	Pimpri Chinchwad Education Trust's **Pimpri Chinchwad College of Engineering**
Assignment No: 6	

1. Problem Statement:

A massive earthquake has struck a remote region, and a relief organization is transporting essential supplies to the affected area. The organization has a limited-capacity relief truck that can carry a maximum weight of W kg. They have N different types of essential items, each with a specific weight and an associated utility value (importance in saving lives and meeting urgent needs). Since the truck has limited capacity, you must decide which items to include to maximize the total utility value while ensuring the total weight does not exceed the truck's limit.

Your Task as a Logistics Coordinator:

1. Model this problem using the 0/1 Knapsack approach, where each item can either be included in the truck (1) or not (0).

2. Implement an algorithm to find the optimal set of items that maximizes utility while staying within the weight constraint.

3. Analyze the performance of different approaches (e.g., Brute Force, Dynamic Programming, and Greedy Algorithms) for solving this problem efficiently.

4. Optimize for real-world constraints, such as perishable items (medicines, food) having priority over less critical supplies. Extend the model to consider multiple trucks or real-time decision-making for dynamic supply chain management.

2.Course Objective:

- To know the basics of computational complexity of various algorithms.
- To select appropriate algorithm design strategies to solve real-world problems..

3. Course Outcome:

- Generate optimal solutions by applying Dynamic Programming strategy.

4. Theory:

Problem Overview:

The **0/1 Knapsack Problem** is a classic optimization problem where we must select a subset of items to maximize the total value without exceeding a given weight limit.
Each item *i* has:

- weight[i]
- value[i]
- capacity = W

The objective is:

Maximize Σ(value[i] * x[i])

subject to Σ(weight[i] * x[i]) ≤ W

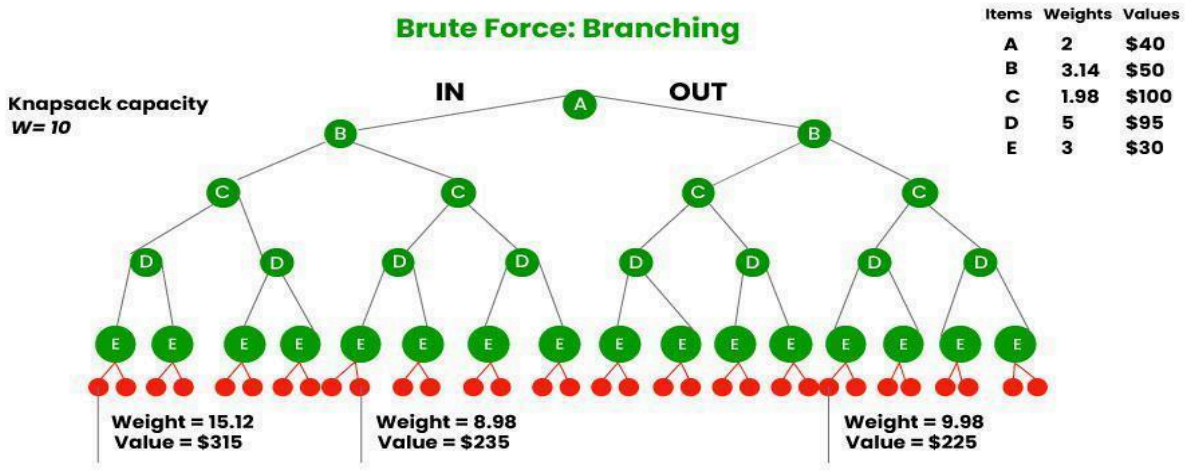
where x[i] ∈ {0, 1}

Algorithmic Approaches:

1. Brute Force Approach:

- Generate all possible subsets of items.
- For each subset, calculate total weight and value.
- Select the subset with the highest value ≤ W.

  Time Complexity: $O(2^n)$



2. Dynamic Programming Approach:

- Uses overlapping subproblems and optimal substructure.

- Create a DP table where dp[i][w] = maximum value using first *i* items with capacity *w*.
- Recurrence relation:

  if weight[i-1] ≤ w:

    dp[i][w] = max(value[i-1] + dp[i-1][w - weight[i-1]], dp[i-1][w])

