WORKSHEET 7

Student Name: Harsh Kumar **UID: 22BCS15754**
Branch: CSE **Section/Group: FL_IOT_603'B'**
Semester: 5th **Date of Performance:12/09/24**
Subject Name: Design and Analysis of Algorithms **Subject Code: 22CSH-311**

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

2. **Objectives:** To implement 0-1 Knapsack problem using Dynamic programming.

3. **Algorithm:**

- Create a table with n + 1 rows (one for each item and an extra row for 0 items) and W + 1 columns (one for each capacity from 0 to W).
- Initialize the first row and the first column of the table to 0. This represents the case where either you have no items or the knapsack has zero capacity — both result in 0 value.
- For each item i (from 1 to n) and each knapsack capacity w (from 1 to W):
- If the weight of the item i is less than or equal to the current capacity w, you have two choices:

1. **Include the item**: Take the value of the item and add it to the value of the knapsack that fits the remaining weight (w - weight of item i).
2. **Exclude the item**: Just use the value without including this item (i.e., the value of the previous item for the same capacity).

- Take the maximum of these two values and store it in the table.
- If the weight of the item is greater than the current capacity w, you cannot include the item, so you just take the value from the previous row (i.e., the value of excluding the item).

4. Implementation/Code:

```
#include <bits/stdc++.h>
using namespace std;

int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    vector<vector<int> > K(n + 1, vector<int>(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 profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);

    cout << knapSack(W, weight, profit, n);

    return 0;
}
```

5. Output:

```
220

...Program finished with exit code 0
Press ENTER to exit console.
```

6. Time Complexity:

O(N * W). where 'N' is the number of elements and 'W' is capacity.

7. Learning Outcome:

1) Learnt how to use Dynamic Programming concepts and how to apply them to solve problems.
2) Learnt The Knapsack Problem and how to put the items into the bag such that the sum of profits associated with them is the maximum possible.