**Aim** :

A Test has ‘N’ questions with a heterogenous 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 whose total points values add up to a maximum score within the time limit ‘T’. Determine which subset of questions gives student the highest possible score.

**Description** :

The 0/1 knapsack problem means the items are completely picked or not picked. For example, if we have say 3 bags of size 1kg,5kg and 3kg and if the knapsack size is 2kg(say). We can pick 1kg item completely but we can’t pick up fractional part of 5kg and 3kg, it has to be completely picked or left.

This is called as 0/1 knapsack. We can solve this using Dynamic Programming strategy which always gives us an optimal solution.

**Algorithm** :

Algorithm knapsack(p,t,W,N) :

//where p,t are profits and weights array of size N respectively, W is the

//knapsack size.

{

B[n+1][W+1]:=0

for i:=0 to n do{

for w:=0 to W do{

if(i==0 or w==0) then B[i][w]:=0

else if(t[i-1]<=w) then B[i][w]:=max(p[i-1]+B[i-1][w-t[i-1]],B[i-1][w]);

else B[i][w]:=B[i-1][w]

return B[n][W]

}

**Code** :

#include <bits/stdc++.h>

using namespace std;

class obj{

public:

int time;

int points;

};

bool cmp(obj a,obj b){

double r1 = (double)a.points/(double)a.time;

double r2 = (double)b.points/(double)b.time;

return r1>r2;

}

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

int B[num+1][max\_time+1];

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

for(int w=0;w<=max\_time;w++){

if(i==0||w==0){

B[i][w]=0;

}

else if(list[i-1].time<=w){

if(list[i-1].points+B[i-1][w-list[i-1].time]>B[i-1][w])

B[i][w]=list[i-1].points+B[i-1][w-list[i-1].time];

else

B[i][w]=B[i-1][w];

}

else

B[i][w]=B[i-1][w];

}

}

//return B[num][max\_time];

int max\_score = B[num][max\_time];

cout << "Maximum Score= " << max\_score << endl;

cout << "Questions: ";

for(int i=num;i>0;i--){

if(max\_score>0){

if(max\_score!=B[i-1][max\_time]){

cout << i << " ";

max\_score-=list[i-1].points;

max\_time-=list[i-1].time;

}

}

}

}

void randgen(obj list[], int num,int mtime){

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

list[i].points=(rand()%300)+1;

list[i].time=(rand()%mtime)+1;

cout<<list[i].points<<"\t"<<list[i].time<<"\n";

}

cout << endl;

}

int main(){

int num;

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

cin >> num;

int val,time;

obj list[num];

int mtime;

cout << "Enter the maximum time: ";

cin >> mtime;

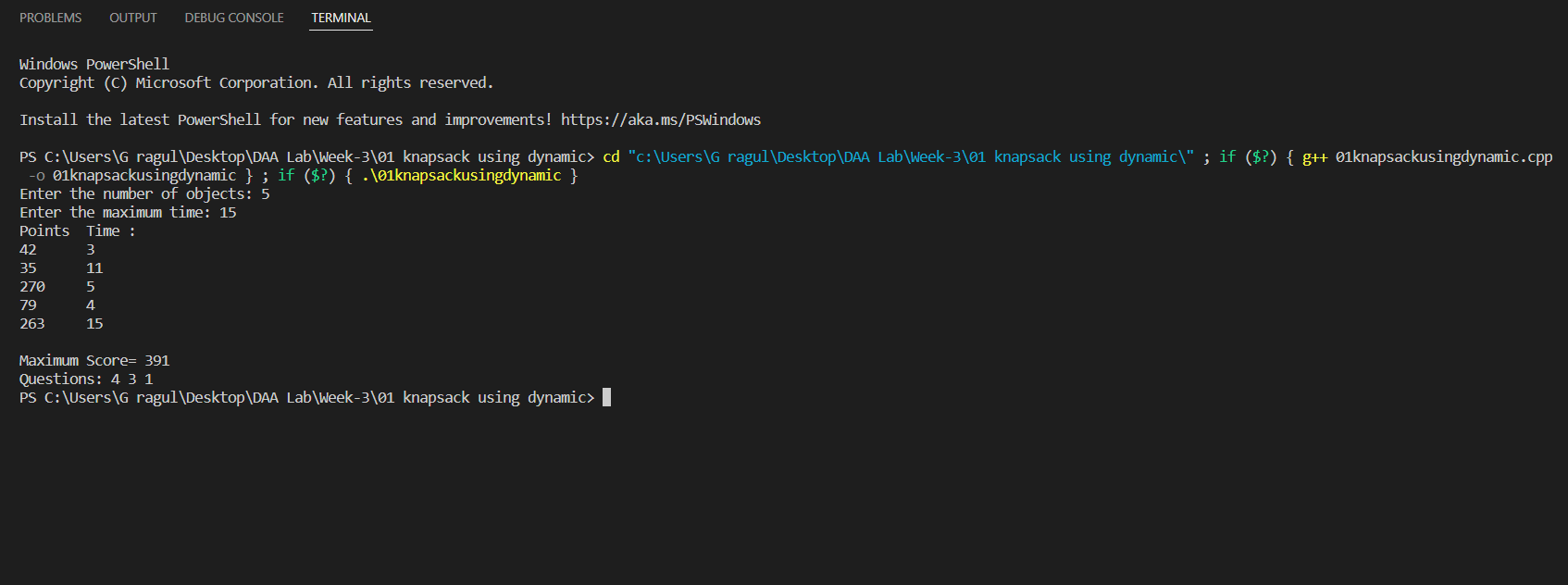
cout<<"Points\tTime : \n";

