|  |  |
| --- | --- |
|  | Department of Computer Science and Engineering  Chandpur Science and Technology University |

**LAB-08**

**Course Title**: Algorithm Design and Analysis Sessional

**Course Code**:CSE 2202

**Submitted To-**

**Mostafiz Ahammed**

*Lecturer****Department of Computer Science and Engineering****Chandpur Science and Technology University*

**Submitted By-**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl** | Class Roll | Name | Batch | Signature |
| 01 | B210101022 | Md. Rabbi Ahammed | CSE-21 | rabbi |

**Date of Submission: 19 July, 2025**

# Experiment: Implementation of 0/1 Knapsack Problem using Dynamic Programming

## Objective

To implement the 0/1 Knapsack Problem using Dynamic Programming in C++, and analyze the time and space complexities of the solution.

## Algorithm

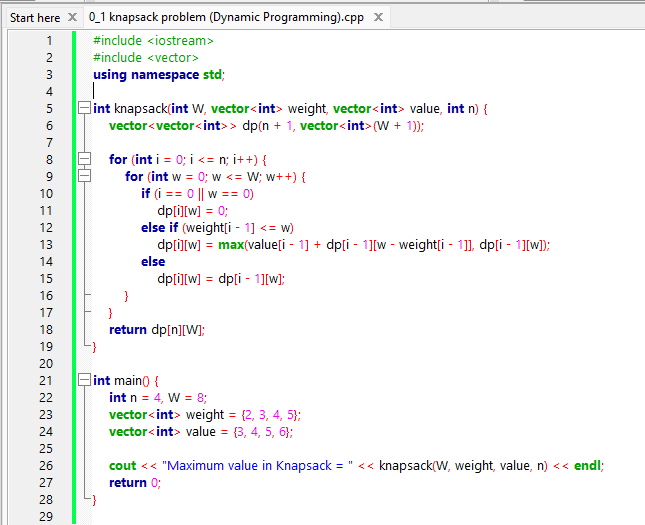
The 0/1 Knapsack Problem involves selecting items with given weights and values to maximize total value without exceeding the knapsack's capacity.   
Each item can either be included or excluded (0/1).  
  
Steps:  
1. Create a 2D array dp[n+1][W+1], where n is the number of items and W is the capacity.  
2. Initialize the first row and column as 0.  
3. For each item i and capacity j:  
 - If weight[i-1] <= j:  
 dp[i][j] = max(dp[i-1][j], value[i-1] + dp[i-1][j - weight[i-1]])  
 - Else:  
 dp[i][j] = dp[i-1][j]  
4. The result is found at dp[n][W].

## Theoretical Solution

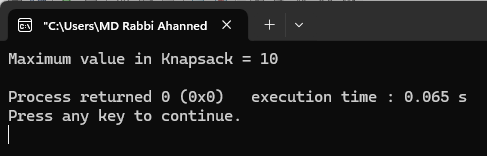
The 0/1 Knapsack Problem is solved using dynamic programming due to:  
- Optimal Substructure: The optimal solution to the problem can be constructed from optimal solutions of its subproblems.  
- Overlapping Subproblems: The same subproblems are solved multiple times.  
  
Dynamic programming stores and reuses these results, greatly improving efficiency over naive recursion.

## Practical Work

**Pseudocode:**  
  
function knapsack(weights[], values[], n, W):  
 create 2D array dp[n+1][W+1]  
 for i from 0 to n:  
 for w from 0 to W:  
 if i == 0 or 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]  
 return dp[n][W]

**Source Code in C++ :**  
  


**Output:**

****

## Analysis Table

| **Metric** | **Value** |
| --- | --- |
| Time Complexity | O(n × W) |
| Space Complexity | O(n × W) |
| Optimal Substructure | Yes |
| Overlapping Subproblems | Yes |

## Observations

- DP approach avoids redundant computations.  
- Solution matrix gives visibility into each decision.  
- Much faster than recursive implementation for large n and W.

## Challenges

- Managing the size of the DP table for large input.  
- Handling off-by-one indexing errors.  
- Choosing correct data types and structures in C++.

## Conclusion

This lab demonstrates how dynamic programming significantly improves performance for the 0/1 Knapsack Problem. By storing and reusing subproblem results, the solution becomes scalable and efficient for larger datasets.