**Worksheet-2.3**

**Student Name:-** Pushpraj Roy **UID:-** 20BCS9866

**Branch:-** BE- CSE **Section/Group:-** WM\_617 “A”

**Subjetct Code:-** 20CSP-312 **Semester:-** 5th

**Subject Name:-** DAA Lab

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

Code to implement 0-1 Knapsack using Dynamic Programming.

# Task to be done/ Which logistics used :-

Dynamic-0-1-knapsack Problem.

1. **Algorithm/Flowchart :-**

* Calculate the profit-weight ratio for each item or product.
* Arrange the items on the basis of ratio in descending order.
* Take the product having the highest ratio and put it in the sack.
* Reduce the sack capacity by the weight of that product.
* Add the profit value of that product to the total profit.
* 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]

1. **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;

}

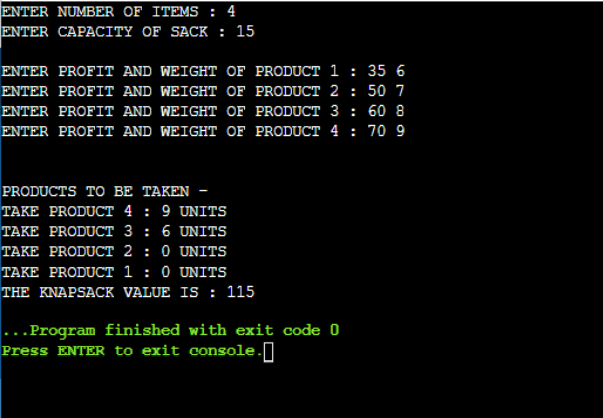
cout<<"\nTHE KNAPSACK VALUE IS : "<<value;

return 0;

}

1. **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 :-**