Experiment - 7

Student Name: Mankaran Singh Tandon **UID:** 23BCS10204
Branch: BE-CSE **Section/Group:** KRG-2B
Semester: 5th **Date of Performance:** 13/10/25
Subject Name: Design and Analysis of Algorithms
Subject Code: 23CSH-301

Aim: Develop a program and analyze complexity to implement 0-1 Knapsack using Dynamic Programming.
Objective: to implement 0-1 Knapsack using Dynamic Programming.
Input/Apparatus Used: In order to fill knapsack, we can pick only complete item not in fraction.

Procedure:

In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a DP[][] table let‟s consider all the possible weights from „1‟ to „W‟ as the

- Fill „wi‟ in the given column.
- Do not fill „wi‟ in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill „ith‟weight in „jth‟ column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of „wi‟+ value of the column weighing „j-wi‟ in the previous row. So we take the maximum of these two possibilities to fill the current state. This visualisation will make the concept clear:
Let weight elements = {1, 2, 3}
Let weight values = {10, 15, 40} Capacity=6
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

For filling 'weight=3', we
come across 'j=4' in which
we take maximum of (25, 40 + DP[2][4-3])
= 50

For filling 'weight=3' we
come across 'j=5' in which
we take maximum of (25, 40 + DP[2][5-3])
= 55

For filling 'weight=3' we
come across 'j=6' in which
we take maximum of (25, 40 + DP[2][6-3])
= 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.

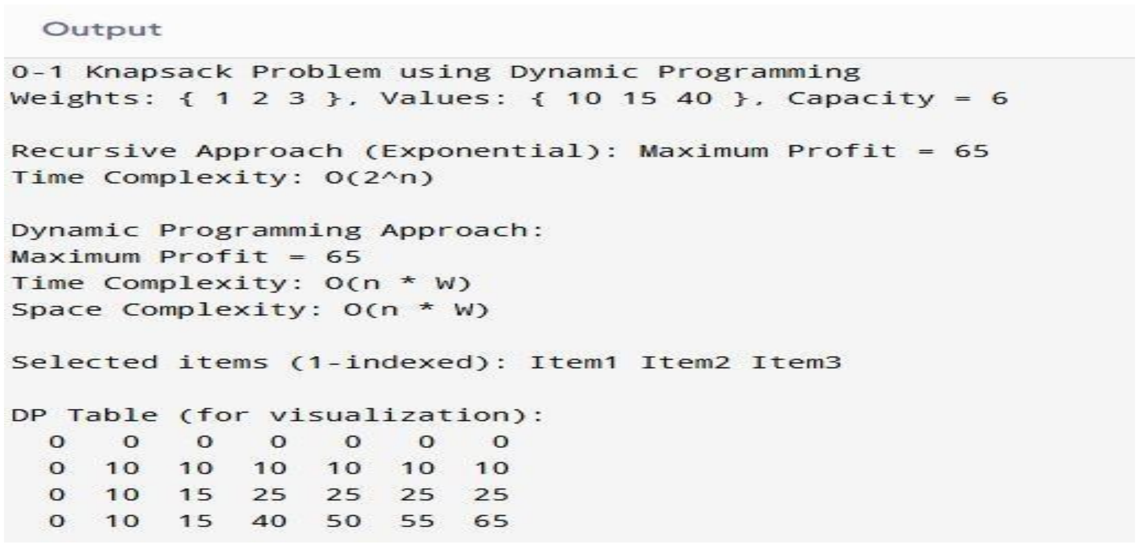
Code and Output:

```
#include <bits/stdc++.h>

using namespace std;



// Recursive approach int knapsackRecursive(int W, vector<int>& wt,

vector<int>& val, int n) { if (n == 0 || W == 0)

        return 0;

    if (wt[n - 1] > W) return

        knapsackRecursive(W, wt, val, n - 1);

    else

        return max(val[n - 1] + knapsackRecursive(W - wt[n - 1], wt, val, n - 1),

                    knapsackRecursive(W, wt, val, n - 1));

}



// Dynamic Programming approach (Bottom-Up) int knapsackDP(int W, vector<int>&

wt, vector<int>& val, vector<vector<int>>& dp) { int n = wt.size();
```

```
    for (int i = 0; i <= n; i++) { for (int w = 0; w <= W; w++) { if (i == 0 ||
        w == 0) dp[i][w] = 0; else if (wt[i - 1] <= w) dp[i][w] = max(val[i -
        1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
            else dp[i][w] = dp[i -
                1][w];
        } } return dp[n]
    [W];
}


// Track selected items
vector<int> trackItems(vector<int>& wt, vector<int>& val, vector<vector<int>>& dp,
int W) {
    int n = wt.size();
    int w = W;
    vector<int> selectedItems;


    for (int i = n; i > 0 && w > 0; i--)
        { if (dp[i][w] != dp[i - 1][w]) {
            selectedItems.push_back(i);
            w -= wt[i - 1];
        } } reverse(selectedItems.begin(),

    selectedItems.end()); return selectedItems;

}


int main() { vector<int> wt = {1,

    2, 3}; vector<int> val = {10,

    15, 40}; int W = 6;

    int n = wt.size();


    cout << "0-1 Knapsack Problem using Dynamic

    Programming\n"; cout << "Weights: { "; for (int x : wt) cout << x

    << " "; cout << "}, Values: { "; for (int x : val) cout << x << " ";

    cout << "}, Capacity = " << W << "\n\n";

    // Recursive Method int recResult =

    knapsackRecursive(W, wt, val, n);

    cout << "Recursive Approach (Exponential): Maximum Profit = " << recResult <<
"\n";

    cout << "Time Complexity: O(2^n)\n\n";
```

```
// Dynamic Programming Method

vector<vector<int>> dp(n + 1, vector<int>(W + 1,

0)); int dpResult = knapsackDP(W, wt, val, dp);


cout << "Dynamic Programming Approach:\n";

cout << "Maximum Profit = " << dpResult <<

"\n"; cout << "Time Complexity: O(n * W)\n";

cout << "Space Complexity: O(n * W)\n\n";


// Tracking chosen items vector<int> selected =

trackItems(wt, val, dp, W); cout << "Selected

items (1-indexed): "; for (int idx : selected) cout

<< "Item" << idx << " ";

cout << "\n\n";


cout << "DP Table (for visualization):\n";

for (int i = 0; i <= n; i++) { for (int j = 0;

j <= W; j++) cout << setw(3) << dp[i][j]

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

}
```

```
    return 0;

}
```



```
Output
0-1 Knapsack Problem using Dynamic Programming
Weights: { 1 2 3 }, Values: { 10 15 40 }, Capacity = 6

Recursive Approach (Exponential): Maximum Profit = 65
Time Complexity: O(2^n)

Dynamic Programming Approach:
Maximum Profit = 65
Time Complexity: O(n * W)
Space Complexity: O(n * W)

Selected items (1-indexed): Item1 Item2 Item3

DP Table (for visualization):
   0    0    0    0    0    0    0
   0   10   10   10   10   10   10
   0   10   15   25   25   25   25
   0   10   15   40   50   55   65
```

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