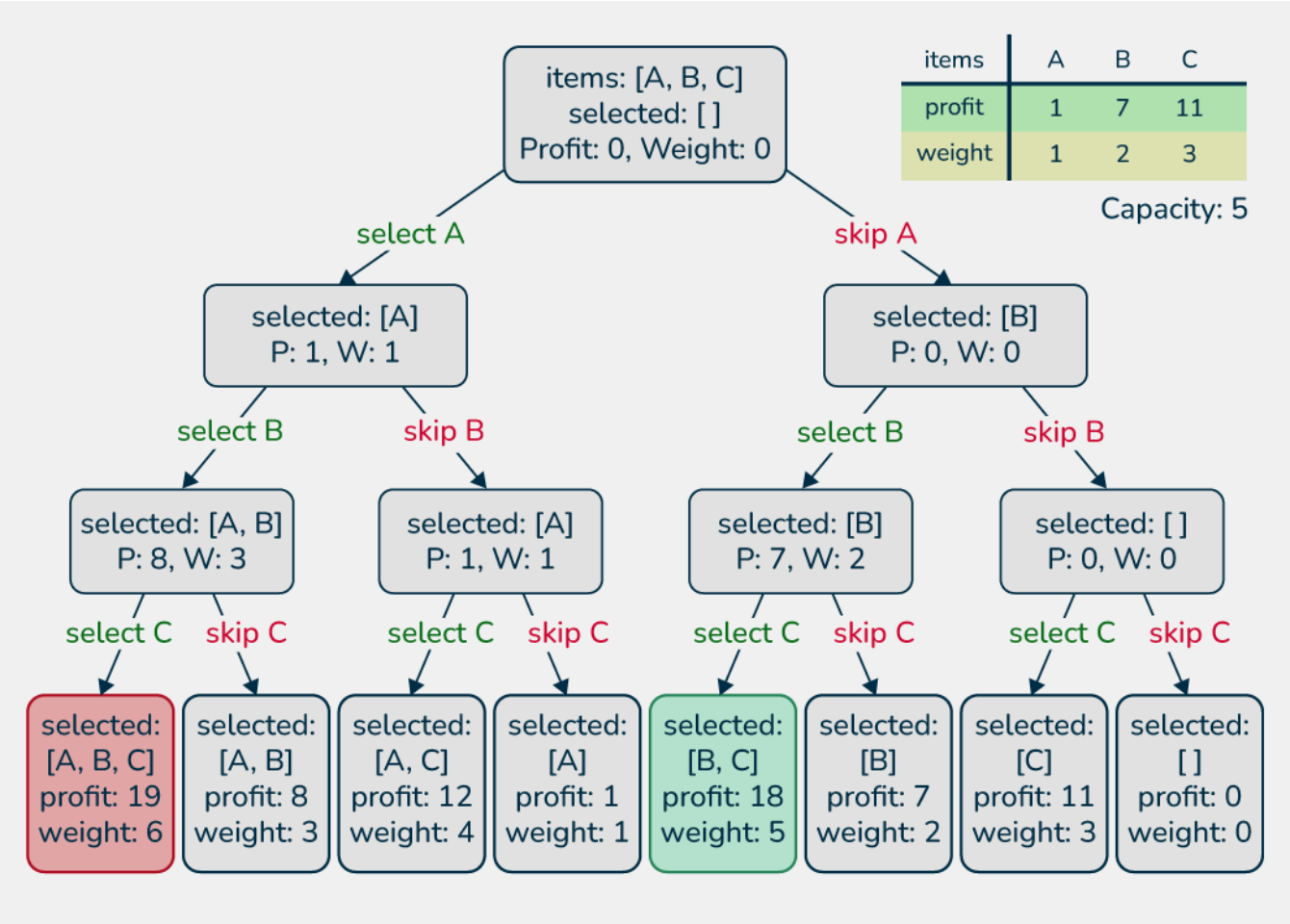
  else:

    dp[i][w] = dp[i-1][w]

  Time Complexity: O(N * W)
  Space Complexity: O(N * W)

3. Greedy Approach (for Fractional Knapsack):

- Sort items by value/weight ratio.
- Pick highest ratio items first until capacity fills.



5. Implementation:

Algorithm: 0/1 Knapsack for Disaster Relief

Input:

- N items, each with weight w[i] and utility value v[i]
- Truck capacity W

Output:

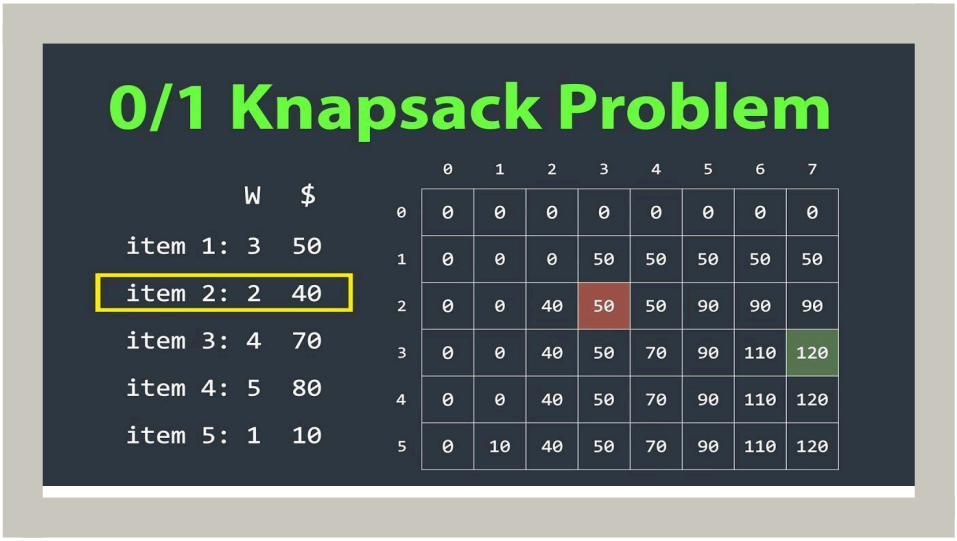
- Maximum total utility that can be carried
- List of items included in the truck

Steps:

1. **Initialize DP table** dp[n+1][W+1] to 0.

