Name	Surti Keyur
Roll No.	25MCE026
Branch	M.Tech (CSE)

Experiment 10

Aim: To implement the 0/1 Knapsack problem using Dynamic Programming and display the DP table.

Theory

The 0/1 Knapsack problem aims to maximize total profit by selecting items with given weights and profits without exceeding bag capacity. Each item can either be taken (1) or not taken (0), hence it is called 0/1 Knapsack. Dynamic Programming is used to optimize the solution by storing intermediate results in a DP table.

Algorithm

1. Input the number of items, their profits, weights, and knapsack capacity.

2. Create a DP table dp[n+1][W+1] initialized to 0.

3. For each item i from 1..n:

    For each weight w from 1..W:

        If weight[i] <= w:

            dp[i][w] = max(profit[i] + dp[i-1][w-weight[i]], dp[i-1][w])

        Else:

            dp[i][w] = dp[i-1][w]

4. Print the DP table.

5. The result (maximum profit) is dp[n][W].

Source Code

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

int main() {
```

```
int n, W;
cout << "Enter number of items: ";
cin >> n;

vector<int> profit(n+1), weight(n+1);

cout << "Enter profits of items:\n";
for(int i = 1; i <= n; i++) cin >> profit[i];

cout << "Enter weights of items:\n";
for(int i = 1; i <= n; i++) cin >> weight[i];

cout << "Enter Knapsack Capacity: ";
cin >> W;

// DP table
vector<vector<int>> dp(n+1, vector<int>(W+1, 0));

// Build DP table
for(int i = 1; i <= n; i++) {
    for(int w = 1; w <= W; w++) {
        if(weight[i] <= w) {
            dp[i][w] = max(profit[i] + dp[i-1][w - weight[i]],
                           dp[i-1][w]);
        } else {
            dp[i][w] = dp[i-1][w];
        }
    }
}

// Print DP Table
cout << "\n===== DP TABLE =====\n";

// Header row (weights 0..W)
cout << "     ";
for(int w = 0; w <= W; w++)
    cout << setw(4) << w;
cout << "\n";

cout << "-----------------------------------------------\n";

for(int i = 0; i <= n; i++) {
    cout << "i=" << i << " | ";
    for(int w = 0; w <= W; w++) {
        cout << setw(4) << dp[i][w];
    }
    cout << "\n";
}

cout << "\nMaximum Profit = " << dp[n][W] << "\n";
```

```
    return 0;
}
```

Output

```
PS E:\DSA\Experiments\EXP_10_01_KP\output> cd 'e:\DSA\Experiments\EXP_10_01_KP\output'
PS E:\DSA\Experiments\EXP_10_01_KP\output> & .\'01_KP.exe'
Enter number of items: 3
Enter profits of items:
60 100 120
Enter weights of items:
10 20 30
Enter Knapsack Capacity: 50

===== DP TABLE =====
        0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23
 24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  4
9  50
------------------------------------------------
i=0 |    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 0   0
i=1 |    0   0   0   0   0   0   0   0   0   0  60  60  60  60  60  60  60  60  60  60  60  60  60  60
  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60  60
60  60
i=2 |    0   0   0   0   0   0   0   0   0   0  60  60  60  60  60  60  60  60  60  60 100 100 100 100
 100 100 100 100 100 100 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 1
60 160
i=3 |    0   0   0   0   0   0   0   0   0   0  60  60  60  60  60  60  60  60  60  60 100 100 100 100
 100 100 100 100 100 100 160 160 160 160 160 160 160 160 160 180 180 180 180 180 180 180 180 180 1
80 220

Maximum Profit = 220
```

Time Complexity

The time complexity is O(n × W) because the algorithm fills a DP table of size n×W.

Space complexity is also O(n × W).

Conclusion

The 0/1 Knapsack problem can be efficiently solved using Dynamic Programming. By constructing and analyzing the DP table, we obtain the optimal solution and understand how each subproblem contributes to the final result.