K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: E-2	**Roll No.: 16010123325**
Experiment No.__7____	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title:Study, Implementation, and Analysis of 0/1 Knapsack Problem.

Objective: To learn 0/1 Knapsack Problem using Dynamic Programming Approach. (Maximize the total value of selected items while ensuring their total weight does not exceed W.)

CO to be achieved:

CO 2	Describe various algorithm design strategies to solve different problems and analyse Complexity.

Books/ Journals/ Websites referred:

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf**
4. **http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/**
5. **http://www.mafy.lut.fi/study/DiscreteOpt/tspdp.pdf**
6. **https://class.coursera.org/algo2-2012-001/lecture/181**
7. **http://www.quora.com/Algorithms/How-do-I-solve-the-travelling-salesman-problem-using-Dynamic-programming**
8. **www.cse.hcmut.edu.vn/~dtanh/download/Appendix_B_2.ppt**
9. **www.ms.unimelb.edu.au/~s620261/powerpoint/chapter9_4.ppt**

Pre Lab/ Prior Concepts:
Data structures, Concepts of algorithm analysis

Historical Profile:

Dynamic Programming (DP) is used heavily in optimization problems (finding the maximum and the minimum of something). Applications range from financial models and operation research to biology and basic algorithm research. So the good news is that understanding DP is profitable. However, the bad news is that DP is not an algorithm or a data structure that you can memorize. It is a powerful algorithmic design technique.

The 0/1 Knapsack Problem is one of the most studied problems in computer science, operations research, and combinatorial optimization. Its origins and development reflect the evolution of mathematical optimization and computational techniques.

Historical Background

Origins in Resource Allocation:The knapsack problem has its roots in resource allocation problems dating back centuries, where individuals needed to maximize their gain (value) from limited resources (capacity).The term "knapsack" derives from the idea of filling a knapsack with items to achieve the greatest benefit without exceeding its weight limit.

Early Formalization: The problem was first mathematically formalized in the early 20th century as part of broader studies in combinatorics and optimization. It gained attention as a theoretical challenge in discrete mathematics.

Greedy Algorithms and Limitations:Researchers also explored greedy approaches for simplified variants (e.g., fractional knapsack).The greedy algorithm does not work for the 0/1 knapsack problem due to its inability to handle binary choices optimally.

New Concepts to be learned:

Application of algorithmic design strategy to any problem, dynamic Programming method of problem solving Vs other methods of problem solving, optimality of the solution, Optimal Binary Search Tree Problems and their applications

Theory:

Algorithm:

Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\, B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

It means, that the best subset of Sk that has total weight w is one of the two:

1) the best subset of Sk-1 that has total weight w, or

2) the best subset of Sk-1 that has total weight w-wk plus the item k

Algorithm :

```
Algorithm 01Knapsack(S, W):
    Input: set S of n items with benefit bi and weight wi; maximum weight W
    Output: benefit of best subset of S with weight at most W
    let A and B be arrays of length W + 1
    for w ← 0 to W do
    B[w] ← 0
for k ← 1 to n do
    copy array B into array A
    for w ← wk to W do
        if A[w−wk] + bk > A[w] then
            B[w] ← A[w−wk] + bk
return B[W]
```

Construct the solution

```
Algorithm KnapsackElements(A,n,W)
{
i=n, k=W;
While (i>0 && k>0)
    {
    if B[i,k]<> B[i-1,k]
        mark ith item in knapsack
        k=k-wi; i=i-1
    else i=i-1
```

