**Experiment No: 5**

**AIM:** Implementation of Knapsack Problem(Greedy Approach) and obtaining its step count.

**THEORY:**

Greedy method is one of the most straightforward design techniques which can be applied on a variety of problems, most of which have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a **Feasible solution.** We need to find a feasible solution that either maximises or minimises a given objective function. A feasible solution that does this is called the **Optimal Solution.**

This method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added. Otherwise, it is added.

Given weights and profits of n items, we need to put these items in a **knapsack** of capacity W to get the maximum total profit in the knapsack. This is the Fractional Knapsack problem that we solve using greedy approach.

The basic idea of the greedy approach is to calculate the **profit/weight ratio** 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 cannot 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.

**ALGORITHM:**

**Algorithm GreedyKnapsack (m, n)**

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

// Objects ordered so 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 // initialize x

U := m;

for i := 1 to n do

{

if (w(i) > U) then break;

x [i] := 1.0; U := U – w[i];

}

if (i < n) then x[i] := U / w[i];

}

**Sample Example tracing:**

Consider the following instance of the knapsack problem: n = 3, m = 20, (p1, p2, p3) =(25, 24, 15) and (w1, w2, w3) = (18, 15,10).

1. First, we try to fill the knapsack by selecting the objects in some order:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x1 | x2 | x3 | wi xi | pi xi |
| 1/2 | 1/3 | 1/4 | 18x1/2+15x1/3+10x1/4  **= 16.5** | 25x1/2+24x1/3+15x1/4 **=24.25** |

2. Select the object with the maximum profit first (p = 24). So, x1 = 1 and profit earned is 24. Now, only 2 units of space is left, select the object with next largest

profit (p = 20). So, x2 = 2/15

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x1 | x2 | x3 | wi xi | pi xi |
| 1 | 2/15 | 0 | 18\*1+15\*2/15+10\*0= **20** | 25\*1+24\*2/15+15\*0= **28.2** |

1. Considering the objects in the order of non-increasing ratio **pi/wi**.

|  |  |  |
| --- | --- | --- |
| P1/W1 | P2/W2 | P3/W3 |
| 25/18 | 24/15 | 15/10 |
| 1.4 | 1.6 | 1.5 |

Select the object with the maximum pi / xi ratio, so, x2 = 1 and profit earned is 24. Now, only 5

units of space is left, select the object with next largest pi / xi ratio, so x3 = ½ and the profit earned is 7.5.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x1 | x2 | x3 | wi xi | pi xi |
| 0 | 1 | 1/2 | 18\*0+15\*1+10\*1/2= **20** | 25\*0+24\*2/15+15\*1/2= **31.5** |

PROGRAM IMPLEMENTATION:

#include<iostream>

using namespace std;

int count=0;

void sort(float \*a,float \*w,float \*p,int n)

{

for(int i=1;i<n;i++)

{

count++;

for(int j=0;j<n-i;j++)

{

count+=2;

if(a[j]<a[j+1])

{

float t = a[j];

a[j] = a[j+1];

a[j+1] = t;

t = w[j];

w[j] = w[j+1];

w[j+1] = t;

t=p[j];

p[j]=p[j+1];

p[j+1]=t;

count+=9;

}

}

count++; //inner for

}

count++; //outer for

}

void greedyknapsack(int m,int n)

{

float \*p = new float[n];

float \*w = new float[n];

int u=m;

count+=3; //assign

cout<<"Enter the profits of the "<<n<<" objects\n";

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

cin>>p[i];

cout<<"Enter the weights of the "<<n<<" objects\n";

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

{

count+=2;

cin>>w[i];

}

count++; //for

float \*pw = new float[n],\*x = new float[n];

count+=2;

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

{

count+=2;

pw[i] = p[i]/w[i];

}

count++;

sort(pw,w,p,n);

cout<<"The new order of objects:"<<endl;

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

{

count+=2;

cout<<"P"<<i+1<<"="<<p[i]<<" "<<"W"<<i+1<<"="<<w[i]<<endl;

}

count++; //for

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

{

count+=2;

x[i] = 0.0;

}

count++; //for

int i=0; count++;

for(i=0;i<n;i++)

{

count+=2;

if(w[i]>u)

{

count++;

break;

}

x[i] = 1.0;

u-=w[i];

count+=2;

}

count++;

count++; //if

if(i<=n)

{

count++;

x[i] = u/w[i];

}

cout<<"\nThe solution vector is\n";

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

cout<<x[i]<<" ";

cout<<"\nHence, the maximum possible profit is:";

float sum=0.0;

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

sum+=x[i]\*p[i];

cout<<sum<<endl;

}

int main()

{

int n,m;

cout<<"Enter number of objects and the weight of the bag\n";

cin>>n>>m;

greedyknapsack(m,n);

cout<<"Count="<<count<<endl;

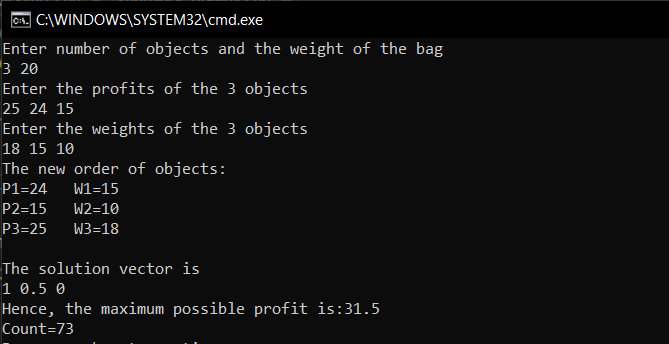
return 0;

}

OUTPUTS:

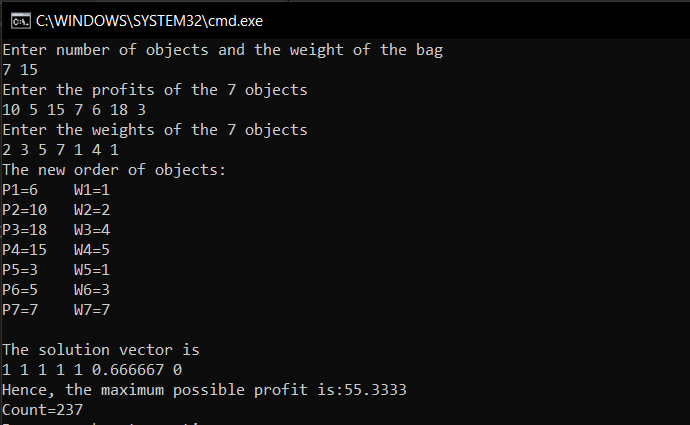
1. When n=3

**Count=73**

****

1. When n=7

**Count=237**

****

**Conclusion**:

1. **The objects are to be sorted into non-decreasing order of pi / wi ratio. But if we**

**disregard the time to initially sort the objects, the algorithm requires only O(n) time.**

1. **The Greedy Approach to solve this problem works only when the objects’ fractional weight can be considered. It will fail if there cannot be a division of their weights.**