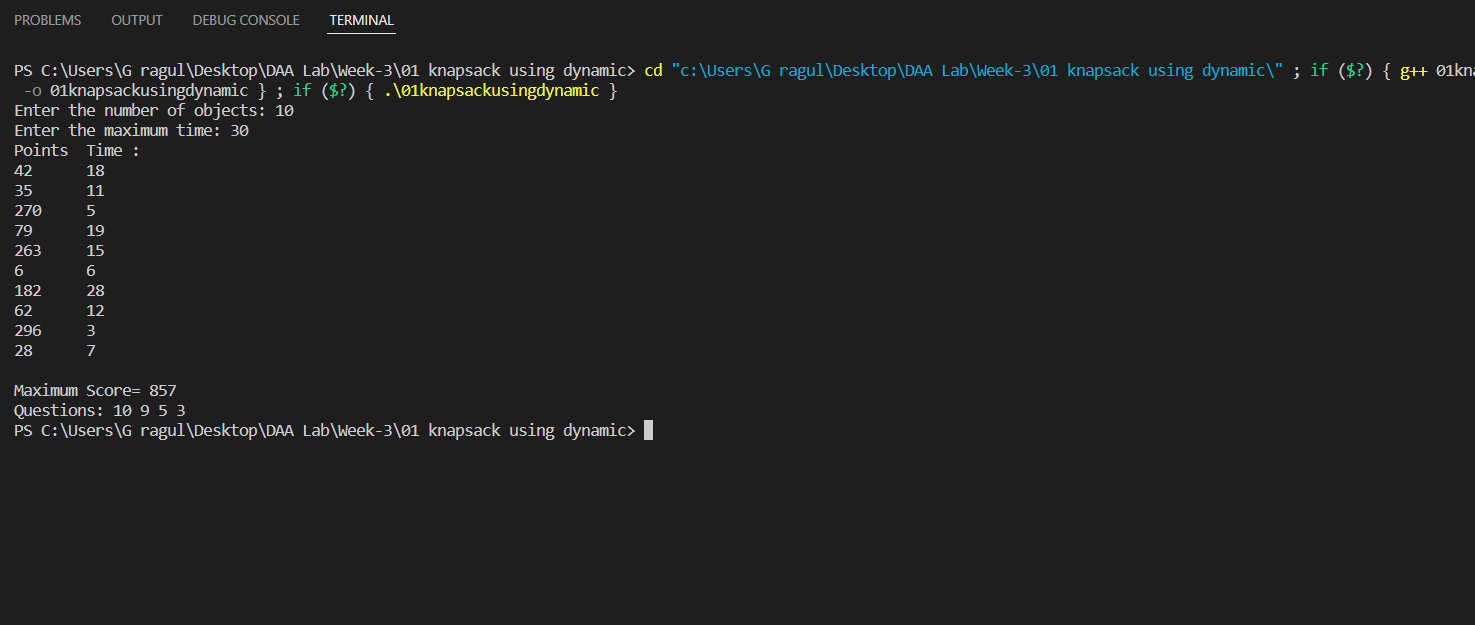
randgen(list,num,mtime);

