



# **DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

Discover. Learn. Empower.



# **CHANDIGARH UNIVERSITY**

Discover. Learn. Empower.

## **UNIVERSITY INSTITUTE OF ENGINEERING**

### **Department of Computer Science & Engineering**

**Subject Name:** DAA Lab

**Subject Code:** 20ITP-312

**Submitted to:**

Er. Hemant kumar saini

**Submitted by:**

**NAME-**Aakash singh

**UID:** 20BCS5620

**Section:** 20BCS-WM-615

**Group:** A

## Worksheet Experiment – 2.3

**Name:** Aakash singh

**UID:** 20BCS5620

**Branch:** BE-CSE

**Section/Group:** 20BCS\_WM-615-A

**Semester:** 5<sup>th</sup>

**Subject:** DAA Lab

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

Code to implement 0-1 Knapsack using Dynamic Programming.

### 2. Task to be done/ Which logistics used:

Dynamic-0-1-knapsack Problem.

### 3. Algorithm/Steps:

1. Calculate the profit-weight ratio for each item or product.
2. Arrange the items on the basis of ratio in descending order.
3. Take the product having the highest ratio and put it in the sack.
4. Reduce the sack capacity by the weight of that product.
5. Add the profit value of that product to the total profit.
6. 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  $w_i \leq w$

then if  $v_i + c[i-1, w-$

$w_i]$  then  $c[i, w] = v_i +$

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

#### 4. 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)
```

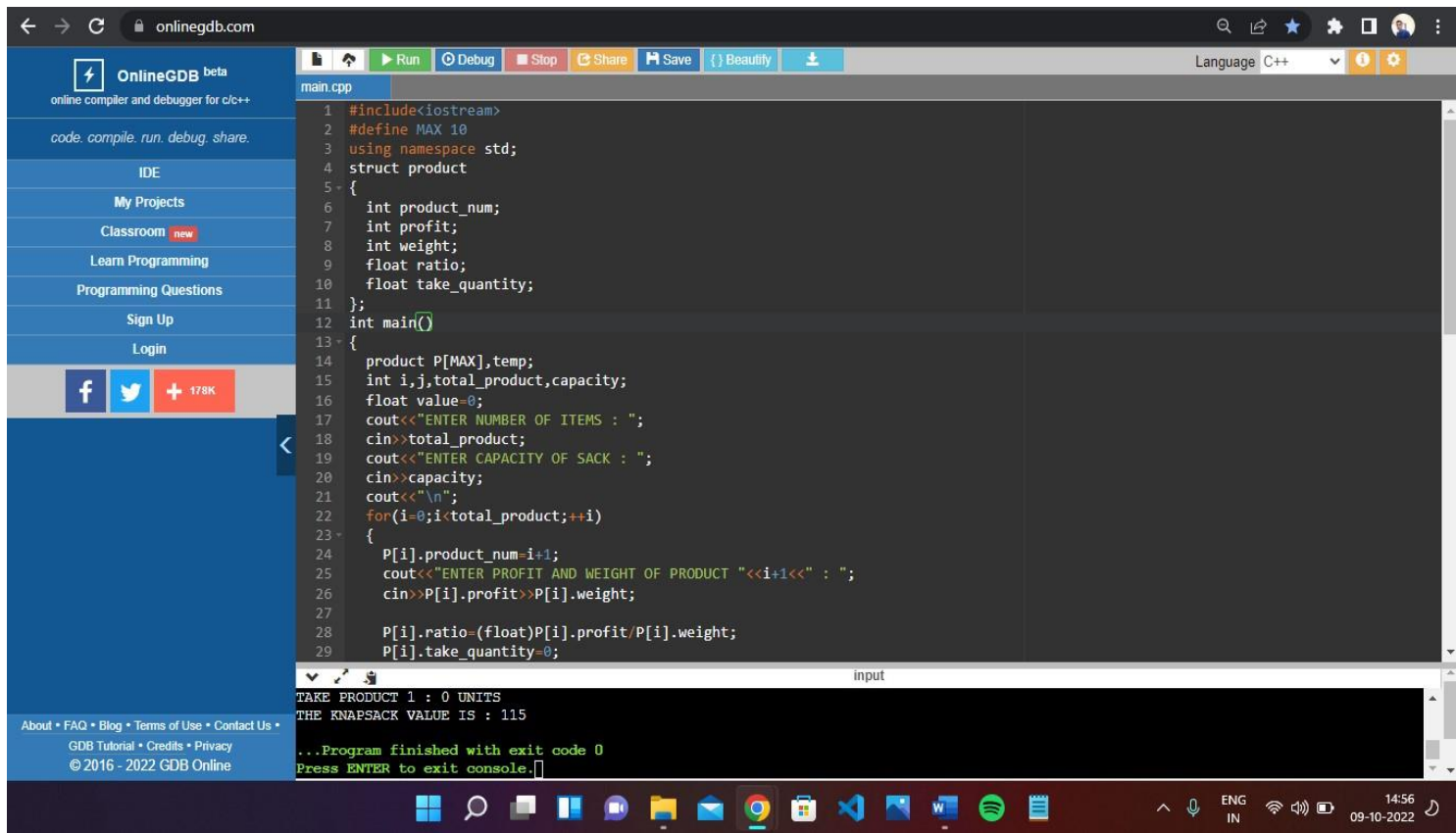
```
{
    if(P[i].ratio<P[j].ratio)
    {
temp=P[i];
    P[i]=P[j];
    P[j]=temp;
    }
}
for(i=0;i<total_product;++i)
{
    if(capacity==0)
break;
    else if(P[i].weight<capacity)
    {
        P[i].take_quantity=1;
        capacity-=P[i].weight;
    }
    else if(P[i].weight>capacity)
    {
        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;
}
```

## 5. Observations/Discussions/ Complexity Analysis:

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

## 6. Result/Output/Writing Summary:



The screenshot shows the OnlineGDB IDE interface. The main editor displays a C++ program for a knapsack problem. The code includes headers, defines a constant MAX, and uses the std namespace. It defines a struct 'product' with fields for product number, profit, weight, ratio, and take quantity. The main function prompts the user to enter the number of items and the capacity of the sack. It then iterates through each item, prompting for its profit and weight, and calculates its ratio and take quantity. The output shows the program's execution results.

