**Aim:**

Given N items with their corresponding weights and values, and a package of capacity C, choose either the entire item or fractional part of the item among these N unique items to fill the package such that the package has maximum value.

**Description:**

* Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.
* In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem.
* A **brute-force solution** would be to try all possible subset with all different fraction but that will be too much time taking.
* An **efficient solution** is to use Greedy approach.
* The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio.
* Then take the item with the highest ratio and add them until we can’t add the next item as a whole and at the end add the next item as much as we can.
* Which will always be the optimal solution to this problem.  
  A simple code with our own comparison function can be written as follows, please see sort function more closely, the third argument to sort function is our comparison function which sorts the item according to value/weight ratio in non-decreasing order.
* After sorting we need to loop over these items and add them in our knapsack satisfying above-mentioned criteria.

**Algorithm**:

Algorithm GreedyKnapsack(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]

}

if(i<=n) then x[i]=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;

int i;

for(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;

}

}

for(int j=i+1;j<num;j++){

cout<<list[j].weight<<"/"<<list[j].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 value of object" << i+1 << ": ";

// cin >> val;

// cout << "Enter weight of object" << i+1 << ": ";

// cin >> weight;

val=rand()%300+1;

weight=rand()%mweight+1;

list[i].value=val;

list[i].weight=weight;

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

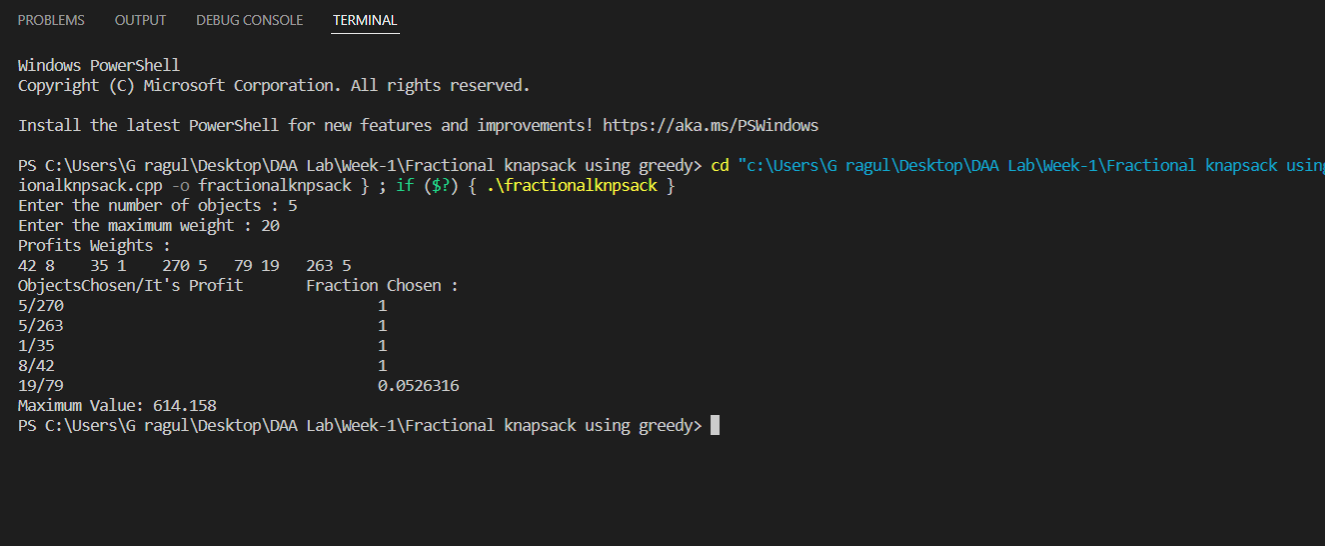
}

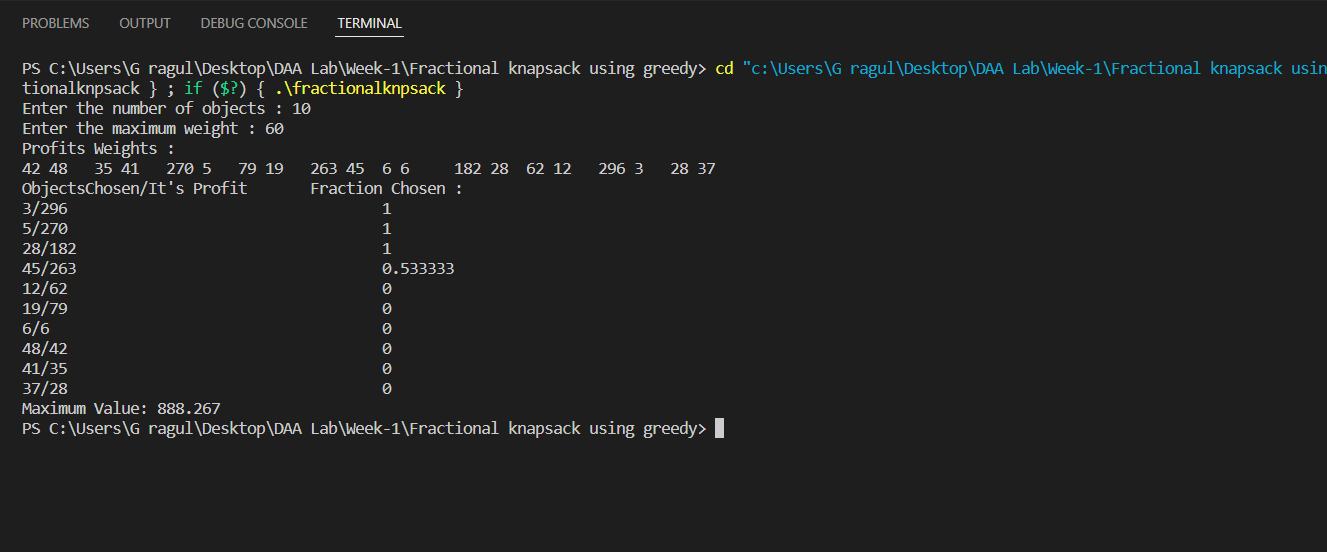
cout<<"\nObjectsChosen/It's Profit\tFraction Chosen : \n";

