**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.
* In this solution we would be trying it to use the dynamic programming approach, so after all the objects are picked by using dynamic approach of 01 knapsack then we send the remaining objects to the fractional knapsack in order to find out the fraction of object picked up.

**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]

}

}

max\_value=B[n][W]

maxi=max\_value

for i:=n to 0 do{

if(max\_value>0) then

if(max\_value!=B[i-1][W]) then

max\_value-= list[i - 1].profits

max\_weight -= list[i - 1].weight

solution\_set[i - 1] = 1

else: solution\_set[i-1]=0

else:

solution\_set[i-1]=0

for i:=0 to n do{

if(solution\_set[i]==0) then a.push\_back(list[i])

}

knapsack\_greedy(max\_weight, a, a.size())

}

**Code:**

#include <bits/stdc++.h>

using namespace std;

class obj

{

public:

int weight;

int profits;

};

bool cmp(obj a, obj b)

{

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

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

return r1 > r2;

}

double knapsack\_greedy(int max\_weight, vector<obj> list, int num)

{

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

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

}

cout << endl;\*/

sort(list.begin(), list.end(), cmp);

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

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

// }

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].profits;

cout << list[i].weight << "/" << list[i].profits << "\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].profits;

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

}

return pval;

}

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

{

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

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

{

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

{

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

{

B[i][w] = 0;

}

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

{

if (list[i - 1].profits + B[i - 1][w - list[i - 1].weight] > B[i - 1][w])

B[i][w] = list[i - 1].profits + B[i - 1][w - list[i - 1].weight];

else

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

}

else

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

}

}

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

int max\_value = B[num][max\_weight];

double maxi = max\_value;

cout << "Maximum Value : " << max\_value << endl;

cout << "ObjectsChosen/Profits\tFractionChosen: \n";

int solution\_set[num];

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

{

if (max\_value > 0)

{

if (max\_value != B[i - 1][max\_weight])

{

// cout << i << " ";

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

max\_value -= list[i - 1].profits;

max\_weight -= list[i - 1].weight;

solution\_set[i - 1] = 1;

}

else

{

solution\_set[i - 1] = 0;

}

}

else

{

solution\_set[i - 1] = 0;

}

}

vector<obj> a;

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

{

// cout<<solution\_set[i]<<" ";

if (solution\_set[i] == 0)

{

a.push\_back(list[i]);

}

}

double from\_greedy = knapsack\_greedy(max\_weight, a, a.size());

cout << "Maximum value : " << double(from\_greedy + maxi);

}

void randgen(obj list[], int num, int mweight)

{

cout << "Weights Profits : \n";

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

{

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

list[i].weight = (rand() % mweight) + 1;

cout << list[i].weight << " " << list[i].profits << "\t";

}

cout << endl;

}

int main()

{

int num;

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

cin >> num;

int val, weight;

obj list[num];

int mweight;

cout << "Enter the maximum weight : ";

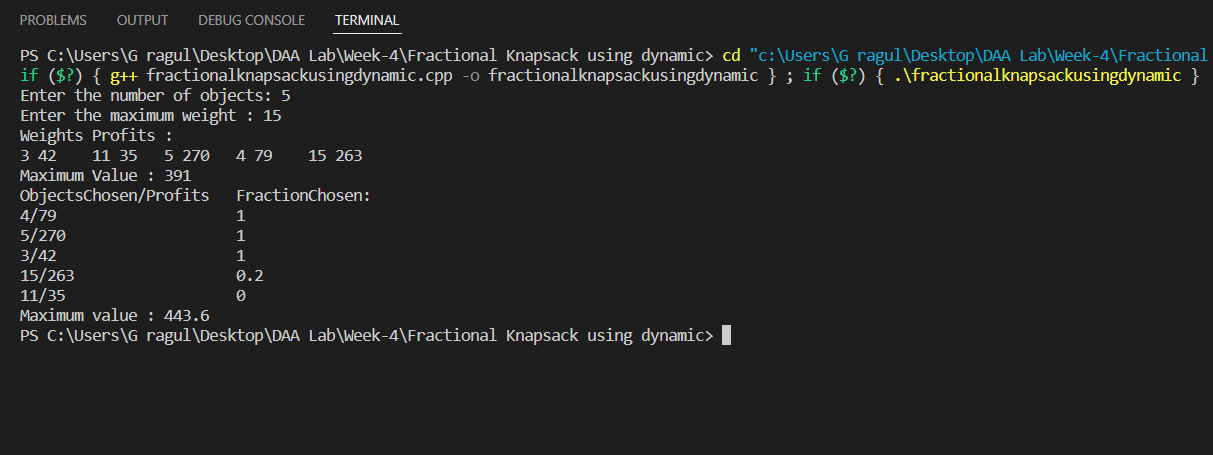
cin >> mweight;