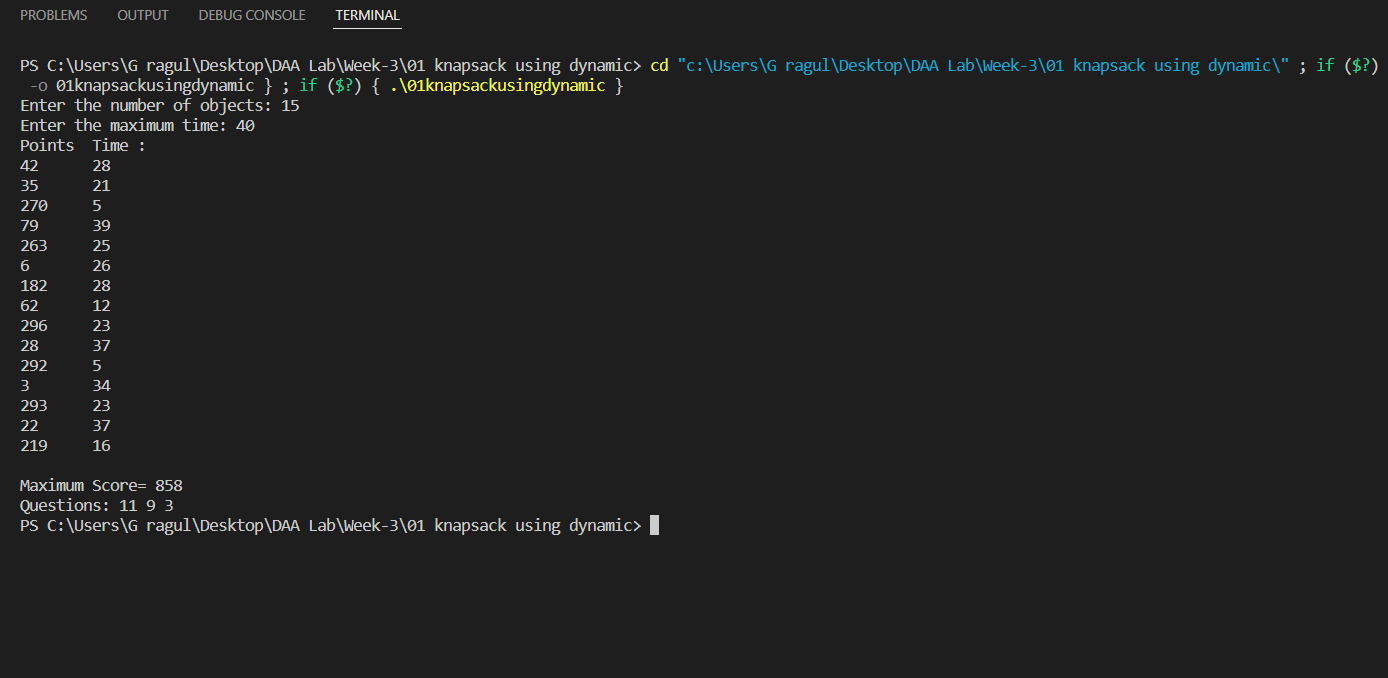
knapsack(mtime,list,num);

}

**Result Analysis** :







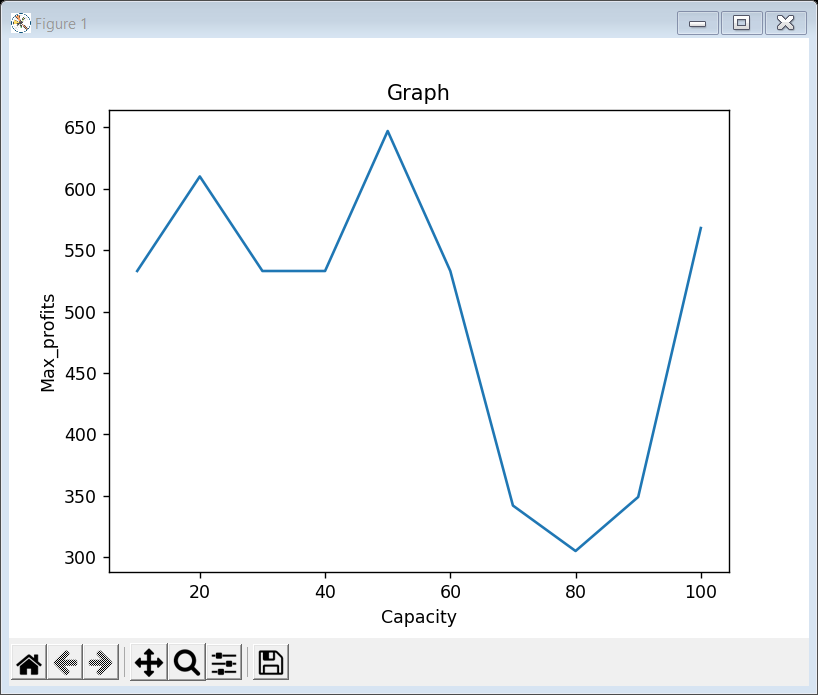
**Time Complexity** : O(n\*w)

If N is the number of elements and W is the knapsack size then time complexity is O(N\*W)

**Space Complexity** : O(N\*W)

Because of the use of 2D-Array of size (N+1)\*(W+1)

**Graph** :



**Conclusion** :

The difference in solving 01 knapsack from greedy approach to that of dynamic programming method is that, greedy method here always doesn’t give an optimal solution whereas dynamic method does. In greedy method we don’t think about the future , we select what is best and optimal in the current situation and then proceed whereas in dynamic prog. method we try out all the possible methods and then proceed with an optimal solution.

**Analysis** :

The greedy approach for 01 knapsack gives us a feasible solution but not an optimal solution but dynamic prog. approach gives us an optimal solution but the only disadvantage in this particular approach is that the time comp. here increases as we go to the future and then decide which objects to pick up, also there is a increase in space compared to greedy as here we use a 2D array.

The time taken in greedy approach is O(nlogn) and in in dynamic approach is O(n\*w).

The space compl. in greedy approach is O(n) whereas in dynamic approach is O(n\*w).

|  |  |  |
| --- | --- | --- |
| **Name of Strategy Used** | **Time Complexity** | **Space Complexity** |
| Greedy | O(nlogn) | O(n) |
| Dynamic | O(n\*w) | O(n\*w) |

As now the devices we use have come up with a larger space capacity we generally don’t worry about the amount of space occupied.