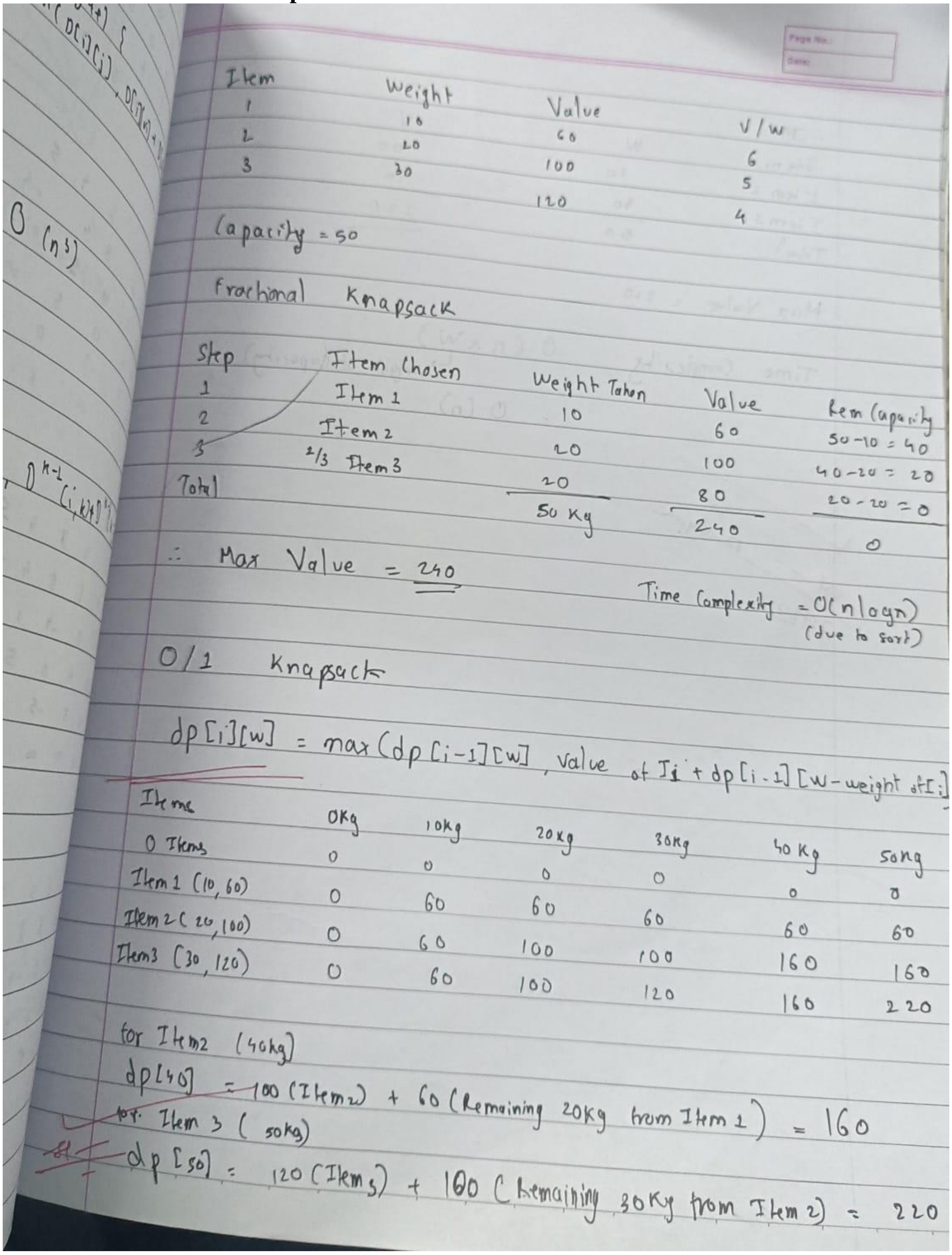
```
        }

}
```

Example:

<u>Truck Loading Optimization</u>
A logistics company needs to maximize profit while loading a truck with different packages. Each package has a weight and value. The truck has a weight limit, and fractional parts of a package can be taken.
Task: Write a function that selects packages using the Fractional Knapsack algorithm to maximize total value.

Input Example:
```
items = [(10, 60), (20, 100), (30, 120)]   # (weight, value)
capacity = 50
```

Solution for the example:



Item	Weight	Value	V/W
1	10	60	6
2	20	100	5
3	30	120	4

Capacity = 50

Fractional Knapsack

Step	Item Chosen	Weight Taken	Value	Rem Capacity
1	Item 1	10	60	50 − 10 = 40
2	Item 2	20	100	40 − 20 = 20
3	2/3 Item 3	20	80	20 − 20 = 0
Total		50 kg	240	0