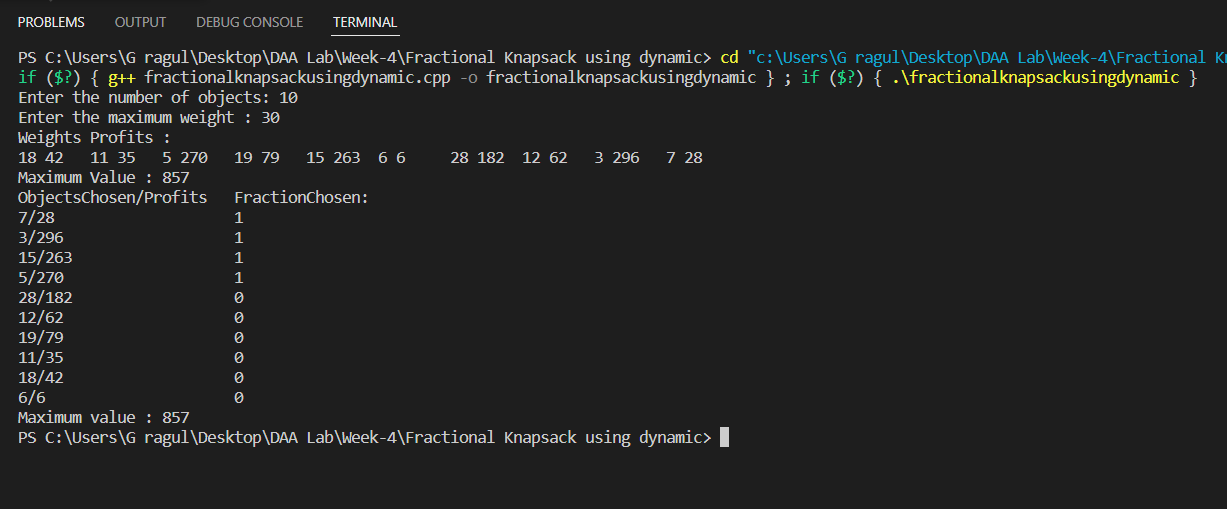
randgen(list, num, mweight);

