Experiment-7

Student Name: Gourav Sharma　　**UID:** 23BCS10857
Branch: BE-CSE　　**Section/Group:** 23BCS_KRG_3A
Semester: 5^{th}　　**Subject Code:** 23CSH-301
Subject Name: DAA

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

2. **Objective:** To implement the 0-1 Knapsack Problem using Dynamic Programming in C++ and understand how DP efficiently solves optimization problems where items can either be fully included or excluded.

3. **Procedure:**
   1. Start
   2. Create a 2D array dp[n+1][W+1] of type integer.
   3. Initialize:
      - dp[i][0] = 0 for all i → knapsack capacity 0 → profit 0
      - dp[0][w] = 0 for all w → 0 items → profit 0
   4. For each item i = 1 to n:
      - For each weight w = 1 to W:
        - If weights[i-1] ≤ w:
          dp[i][w] = max(values[i-1] + dp[i-1][w - weights[i-1]], dp[i-1][w])
        - Else:
          dp[i][w] = dp[i-1][w]
   5. The final solution is stored in dp[n][W].
   6. To find selected items, backtrack from dp[n][W]:
      - If dp[i][w] != dp[i-1][w] → item i is included → subtract its weight from w and continue.

4. **Code:**

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int n, W;
    cout << "Enter number of items: ";
    cin >> n;
```

```
    vector<int> values(n), weights(n);
    cout << "Enter values: ";
    for(int i=0; i<n; i++) cin >> values[i];
    cout << "Enter weights: ";
    for(int i=0; i<n; i++) cin >> weights[i];
    cout << "Enter knapsack capacity: ";
    cin >> W;
    vector<vector<int>> dp(n+1, vector<int>(W+1, 0));
    for(int i=1; i<=n; i++){
        for(int w=1; w<=W; w++){
            if(weights[i-1] <= w)
                dp[i][w] = max(values[i-1] + dp[i-1][w - weights[i-1]], dp[i-1][w]);
            else
                dp[i][w] = dp[i-1][w];
        }
    }
    cout << "Maximum value in Knapsack = " << dp[n][W] << endl;
    cout << "Items selected: ";
    int w = W;
    for(int i = n; i > 0 && w > 0; i--){
        if(dp[i][w] != dp[i-1][w]){
            cout << i << " ";
            w -= weights[i-1];
        }
    }
    return 0;
}
```

5. Observations:

```
Enter number of items: 4
Enter values: 60 100 120 80
Enter weights: 10 20 30 40
Enter knapsack capacity: 50
Maximum value in Knapsack = 220
Items selected: 3 2
```

6. Time Complexity:

- Filling the DP table takes $O(n \times W)$ → n items × W capacity.
- Backtracking items → $O(n)$

7. Learning Outcome:

- ❖ Learned how to implement the 0-1 Knapsack problem using Dynamic Programming in C++.
- ❖ Understood the concept of optimal substructure and overlapping subproblems in DP.
- ❖ Gained practical experience in constructing and filling a DP table to compute maximum profit.
- ❖ Learned how to backtrack through the DP table to determine which items are included in the optimal solution.
- ❖ Strengthened understanding of time and space complexity analysis for DP-based algorithms.