

	<p style="text-align: center;">Pimpri Chinchwad Education Trust's Pimpri Chinchwad College of Engineering</p>
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:

- $\text{weight}[i]$
- $\text{value}[i]$
- capacity = W

The objective is:

$$\text{Maximize } \sum(\text{value}[i] * x[i])$$

$$\text{subject to } \sum(\text{weight}[i] * x[i]) \leq W$$

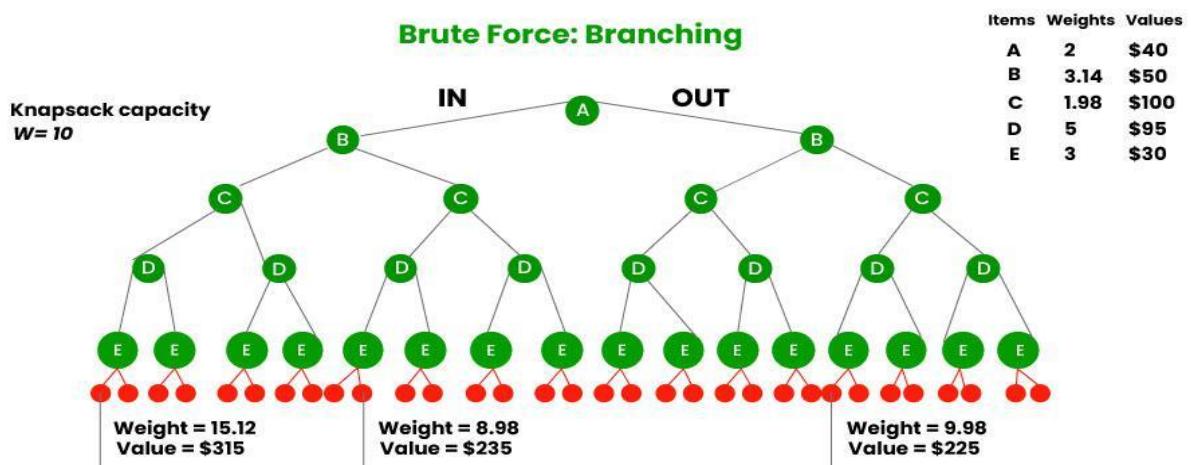
$$\text{where } x[i] \in \{0, 1\}$$

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 $\leq W$.

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 $weight[i-1] \leq w$:

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

else:

