



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:

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

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

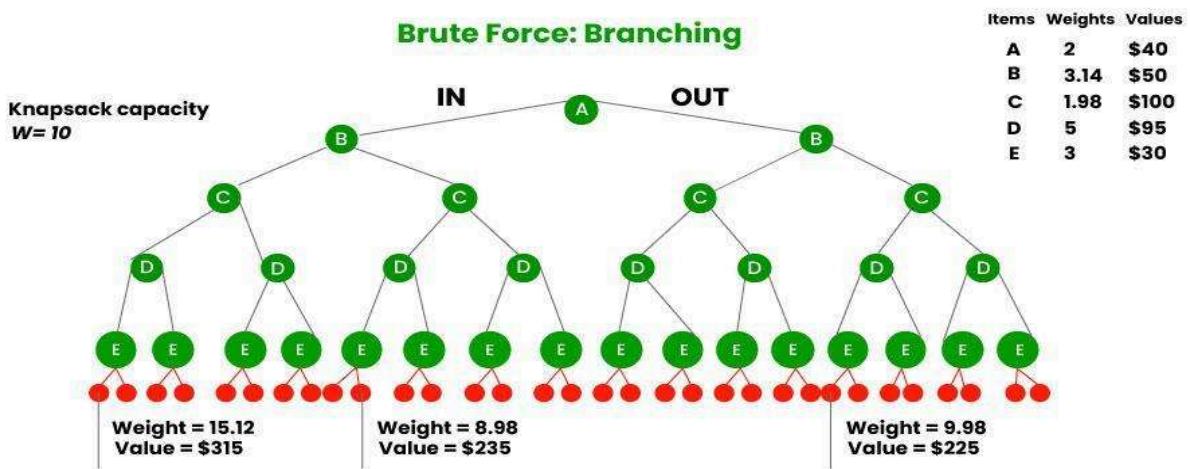
$$\text{where } x[i] \in \{0, 1\}$$

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

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

else:

