**Aim** :

A test has N questions with a heterogeneous distribution of points. The test taker has a choice as to which questions can be answered. Each question Qi has points Pi and time Ti to answer the question, where 1<=i<=N. The students are asked to answer the possible subsets of problems whose total point values add up to a maximum score within the time limit T. Determine which subset gives student the highest possible score

**Description** :

* We have been given N questions with each question having it’s corresponding points and time to answer the question.
* The task here is to find the questions that has to be answered by the students which add up to a maximum score.
* In order to solve this we follow a greedy approach which gives us a feasible solution but may or may not give us a optimal solution.
* Here we pick up the questions with respect to decreasing order of points/time ratio and keep decreasing the time of the question chosen from the maximum time.
* Here, the student either has to pick up a question or leave it in such a way that the questions he answered adds up to a maximum value.

**Algorithm** :

Algorithm 01Knapsack(m,n)

//p[1:n] and w[1:n] contain the profits and weights respectively

// of the n objects ordered such that p[i]/w[i]>= p[i+1]/w[i+1].

// m is the knapsack size and x[1:n] is the solution vector.

{

for i=1 to n do x[i]=0.0 U=m

for i=1 to n do

if(w[i]>U) then break; x[i]=1

U=U-w[i]

}

**Code** :

#include <bits/stdc++.h>

using namespace std;

class obj{

public:

int weight;

int value;

};

bool cmp(obj a,obj b){

double r1 = (double)a.value/(double)a.weight;

double r2 = (double)b.value/(double)b.weight;

return r1>r2;

}

void knapsack(int max\_weight,obj list[],int num){

/\*for(int i=1;i<=num;i++){

cout << list[i].value << list[i].weight << endl;

}

cout << endl;\*/

sort(list,list+num,cmp);

double pweight=0,pval=0;

for(int i=0;i<num;i++){

if(pweight+list[i].weight<=max\_weight){

pweight+=list[i].weight;

pval+=list[i].value;

cout<<list[i].weight<<"/"<<list[i].value<<"\t\t\t\t\t"<<1<<"\n";

}

else{

// int frac\_weight=max\_weight-pweight;

// pweight = max\_weight;

// pval += ((double)frac\_weight/(double)list[i].weight)\*list[i].value;

// cout<<list[i].weight<<"/"<<list[i].value<<"\t\t\t\t\t"<<(double)frac\_weight/(double)list[i].weight<<"\n";

// break;

cout<<list[i].weight<<"/"<<list[i].value<<"\t\t\t\t\t"<<0<<"\n";

}

}

cout << "Maximum Value: " << pval;

}

int main(){

int num;

cout << "Enter the number of objects : ";

cin >> num;

int mweight;

cout << "Enter the maximum weight : ";

cin >> mweight;

int val,weight;

obj list[num];

cout<<"Profits Weights :\n";

for(int i=0;i<num;i++){

// cout << "Enter points for question " << i+1 << ": ";

// cin >> val;

// cout << "Enter time for question " << i+1 << ": ";

// cin >> weight;

val=rand()%300;

weight=rand()%mweight;

list[i].value=val;

list[i].weight=weight;

cout<<val<<" "<<weight<<"\t";

