# Experiment Number: 5

**Problem Statement:-** Implementation of Fractional Knapsack problem.

**Aim:-**Write a program to solve Fractional Knapsack problem using Greedy method.

**Theory:-**

There are n items in a store. For i =1,2. . . n. Item i have weight $w_i > 0$ and worth $p_i > 0$. We can carry a maximum weight of $W$ in a knapsack. In this version of a problem the items can be broken into smaller piece, so that we can decide to carry only a fraction xi of object i, where 0 ≤ xi ≤ 1. Item $i$ contribute $w_i x_i$ to the total weight in the knapsack, and $p_i x_i$ to the value of the load.

In Symbol, the fraction knapsack problem can be stated as follows.

Maximize

$$\sum_{i=1}^{n} x_i p_i$$

Subject to constraint

$$\sum_{i=1}^{n} w_i x_i < W$$

It is clear that an optimal solution must fill the knapsack exactly, for otherwise we could add a fraction of one of the remaining objects and increase the value of the load.

**Algorithm:-**

Algorithm main()

{

Read the weight w [1: n] and profit p [1: n] of all the n items.

Calculate the ratio [1: n] for all the objects as p [1: n] / w [1: n]

Arrange all the objects in the decreasing order of ratios.

Call the prims function.

}

Algorithm prims()

{

For i = 1 to n do

if w[i] > W then break;

else

x[i]=1

tp=p[i]

W=W-w[i]

if ( i < n ) then x [i] = W / w[i] and tp = tp + x[i] * p [i]

Display x [1: n] and tp.

**Analysis:-**

The time required to sort objects into decreasing order of $p_i x_i$ is *O(n log n)*. Then the while loop takes a time in *O(n)*.Therefore, the total time including the sort is in *O(n log n)*.
If we keep the items in heap with largest $v_i/w_i$ at the root. Then

- creating the heap takes *O(n)* time
- while-loop now takes *O(log n)* time (since heap property must be restored after the removal of root).
- Although this data structure does not alter the worst-case, it may be faster if only a small number of items are need to fill the knapsack.

NAME:HARSHBHAI SOLANKI
ROLL.N0:62
SE-4-D

# EXPERIMENT N0-5

## AIM: Implementation of Fractional Knapsack problem.

## CODE:

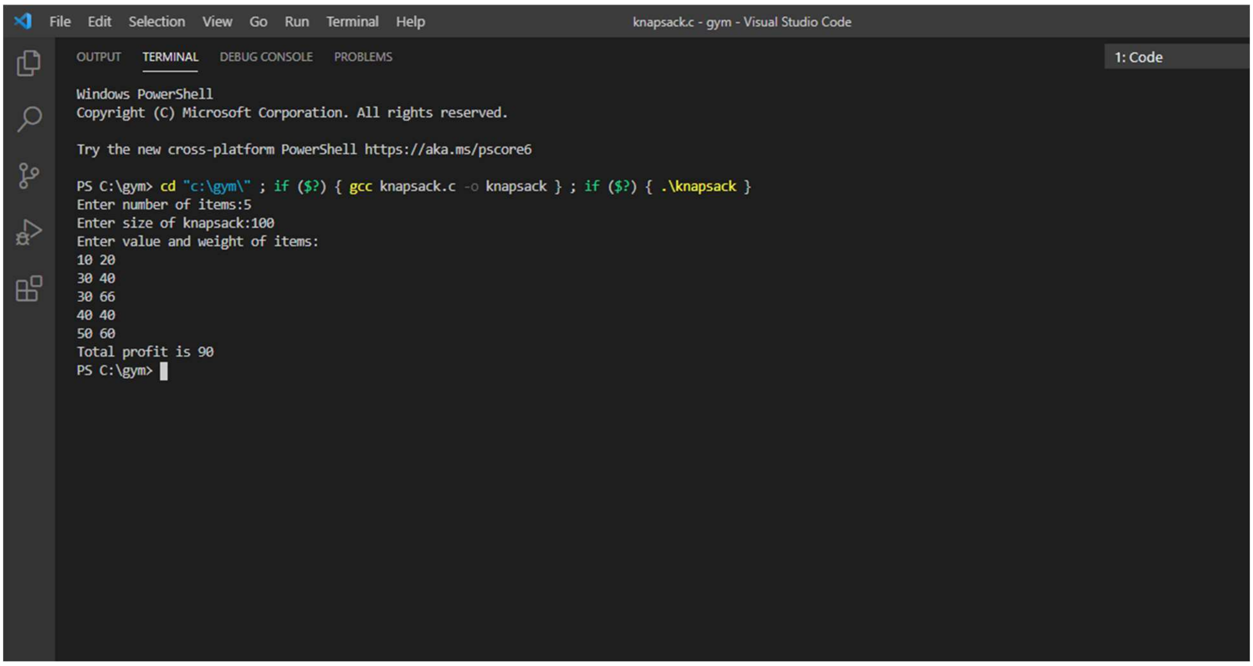
```
#include<stdio.h>
int max(int a, int b) { return (a > b)? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1];

   for (i = 0; i <= n; i++)
   {
       for (w = 0; w <= W; w++)
       {
           if (i==0 || w==0)
               K[i][w] = 0;
           else if (wt[i-1] <= w)
                 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
           else
                 K[i][w] = K[i-1][w];
       }
   }

   return K[n][W];
}
```

```
int main()
{
   int i, n, val[20], wt[20], W;
   printf("Enter number of items:");
   scanf("%d", &n);
   printf("Enter size of knapsack:");
   scanf("%d", &W);
   printf("Enter value and weight of items:\n");
   for(i = 0;i < n; ++i){
        scanf("%d%d", &val[i], &wt[i]);
   }
   printf("Total profit is %d", knapSack(W, wt, val, n));
   return 0;
}
```

## OUTPUT:



## CONCLUSION:

By performing above practical we can conclude that for n items, knapsack has $2^n$ choices. So brute force approach runs in $O(2^n)$ time. We can improve performance by sorting items in advance. Using merge sort or heap sort, n items can be sorted in $O(n\log_2 n)$ time. Merge sort and heap sort are non-adaptive and their running time is same in best, average and worst case. To select the items, we need one scan to this sorted list, which will take $O(n)$ time. Total time required is $T(n) = O(n\log_2 n) + O(n) = O(n\log_2 n)$