2. **Fill the DP table:**
   - For each item i from 1 to N:
     - For each weight w from 1 to W:
       - If weight[i-1] ≤ w:
       - dp[i][w] = max(value[i-1] + dp[i-1][w - weight[i-1]], dp[i-1][w])
       - Else:
       - dp [i][w] = dp[i-1][w]
3. **Maximum utility** is dp[N][W].
4. **Trace back** to find items included:
   - Start from dp[N][W].
   - For i = N down to 1:
     - If dp[i][w] ≠ dp[i-1][w], include item i and update w = w - weight[i-1

5.**Assign priority** to perishable or life-saving items by increasing their utility if needed.



6.Pseudocode: 0/1 Knapsack

function Knapsack(weight[], value[], N, W):

```
    // Step 1: Initialize DP table
    create dp[N+1][W+1] and initialize all to 0

    // Step 2: Fill DP table
    for i from 1 to N:
        for w from 1 to W:
            if weight[i-1] <= w:
                dp[i][w] = max(value[i-1] + dp[i-1][w - weight[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]

    // Step 3: Maximum utility
    maxUtility = dp[N][W]

    // Step 4: Traceback to find included items
    w = W
    includedItems = []
    for i from N down to 1:
        if dp[i][w] != dp[i-1][w]:
            includedItems.append(i)
            w = w - weight[i-1]

    return maxUtility, includedItems
```

CODE:

```
import java.util.*;

public class KnapsackVisualization {

    public static class Result {
        int maxValue;
        List<Integer> chosenItems;
```

```
    Result(int maxValue, List<Integer> chosenItems) {
        this.maxValue = maxValue;
        this.chosenItems = chosenItems;
    }
}

public static Result knapsack(int[] weights, int[] values, int W, int[] priorities) {
    int n = weights.length;
    int[][] dp = new int[n + 1][W + 1];

    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= W; w++) {
            int effectiveValue = values[i - 1];
            if (priorities != null && priorities.length > 0) {
                effectiveValue += priorities[i - 1] * 1000; // boost perishable
            }

            if (weights[i - 1] <= w) {
                dp[i][w] = Math.max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + effectiveValue);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }

        // ---- VISUALIZATION ----
        System.out.println("\nDP Table after item " + i
                + " (weight=" + weights[i - 1]
                + ", value=" + values[i - 1]
                + ", perishable=" + priorities[i - 1] + "):");
        for (int r = 0; r <= i; r++) {
            for (int c = 0; c <= W; c++) {
```

```
            System.out.printf("%5d ", dp[r][c]);
        }
        System.out.println();
    }
    System.out.println("-------------------------------------------------");
}

// Reconstruct chosen items
List<Integer> chosen = new ArrayList<>();
int w = W;
for (int i = n; i > 0; i--) {
    if (dp[i][w] != dp[i - 1][w]) {
        chosen.add(i - 1);
        w -= weights[i - 1];
    }
}
Collections.reverse(chosen);
return new Result(dp[n][W], chosen);
}

public static void main(String[] args) {
    int[] weights = {10, 20, 30, 40, 15};
    int[] values  = {60, 100, 120, 150, 80};
    int[] priorities = {1, 0, 0, 0, 1}; // 1 = perishable
    int W = 50;

    Result result = knapsack(weights, values, W, priorities);

    System.out.println("\nMax Utility Achieved: " + result.maxValue);
    System.out.println("Items Selected:");
    for (int idx : result.chosenItems) {
        System.out.println("  Item " + (idx + 1)
```

```
                + ": Weight=" + weights[idx]
                + ", Value=" + values[idx]
                + ", Perishable=" + (priorities[idx] == 1 ? "Yes" : "No"));
        }
    }
}
```

OUTPUT:

```
DP Table after item 1 (weight=10, value=60, perishable=1):
   1001    1001    1001    1001    1001 ... (rest up to W=50)
--------------------------------------------------

DP Table after item 2 (weight=20, value=100, perishable=0):
   1001  1001  1001  1001 ...
--------------------------------------------------

... (tables continue for all 5 items) ...

Max Utility Achieved: 2121
Items Selected:
  Item 1: Weight=10, Value=60, Perishable=Yes
  Item 5: Weight=15, Value=80, Perishable=Yes
```

7.Conclusion:

The 0/1 Knapsack model efficiently allocates limited truck capacity to maximize relief utility. Using Dynamic Programming, we ensure an optimal selection of critical items like food and medicines while respecting weight constraints.The approach is scalable, adaptable for multiple trucks and real-time updates, helping in smarter disaster management and life-saving decision-making.