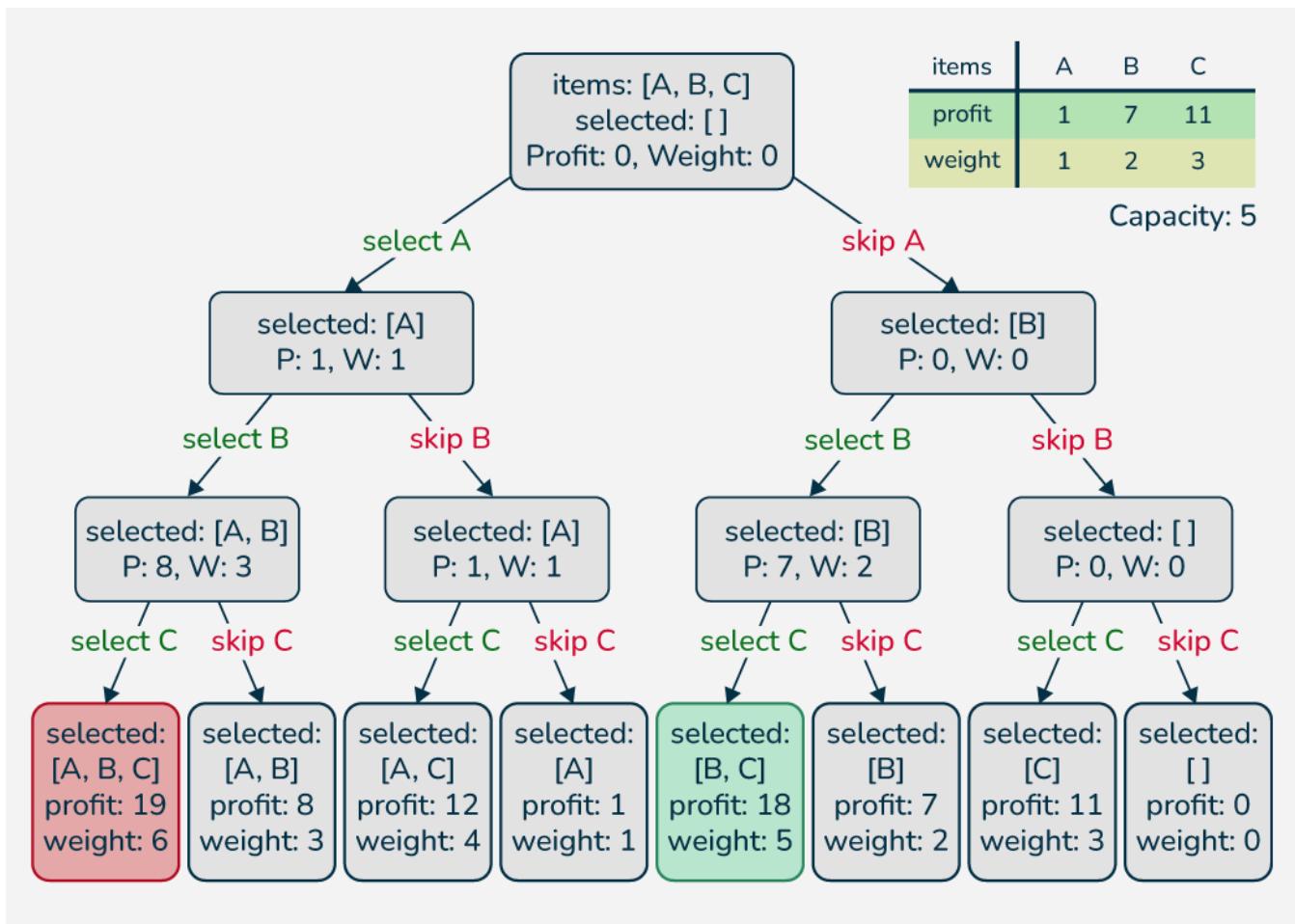
$$dp[i][w] = dp[i-1][w]$$

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 $weight[i-1] \leq w$:
 - $dp[i][w] = \max(value[i-1] + dp[i-1][w - weight[i-1]], dp[i-1][w])$
 - 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 $dp[i][w] \neq dp[i-1][w]$, 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.

0/1 Knapsack Problem								
W	\$	0	1	2	3	4	5	6
item 1:	3 50	0	0	0	0	0	0	0
item 2:	2 40	0	0	0	50	50	50	50
item 3:	4 70	0	0	40	50	70	90	110
item 4:	5 80	0	0	40	50	70	90	110
item 5:	1 10	0	10	40	50	70	90	110

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:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;
```

```

struct Item {
    string name;
    int value;
    int weight;
};

void solveKnapsack(int W, const vector<Item>& items) {
    int n = items.size();
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

    for (int i = 1; i <= n; ++i) {
        int currentWeight = items[i - 1].weight;
        int currentValue = items[i - 1].value;
        for (int w = 1; w <= W; ++w) {
            if (currentWeight <= w) {
                dp[i][w] = max(dp[i - 1][w], currentValue + dp[i - 1][w - currentWeight]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    int maxValue = dp[n][W];
    cout << "\nOptimal Solution:" << endl;
    cout << "-----" << endl;
    cout << "Maximum Utility Value: " << maxValue << endl;
    cout << "Items to include in the truck:" << endl;

    int w = W;
    for (int i = n; i > 0 && maxValue > 0; --i) {
        if (maxValue != dp[i - 1][w]) {
            cout << "- " << items[i - 1].name << " (Value: " << items[i - 1].value << ", "
Weight: " << items[i - 1].weight << ")" << endl;
            maxValue -= items[i - 1].value;
            w -= items[i - 1].weight;
        }
    }
    cout << "-----" << endl;
}

int main() {
    int truckCapacity;
    cout << "Enter the truck's maximum weight capacity: ";
    cin >> truckCapacity;

    int n;
    cout << "Enter the number of available item types: ";
    cin >> n;

    vector<Item> availableItems(n);
    cout << "Enter the details for each item (name value weight):" << endl;
    for (int i = 0; i < n; ++i) {
        cin >> availableItems[i].name >> availableItems[i].value >>
availableItems[i].weight;
    }
}

```

```
solveKnapsack(truckCapacity, availableItems);

    return 0;
}
```

Output:

```
Enter the truck's maximum weight capacity: 20
Enter the number of available item types: 5
Enter the details for each item (name value weight):
Medical_Kits 10 5
Water_Purifiers 8 8
Food_Rations 12 10
Tents 7 7
Blankets 5 4
```

```
Optimal Solution:
```

```
-----
Maximum Utility Value: 27
Items to include in the truck:
- Blankets (Value: 5, Weight: 4)
- Food_Rations (Value: 12, Weight: 10)
- Medical_Kits (Value: 10, Weight: 5)
-----
```

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.