∴ Max Value = 240

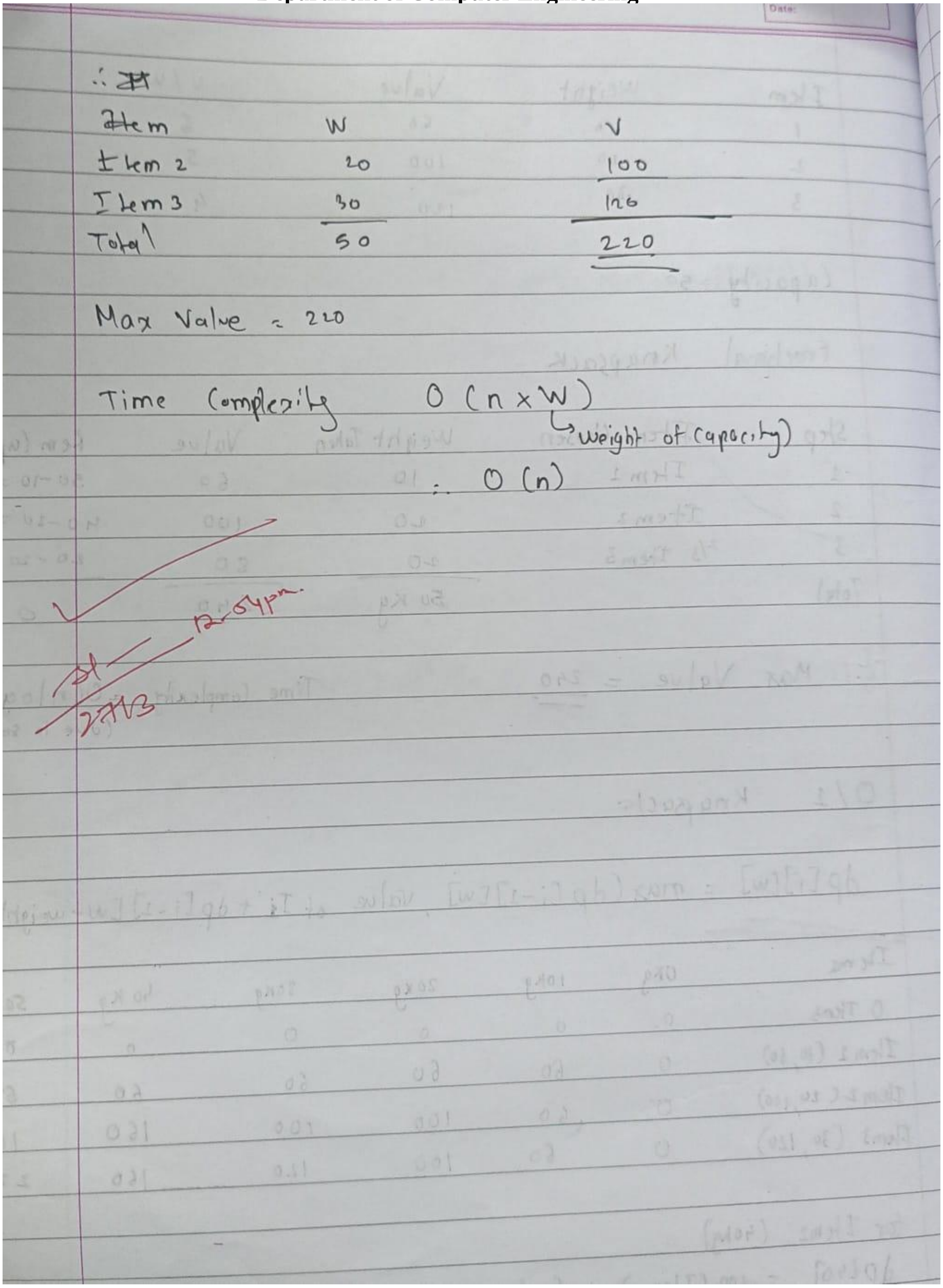
Time Complexity = O(n log n)
(due to sort)

0/1 Knapsack

dp[i][w] = max(dp[i−1][w], value of I_i + dp[i−1][w − weight of I_i]

Items	0Kg	10kg	20kg	30kg	40 Kg	50kg
0 Items	0	0	0	0	0	0
Item 1 (10, 60)	0	60	60	60	60	60
Item 2 (20, 100)	0	60	100	100	160	160
Item 3 (30, 120)	0	60	100	120	160	220

for Item2 (40kg)
dp[40] = 100 (Item2) + 60 (Remaining 20kg from Item 1) = 160
for Item 3 (50kg)
dp[50] = 120 (Item3) + 100 (Remaining 30kg from Item 2) = 220



∴ ~~It~~

Item	W	V
Item 2	20	100
Item 3	30	120
Total	50	220

Max Value = 220

Time Complexity $O(n \times \underbrace{W}_{\text{weight of capacity}})$

$\therefore O(n)$

✓

[illegible]

[illegible] 27/3

Analysis of algorithm:

<u>Fractional Knapsack</u>

Sorting: O(n logn)

Selection: O(n)

Thus, the overall time complexity is **O(n logn)**

<u>0/1 Knapsack</u>

Time Complexity: O(N * W) ~ O(N)

There will be 'N * W' calls in a 2D array using the nested 'for' loop. Therefore the overall time complexity is O(N * W).