```

1 #include<iostream>
2 #define MAX 10
3 using namespace std;
4 struct product
5 {
6     int product_num;
7     int profit;
8     int weight;
9     float ratio;
10    float take_quantity;
11 };
12 int main()
13 {
14     product P[MAX],temp;
15     int i,j,total_product,capacity;
16     float value=0;
17     cout<<"ENTER NUMBER OF ITEMS : ";
18     cin>>total_product;
19     cout<<"ENTER CAPACITY OF SACK : ";
20     cin>>capacity;
21     cout<<"\n";
22     for(i=0;i<total_product;++i)
23     {
24         P[i].product_num=i+1;
25         cout<<"ENTER PROFIT AND WEIGHT OF PRODUCT "<<i+1<<" : ";
26         cin>>P[i].profit>>P[i].weight;
27
28         P[i].ratio=(float)P[i].profit/P[i].weight;
29         P[i].take_quantity=0;

```

input

TAKE PRODUCT 1 : 0 UNITS  
THE KNAPSACK VALUE IS : 115  
...Program finished with exit code 0  
Press ENTER to exit console.

```
main.cpp
1  #include<iostream>
2  #define MAX 10
3  using namespace std;
4  struct product
5  {
6      int product_num;
7      int profit;
8      int weight;
9      float ratio;
10     float take_quantity;
11 };
12 int main()
13 {
    ENTER NUMBER OF ITEMS : 4
    ENTER CAPACITY OF SACK : 15

    ENTER PROFIT AND WEIGHT OF PRODUCT 1 : 35 6
    ENTER PROFIT AND WEIGHT OF PRODUCT 2 : 50 7
    ENTER PROFIT AND WEIGHT OF PRODUCT 3 : 60 8
    ENTER PROFIT AND WEIGHT OF PRODUCT 4 : 70 9

    PRODUCTS TO BE TAKEN -
    TAKE PRODUCT 4 : 9 UNITS
    TAKE PRODUCT 3 : 6 UNITS
    TAKE PRODUCT 2 : 0 UNITS
    TAKE PRODUCT 1 : 0 UNITS
    THE KNAPSACK VALUE IS : 115

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

### **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.