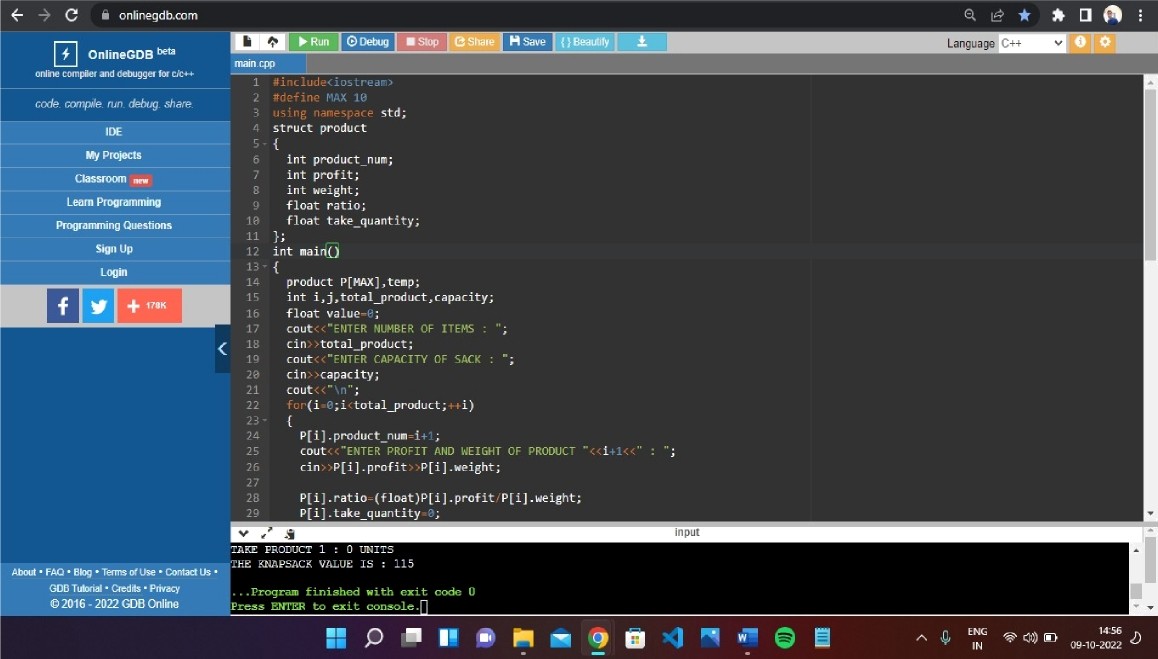
return 0;

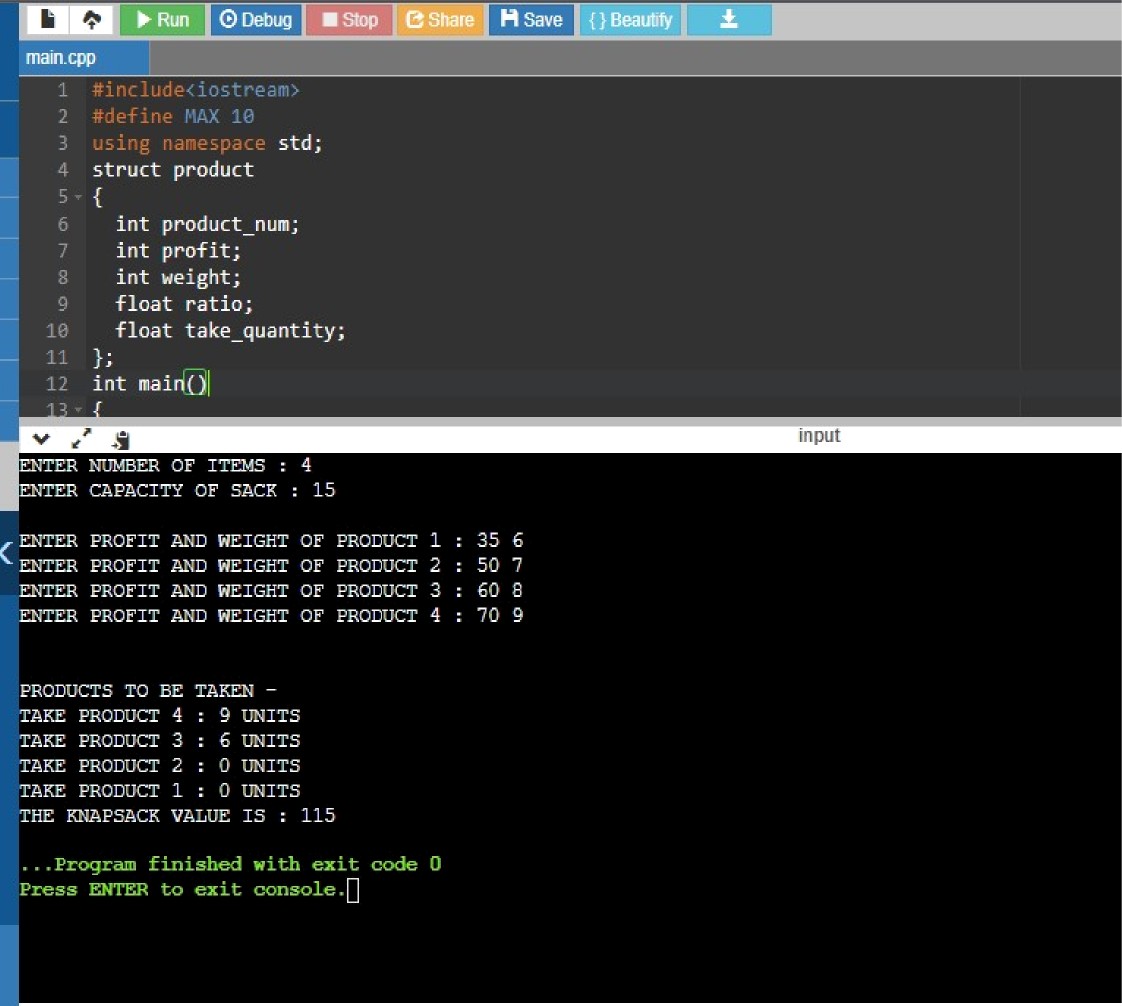
}

# Observations/Discussions/ Complexity Analysis:

This algorithm takes θ(n, w) times as table c has (n + 1).(w + 1) entries, where each entry requires θ(1) time to compute .

1. **Result/Output/Writing Summary:**







**Learning Outcomes:-**

* 1. Create a program keeping in mind the time complexity
  2. Create a program keeping in mind the space complexity
  3. Steps to make optimal algorithm
  4. Learnt about how to implement 0-1 Knapsack problem using dynamic programming.

**Evaluation Grid (To be created per the faculty's SOP and Assessment guidelines):**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. No. | Parameters | Marks Obtained | Maximum Marks |
| 1. | Worksheet completion including writing learning objectives/Outcomes.  (To be submitted at the end of the day). |  |  |
| 2. | Post-Lab Quiz Result. |  |  |
| 3. | Student Engagement in  Simulation/Demonstration/Performance and Controls/Pre-Lab Questions. |  |  |
|  | Signature of Faculty (with Date): | Total Marks Obtained: |  |