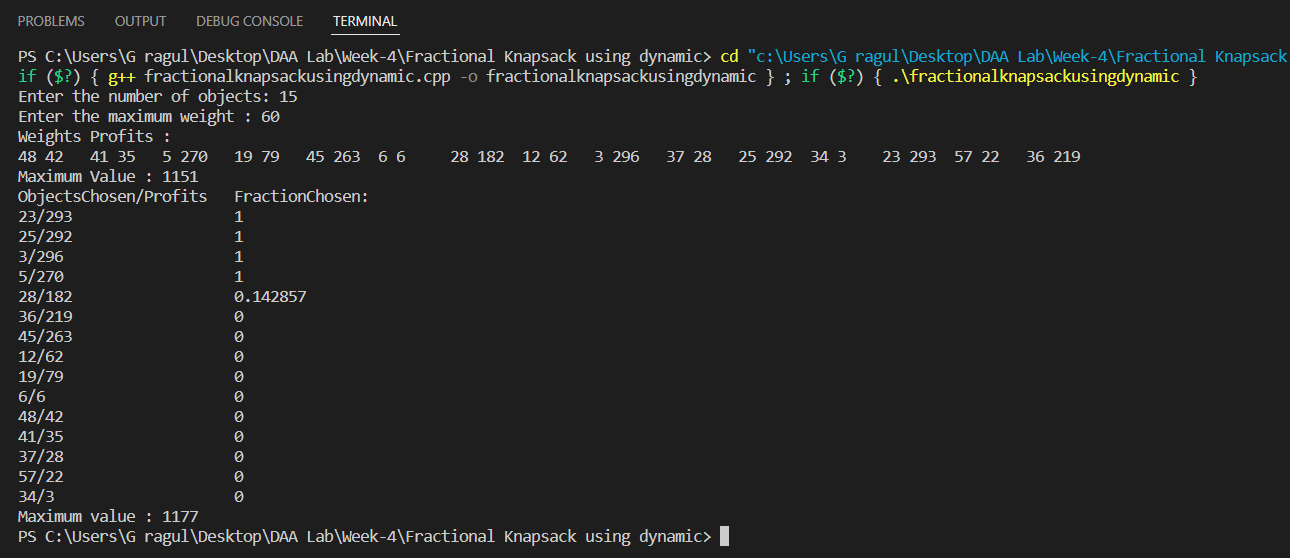
knapsack(mweight, list, num);

}

**Result Analysis :**

****

****

****

**Time Complexity :**

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

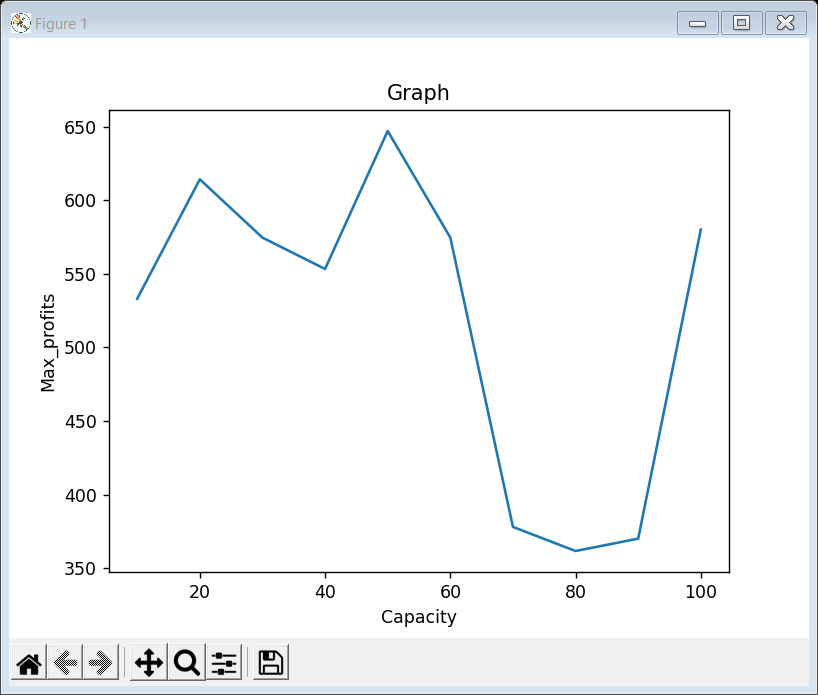
Then in greedy\_knapsack function the time comp. is O(NlogN) .

Therefore overall time complexity is O(N\*W+NlogN)

**Space Complexity** :

A 2D array for dynamic programming approach and an array of size n where n is the number of objects that weren’t picked and sent it to greedy\_knapsack function . Therefore, overall time complexity is O(N\*W+n).

**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.

**Analysis** :

The greedy approach though gives us an optimal solution in fractional knapsack , we just try to apply the dynamic approach in order to compare the time and space complexities of both the approaches.

|  |  |  |
| --- | --- | --- |
| **Name of the Strategy Used** | **Time Complexity** | **Space Complexity** |
| Greedy | O(nlogn) | O(n) |
| Dynamic | O(N\*W+NlogN) | O(N\*W+n) |

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.