Final Report

Problem 2: Cargo Loading Optimization

1. Problem Description

The **Knapsack Problem** is a classic optimization problem. The objective is to determine the most valuable subset of items that can be placed into a knapsack without exceeding a given weight limit. In this scenario, each item has a weight and a value associated with it, and the knapsack has a maximum weight capacity.

Problem Statement:

We are tasked with selecting a subset of items such that:

- The **total weight** of the selected items is **less than or equal** to the knapsack's weight limit.
- The **total value** of the selected items is **maximized**.

- Illustrative Input-Output Examples

Example 1:

- **Weights** = [2, 3, 4, 5]
- **Values** = [3, 4, 5, 6]
- **Max Weight** = 5

Output:

- **Maximum Value** = 7
- **Best Combination** = (0, 1) [Select items 1 and 2]

Example 2:

- **Weights** = [1, 2, 3, 8, 7, 4]
- **Values** = [20, 5, 10, 40, 30, 50]
- **Max Weight** = 10

Output:

- **Maximum Value** = 60
- **Best Combination** = (0, 1, 5) [Select items 1, 2, and 6]

Example 3:

- **Weights** = [1, 2, 3]
- **Values** = [10, 20, 30]
- **Max Weight** = 4

Output:

- **Maximum Value** = 30
- **Best Combination** = (1, 2) [Select items 2 and 3]

2. Pseudocode: Brute Force Knapsack Algorithm

```
Algorithm knapsackBruteforce(weights, values, max_weight)
Input:
    - Array weights[0...n-1], representing weights of items
    - Array values[0...n-1], representing values of items
    - Integer max_weight, the maximum weight the knapsack can carry
Output:
    - Integer max_value, the maximum achievable value
    - Array best_combination, the indices of items that form the best combination

Begin
    n ← length of weights
    currentMax ← 0
    best_combination ← []

    for i ← 1 to n do
        for combination ← allCombinations(range(0, n-1), i) do
            total_weight ← 0
            total_value ← 0

            for each item in combination do
                total_weight ← total_weight + weights[item]
                total_value ← total_value + values[item]

            if total_weight ≤ max_weight and total_value > currentMax then
                currentMax ← total_value
                best_combination ← combination

    return currentMax, best_combination
End
```

3. Complexity Analysis

Time Complexity:

- *O(2^n*n):*
  - *2^n* represents the number of combinations of items (subsets of the available items).
  - **n** represents the time required to calculate the total weight and total value for each combination.

Space Complexity:

- *O(n)*:
  - The space is used to store the current combination and its weight and value.

4. Empirical Results

Experimental Setup:

To evaluate the performance of the brute force algorithm, we tested it on multiple datasets with varying numbers of items. The goal was to observe how the execution time increases as the number of items (n) grows. For each dataset, we measured the time taken by the algorithm to find the optimal solution and the best combination of items selected under the given constraints.

Test Case 1 (n = 4):

- **Weights** = [2, 3, 4, 5]
- **Values** = [3, 4, 5, 6]
- **Max Weight** = 5

Output:

- **Maximum Value** = 7
- **Best Combination** = (0, 1)

Execution Time:

- **For n=4**, the algorithm took **0.000124 seconds** to compute the solution.

Test Case 2 (n = 5):

- **Weights** = [2, 3, 4, 5, 6]
- **Values** = [3, 4, 5, 6, 7]
- **Max Weight** = 10

Output:

- **Maximum Value** = 13
- **Best Combination** = (0, 1, 3)

Execution Time:

- **For n=5**, the algorithm took **0.0001816 seconds** to compute the solution.

Test Case 3 (n = 20):

- **Weights** = [i for i in range(1 to 21)]
- **Values** = [i +2 for i in range(1 to 21)]
- **Max Weight** = 50

Output:

- **Maximum Value** = 68
- **Best Combination** = (0, 1, 2, 3, 4, 5, 6, 7, 13)

Execution Time:

- **For n=10**, the algorithm took **2.5 seconds** to compute the solution.

Test Case 4 (n = 24):

- **Weights** = [i for i in range(1 to 25)]

- **Values** =[i + 2 for i in range(1 to 25)]
- **Max Weight** = 50

Output:

- **Maximum Value** = 68
- **Best Combination** = (0, 1, 2, 3, 4, 5, 6, 7, 13)

Execution Time:

- **For n=10**, the algorithm took 43.233 **seconds** to compute the solution.

Analysis:

- **Time Complexity**: The execution time grows exponentially with the increase in the number of items n. Since the algorithm considers all possible subsets of items, the time complexity is $O(2^n)$.

- **Empirical Observations**:
  - For smaller values of n, the execution time is negligible (fractions of a second).
  - As the number of items increases, the execution time grows significantly. For n=24, the execution time is 43.233 seconds, which indicates the inefficiency of the brute-force approach for larger inputs.

- **Conclusion**: The brute-force algorithm is effective for small problem sizes but becomes impractical as the problem size grows. The results confirm that more efficient approaches (such as dynamic programming) should be considered for larger datasets.