EXPERIMENT – 7

Student Name: Gaganjot Singh **UID: 22BCS14843**
Branch: BE-CSE **Section/Group:** 22BCS_JT_802-B
Semester: 5^{th} **Date of Performance:** 2024
Subject Name: Design & Analysis of Algorithms **Subject Code:** 22CSH-311

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

2. **Objective:** Implementing the 0-1 Knapsack problem using Dynamic Programming focus on optimizing the knapsack capacity, selecting items efficiently, and utilizing dynamic programming to solve the problem.

3. **Algorithm**
   - Initialize a 2D table dp of size (n+1) x (capacity+1) with all elements set to 0, where n is the number of items and capacity is the maximum capacity of the knapsack.
   - Iterate through each item i from 1 to n:
   - Iterate through each possible capacity w from 1 to capacity:
   - If the weight of the current item i is less than or equal to the current capacity w, calculate the maximum value by considering two options:
   - Take the current item: values[i-1] + dp[i-1][w-weights[i-1]]
   - Do not take the current item: dp[i-1][w]
   - Choose the maximum value and store it in dp[i][w].
   - If the weight of the current item i is more than the current capacity w, do not take the current item and store the value from the previous row: dp[i-1][w].
   - Return the maximum value that can be put in the knapsack, which is stored in dp[n][capacity].

4. **Implementation/Code:**

```
#include <iostream>
#include <vector>
using namespace std;

int knapsack(vector<int> weights, vector<int> values, int capacity)
{  int n = values.size();
   vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));

   for (int i = 0; i <= n; i++)
   {  for (int w = 0; w <= capacity; w++)
      {    if (i == 0 || w == 0)
```

```
{ dp[i][w] = 0;}
            else 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 << "Knapsack Table:" << endl;
    cout << "  | ";
    for (int w = 0; w <= capacity; w++)
    {cout << w << " ";}
    cout << endl;
    for (int i = 0; i <= n; i++)
    {       if (i == 0)
        {           cout << "  | ";}
        else
        {           cout << "Item " << i << " | ";}
     for (int w = 0; w <= capacity; w++)
        {
            cout << dp[i][w] << " ";}
        cout << endl;}
return dp[n][capacity];}

int main()
{   int n;
    cout << "Enter the number of items: ";
    cin >> n;

    vector<int> weights(n);
    vector<int> values(n);

    cout << "Enter the weights of the items: ";
    for (int i = 0; i < n; i++)
    {cin >> weights[i];}

    cout << "Enter the values of the items: ";
    for (int i = 0; i < n; i++)
    {cin >> values[i];}

    int capacity;
    cout << "Enter the capacity of the knapsack: ";
    cin >> capacity;

    int max_value = knapsack(weights, values, capacity);
    cout << "Maximum value that can be put in a knapsack of capacity " << capacity << " is " <<
max_value << endl;    return 0;}
```

5. Output

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    SEARCH ERROR

PS E:\CU Study\22CSH 311 DAA\codes> cd "e:\CU Study\22CSH 311 DAA\codes\" ;
Enter the number of items: 2
Enter the weights of the items: 3
4
Enter the values of the items: 5
7
Enter the capacity of the knapsack: 5
Knapsack Table:
  | 0 1 2 3 4 5
  | 0 0 0 0 0 0
Item 1 | 0 0 0 5 5 5
Item 2 | 0 0 0 5 7 7
Maximum value that can be put in a knapsack of capacity 5 is 7
PS E:\CU Study\22CSH 311 DAA\codes>
```

6. Time Complexity :

Since the outer loop and inner loop are nested, the total number of iterations is (n+1) x (W+1), which simplifies to O(nW). The time complexity is linear with respect to both the number of items and the maximum capacity of the knapsack.

7. Learning Outcomes:

- Understanding of Dynamic Programming, which involves breaking down a complex problem into smaller subproblems and solving each subproblem only once.
- Using a 2D table dp to store the solutions to subproblems, which is a fundamental data structure in dynamic programming.
- The code illustrates a systematic approach to solving a complex problem, such as the 0/1 knapsack problem, by breaking it down into smaller subproblems and solving them iteratively.
- The problem is related to the field of operations research and optimization, providing an opportunity to learn about the 0/1 knapsack problem and its applications in real-world scenarios.
- Understand how to declare, initialize, and manipulate vectors, as well as how to use vector operations such as push_back and size.
- Used the vector data structure to store the weights and values of items, demonstrating the use of dynamic memory allocation and the benefits of using vectors over traditional arrays.