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

NAME-Ruchika Raj

UID: 20BCS9285

Section: 20BCS-WM-615

Group: B

Worksheet Experiment – 2.3

Name: Ruchika Raj **UID: 20BCS9285**
Branch: BE-CSE **Section/Group: 20BCS_WM-615-B**
Semester: 5th **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 wi ≤ w
then  if vi + c[i-1, wwi]
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]

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()
{
    cout<<endl<<"This worksheet belongs to Ruchika Raj (20BCS9285)\n";
    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;
    }

    for (i = 0; i < total_product; ++i)
    {
        for (j = i + 1; j < total_product; ++j)
        {
```

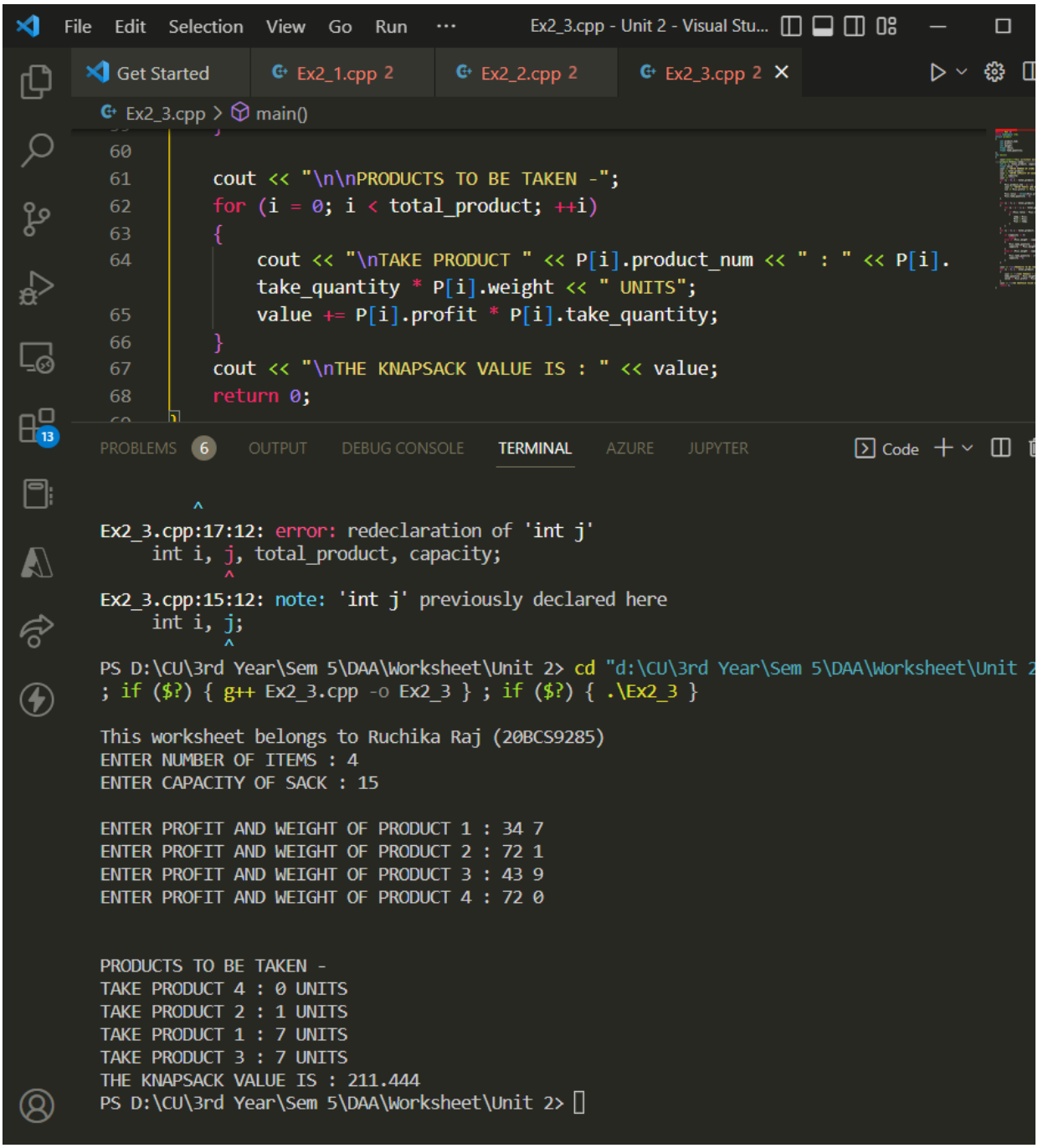
```
            if (P[i].ratio < P[j].ratio)
            {
                temp = P[i];
                P[i] = P[j];
                P[j] = temp;
            }
        }
    }
    for (i = 0; i < total_product; ++i)
    {
        if (capacity == 0)
            break;
        else if (P[i].weight < capacity)
        {
            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 θ(n, w) times as table c has (n + 1).(w + 1) entries, where each entry requires θ(1) time to compute .

6. 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.