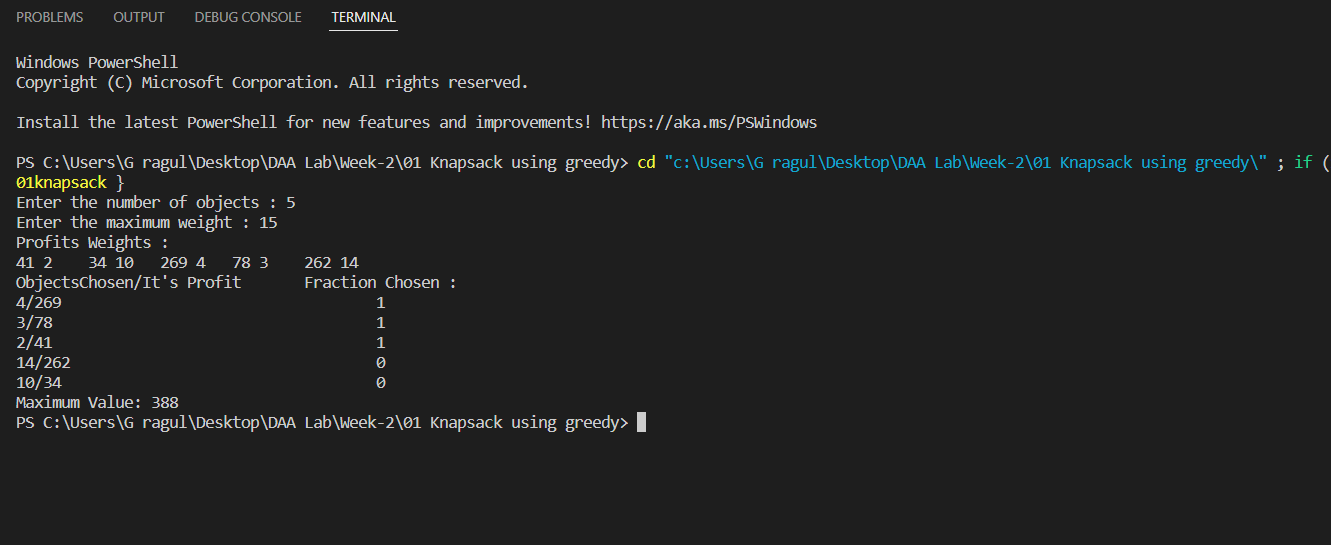
}

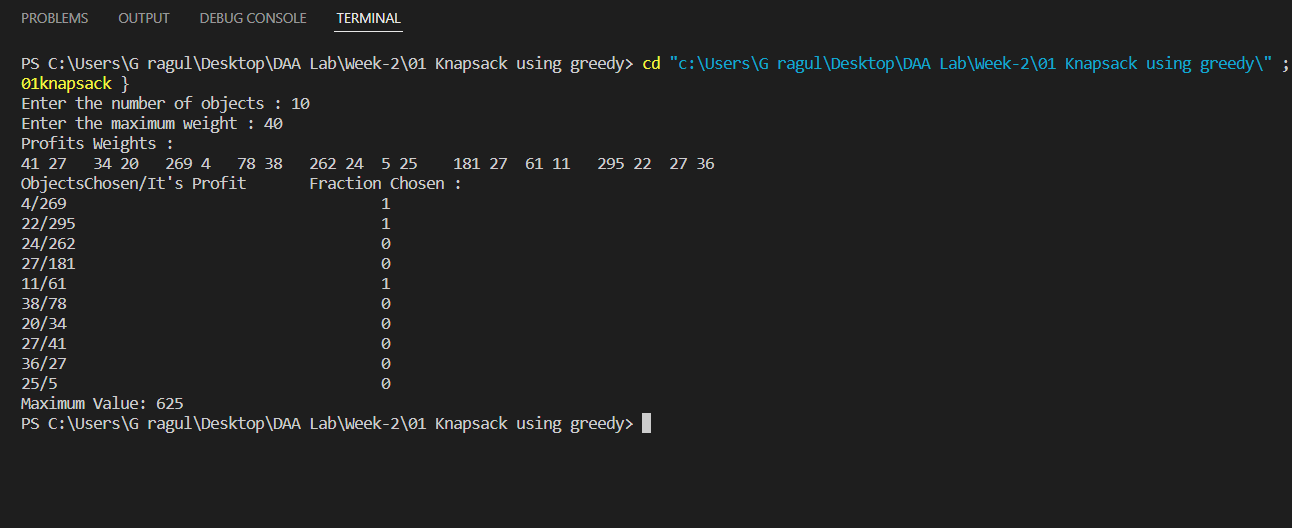
cout<<"\nObjectsChosen/It's Profit\tFraction Chosen : \n";