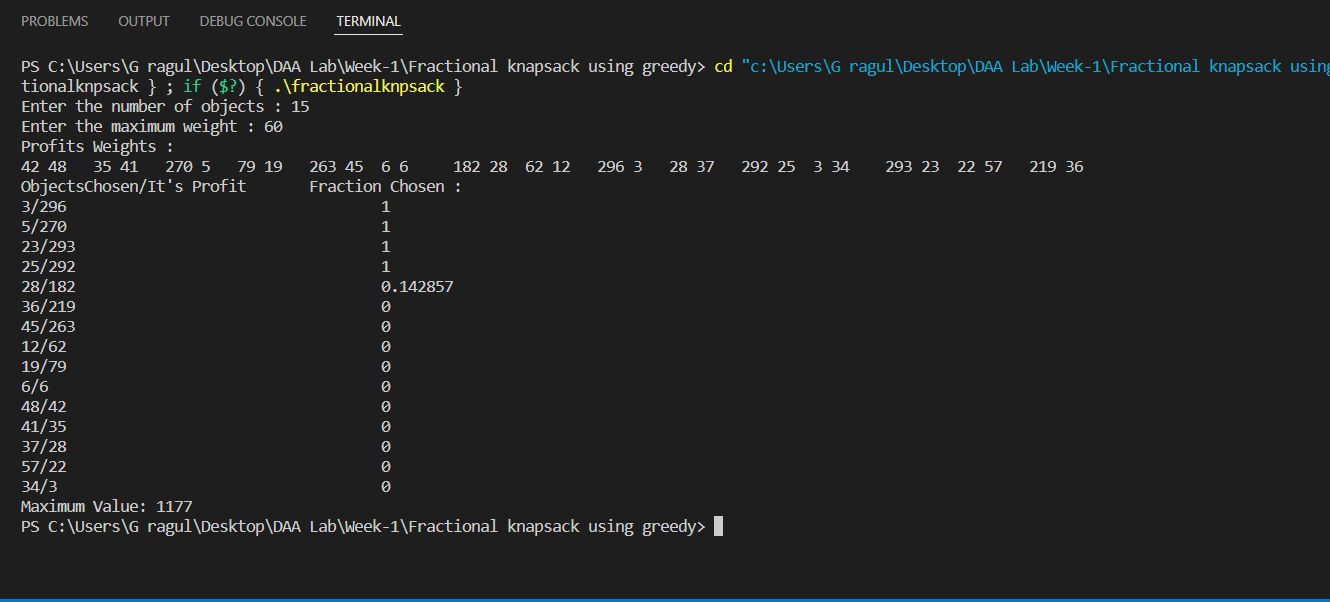
knapsack(mweight,list,num);

}

**Result Analysis :**

****

****

****

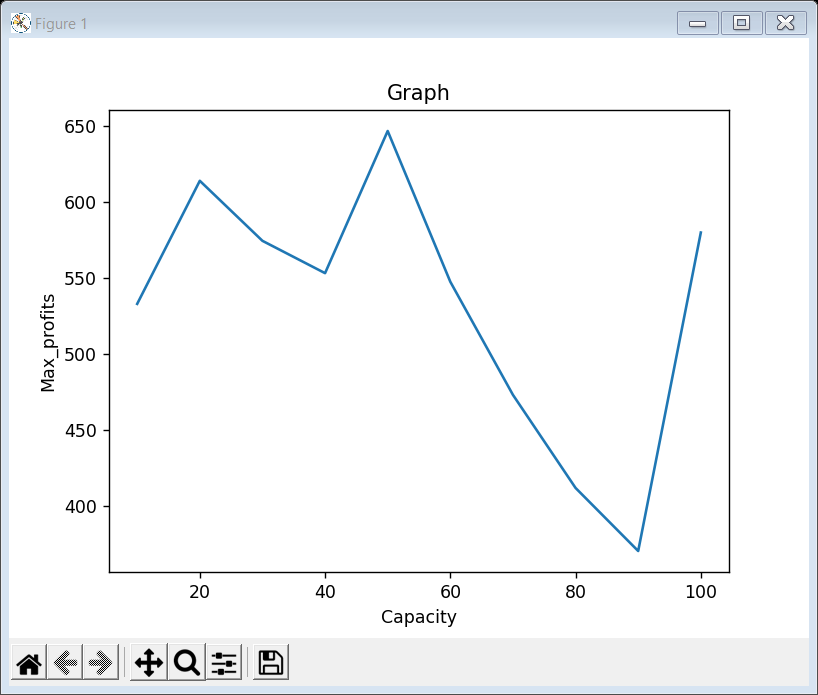
**Time Complexity :**

The algorithm uses sorting to sort the items which takes O(n×logn) time complexity and then loops through each item which takes O(n).Hence summing up to a time complexity of O(n×logn+n)=O(n×logn).If the items are already sorted then it takes O(n) time complexity.

**Space Complexity** : O(n)

where n is the no. of objects.

**Graph** :



**Conclusion:**

From the above outputs we can see that the maximum profit increases as we increase values and it is less dependent on weights and no\_of\_items , though we increase max\_weight and no\_of\_items the max\_profit nearly remains same.