Experiment – 7

Student Name: Himanshu Gupta
UID: 23BCS10889
Branch: BE-CSE
Section/Group: KRG-2B
Semester: 5th
Date of Performance: 13/10/25
Subject Name: DAA
Subject Code: 23CSH-301

1. **Aim:** Develop a program and analyze complexity to implement 0-1 Knapsack using Dynamic Programming.

2. **Objective:** to implement 0-1 Knapsack using Dynamic Programming.

3. **Input/Apparatus Used:** In order to fill knapsack, we can pick only complete item not in fraction.

4. **Procedure:**

In **Dynamic Programming (DP)**, we consider the same cases as mentioned in the recursive approach.

We maintain a DP table `DP[i][j]`, where we consider all possible weights from `1` to `W`.

Steps:

1. Fill `wi` in the given column.
2. Do not fill `wi` in the given column.

Now, we take the **maximum** of these two possibilities:

- If we **do not fill** the `ith` weight in the `jth` column, then
  DP[i][j] = DP[i-1][j]
- If we **fill** the `ith` weight, then
  DP[i][j] = value of wi + DP[i-1][j - wi]

Hence, the recurrence relation is:

DP[i][j] = max(DP[i-1][j], value[i] + DP[i-1][j - weight[i]])

This visualization will make the concept clear.

Example:

Weights = {1, 2, 3}
Values = {10, 15, 40}
Capacity = 6

DP Table Construction:

i / Weight	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	?	?	?

Step-by-step Updates for weight = 3

For j = 4:
DP[3][4] = max(25, 40 + DP[2][4 - 3])
= max(25, 40 + 10)
= 50

For j = 5:
DP[3][5] = max(25, 40 + DP[2][5 - 3])
= max(25, 40 + 15)
= 55

For j = 6:
DP[3][6] = max(25, 40 + DP[2][6 - 3])
= max(25, 40 + 25)
= 65

Final DP Table:

i / Weight	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

0/1 Knapsack

Earlier we have discussed Fractional Knapsack problem using Greedy approach. We have

shown that Greedy approach gives an optimal solution for Fractional Knapsack. However, this chapter will cover 0-1 Knapsack problem and its analysis.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of ***xi*** can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight (***pi/wi***), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute maximum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

5. Code and Output:

6.

```
b31979\redhat.java\jdt_ws\DAA_23BCS10889_KRG-2B_4bf06f46\bin'
Enter number of items: 5
Enter weights: 10 20 30 10 50
Enter profits: 10 50 20 60 20
Enter maximum capacity of knapsack: 50
Maximum Profit = 120
Items picked (1-based index): 1 2 4
PS C:\Users\ASUS\Desktop\Sem 5\DAA_23BCS10889_KRG-2B>
```

7. **Time Complexity**: O(nW) where n is the number of items and W is the capacity of knapsack.