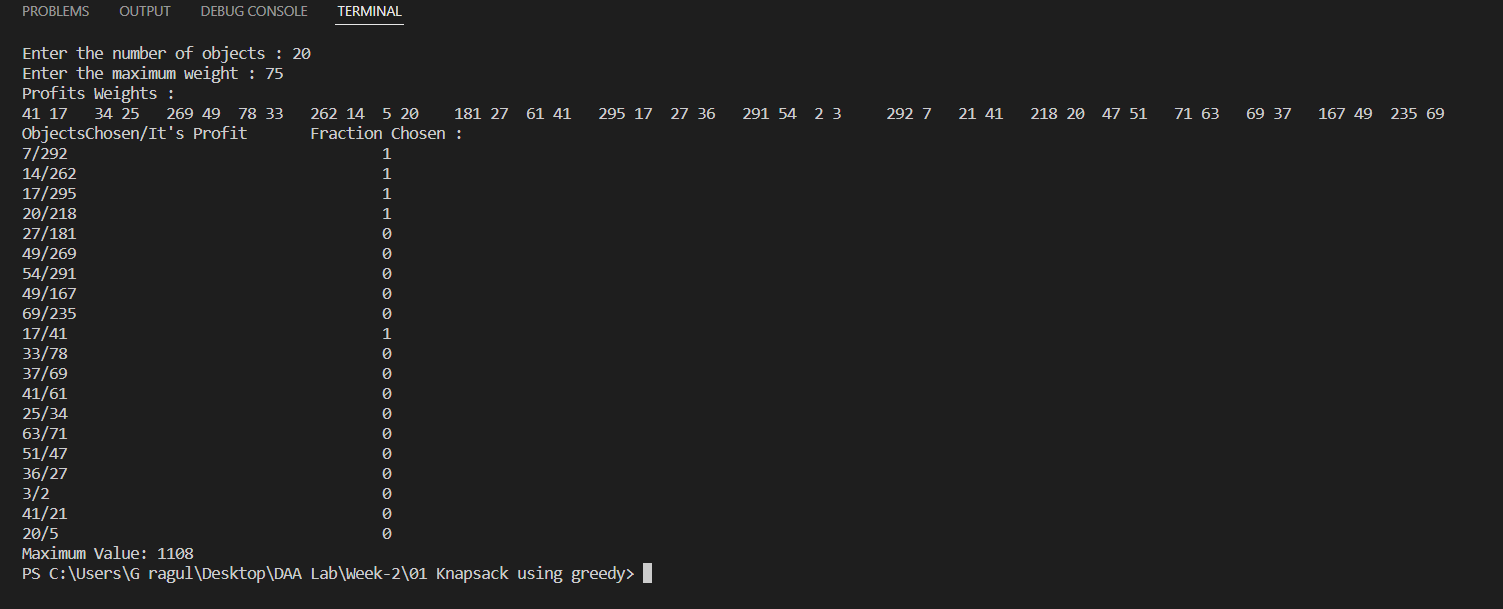
knapsack(mweight,list,num);

}

**Result Analysis** :







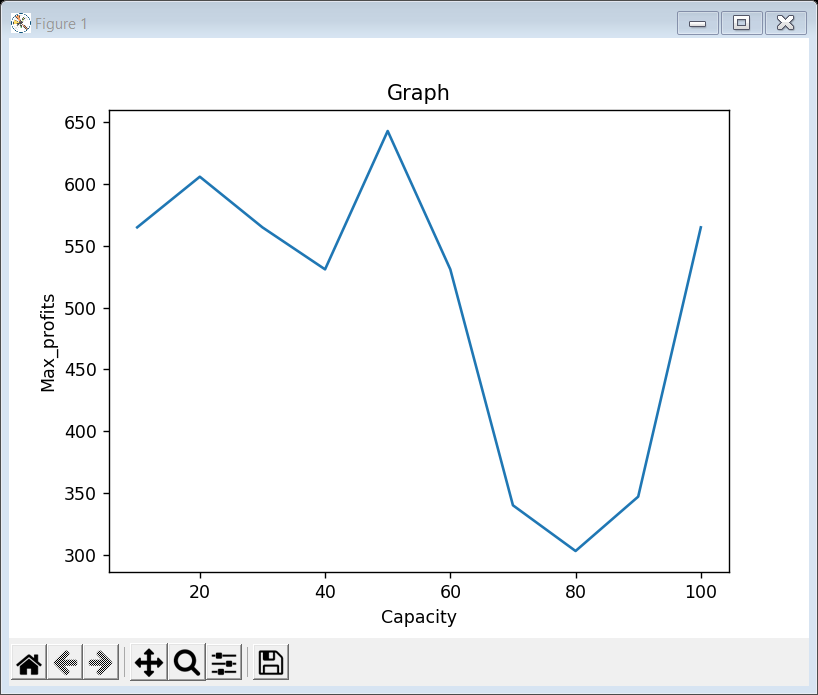
**Time Complexity** :

Time take by sort() function in C++ is nlogn and therefore total time comp. from the code is n+nlogn which overall leads to O(nlogn), where n is the no. of objects .

**Space Complexity** : O(n)

where n is the no. of objects .

**Graph** :



**Conclusion** :

The greedy approach for the 0/1 knapsack gives us a feasible solution but not an optimal solution.

In order to achieve optimal solution we follow dynamic programming approach.