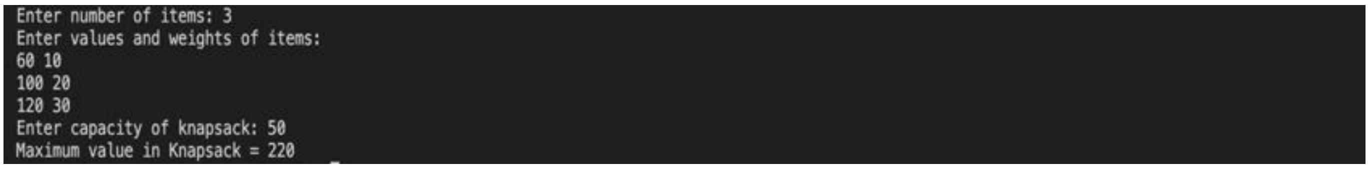
Space Complexity: O(W)

Code:

<u>0/1 Knapsack</u>

```
import java.util.Scanner;

class dp {
    public static int knapsack(int W, int wt[], int val[], int n) {
        int[][] dp = new int[n + 1][W + 1];

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

        return dp[n][W]; // Final answer in the last cell
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of items: ");
        int n = sc.nextInt();
        int val[] = new int[n];
        int wt[] = new int[n];

        System.out.println("Enter values and weights of items: ");
        for (int i = 0; i < n; i++) {
            val[i] = sc.nextInt();
            wt[i] = sc.nextInt();
        }

        System.out.print("Enter capacity of knapsack: ");
        int W = sc.nextInt();

        System.out.println("Maximum value in Knapsack = " + knapsack(W, wt, val,
n));

        sc.close();
    }
}
```

Output



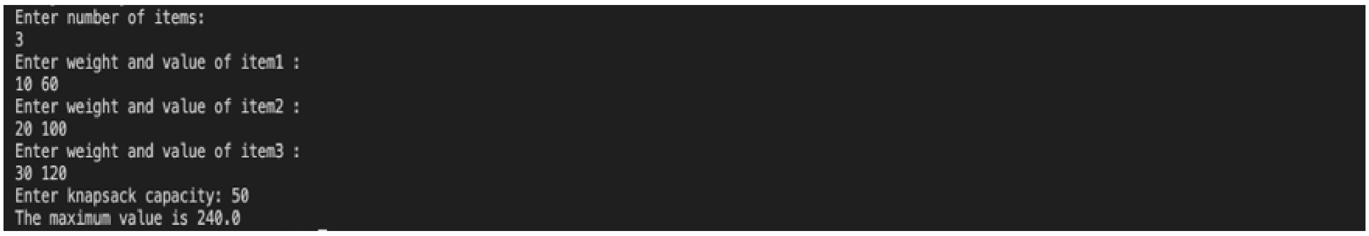
```
Enter number of items: 3
Enter values and weights of items:
60 10
100 20
120 30
Enter capacity of knapsack: 50
Maximum value in Knapsack = 220
```

Fractional Knapsack

```java
import java.util.*;

public class fract{

    public static double fractionalK(int w,Item arr[],int n){
        Arrays.sort(arr, new ItemComparator());

        int curr=0;
        double fin=0.0;

        for(int i=0;i<n;i++){
            if(curr+arr[i].weight<=w){
                curr+=arr[i].weight;
                fin+=arr[i].value;
            }
            else{
                int rem=w-curr;
                fin+=((double)arr[i].value/arr[i].weight)*rem;
                break;
            }
        }
        return fin;
    }

    public static void main(String[] args) {
        Scanner sc=new Scanner(System.in);

        System.out.println("Enter number of items: ");
        int n=sc.nextInt();
        Item arr[]=new Item[n];

        for(int i=0;i<n;i++){
            System.out.println("Enter weight and value of item" + (i+1) +" :");
            int w=sc.nextInt();
            int v=sc.nextInt();
            arr[i]=new Item(w,v);
        }

        System.out.print("Enter knapsack capacity: ");
        int weight = sc.nextInt();

        sc.close();

        double ans = fractionalK(weight, arr, n);
        System.out.println("The maximum value is " + ans);
    }
}
```

```
class Item{
    int weight;
    int value;

    Item(int weight,int value){
       this.weight=weight;
       this.value=value;
    }
}

 class ItemComparator implements Comparator<Item>{
    @Override

    public int compare(Item a,Item b){
        double r1=a.value/a.weight;
        double r2=b.value/b.weight;

        return Double.compare(r2, r1);
    }
}
```

Output



```
Enter number of items:
3
Enter weight and value of item1 :
10 60
Enter weight and value of item2 :
20 100
Enter weight and value of item3 :
30 120
Enter knapsack capacity: 50
The maximum value is 240.0
```

CONCLUSION:

The above experiment shows implementation of 0/1 Knapsack using Dynamic Programming, and draws comparison between the difference between 0/1 knapsack & fractional knapsack problem.