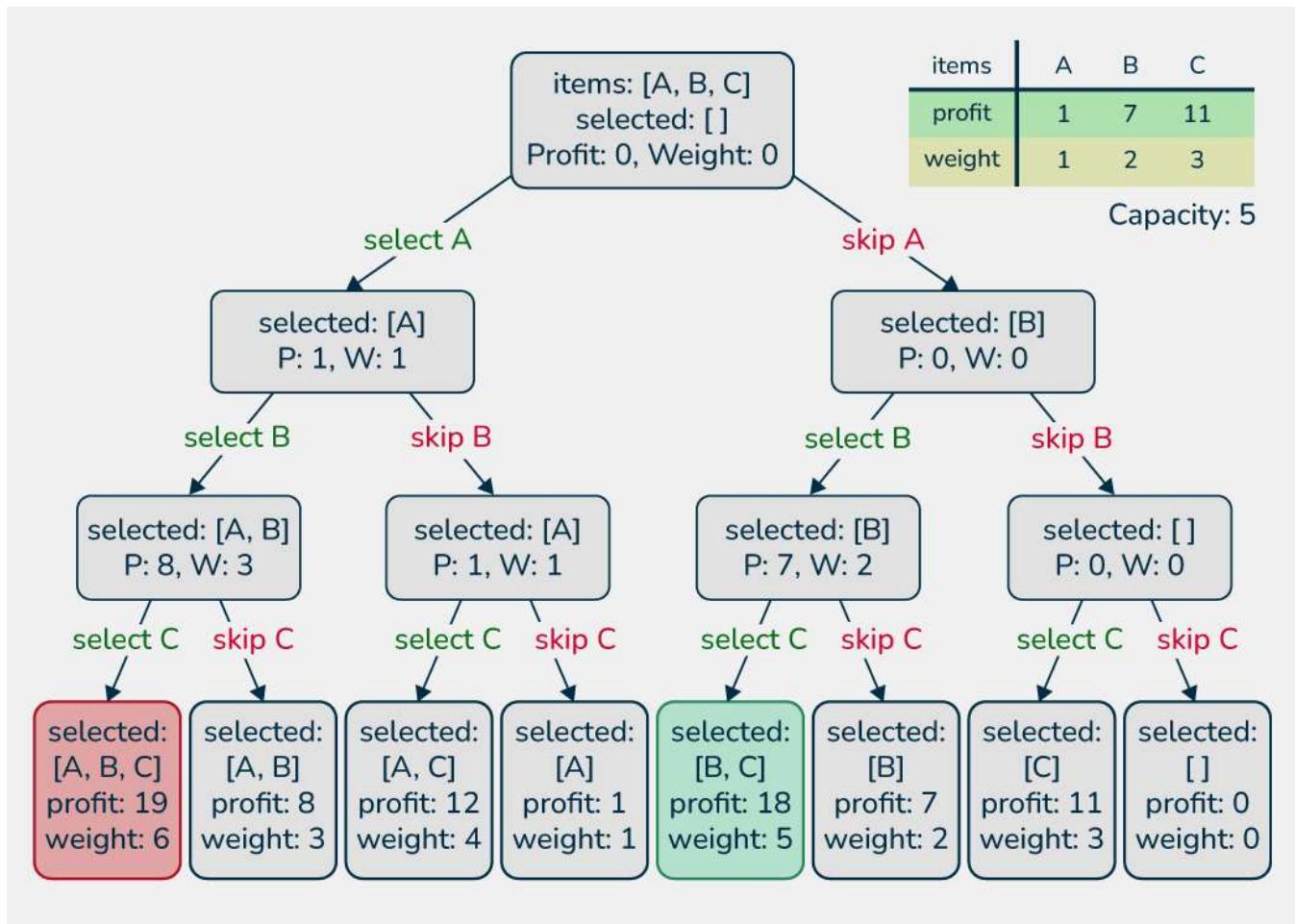
$$dp[i][w] = dp[i-1][w]$$

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	2	3	4	5	6	7
W	\$	0	0	0	0	0	0	0	0
item 1:	3 50	1	0	0	0	50	50	50	50
item 2:	2 40	2	0	0	40	50	50	90	90
item 3:	4 70	3	0	0	40	50	70	90	110
item 4:	5 80	4	0	0	40	50	70	90	110
item 5:	1 10	5	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

```
import java.util.*;
```

```
class Item {
    String name;
    int weight;
    int value;
```

```

boolean perishable;

Item(String name, int weight, int value, boolean perishable) {
    this.name = name;
    this.weight = weight;
    this.value = value;
    this.perishable = perishable;
}

public class DisasterReliefKnapsack {

    // Dynamic Programming 0/1 Knapsack
    public static int knapsackDP(Item[] items, int W, boolean[] selected) {
        int n = items.length;
        int[][] dp = new int[n + 1][W + 1];

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

        // Backtrack to find selected items
        int w = W;

```

```

for (int i = n; i > 0; i--) {
    if (dp[i][w] != dp[i - 1][w]) {
        selected[i - 1] = true;
        w = items[i - 1].weight;
    }
}

return dp[n][W];
}

// Greedy Approach (for approximation)
public static int greedyApproach(Item[] items, int W) {
    Arrays.sort(items, (a, b) -> Double.compare((double)b.value / b.weight, (double)a.value / a.weight));
    int totalValue = 0, totalWeight = 0;

    for (Item item : items) {
        if (totalWeight + item.weight <= W) {
            totalWeight += item.weight;
            totalValue += item.value;
        }
    }
    return totalValue;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter number of available items: ");
    int n = sc.nextInt();

    System.out.print("Enter maximum truck capacity (in kg): ");
}

```

```

int W = sc.nextInt();

Item[] items = new Item[n];
System.out.println("\nEnter item details:");
System.out.println("(Name Weight(kg) UtilityValue Perishable[true/false])");

for (int i = 0; i < n; i++) {
    System.out.print("Item " + (i + 1) + ": ");
    String name = sc.next();
    int weight = sc.nextInt();
    int value = sc.nextInt();
    boolean perishable = sc.nextBoolean();
    items[i] = new Item(name, weight, value, perishable);
}

// Give priority to perishable items by slightly boosting their utility
for (Item item : items) {
    if (item.perishable) {
        item.value += 5; // bonus priority for perishable supplies
    }
}

boolean[] selected = new boolean[n];
int maxValue = knapsackDP(items, W, selected);

System.out.println("\n===== OPTIMAL SOLUTION (Dynamic Programming) =====");
System.out.println("Maximum Utility Value: " + maxValue);
System.out.println("Items to include in truck:");

int totalWeight = 0;
for (int i = 0; i < n; i++) {
    if (selected[i]) {

```

```

        System.out.println( items[i].name + " (Weight: " + items[i].weight + "kg, Value: " +
items[i].value + ")");
        totalWeight += items[i].weight;
    }
}

System.out.println("Total Weight Used: " + totalWeight + " / " + W + " kg");

// Comparison with Greedy Approach
int greedyValue = greedyApproach(Arrays.copyOf(items, items.length));
System.out.println("\n===== GREEDY APPROACH (Approximation) =====");
System.out.println("Estimated Utility Value: " + greedyValue);

sc.close();
}
}

```

Output

```
Enter number of available items: 6
Enter maximum truck capacity (in kg): 20

Enter item details:
(Name Weight(kg) UtilityValue Perishable[true/false])
Item 1: Food 5 30 true
Item 2: Water 7 40 true
Item 3: Medicine 3 50 true
Item 4: Blanket 6 20 false
Item 5: Tent 10 35 false
Item 6: Clothes 4 25 false

===== OPTIMAL SOLUTION (Dynamic Programming) =====
Maximum Utility Value: 130
Items to include in truck:
Medicine (Weight: 3kg, Value: 55)
Food (Weight: 5kg, Value: 35)
Water (Weight: 7kg, Value: 45)
Clothes (Weight: 4kg, Value: 25)
Total Weight Used: 19 / 20 kg

===== GREEDY APPROACH (Approximation) =====
Estimated Utility Value: 125

PS D:\DSA>
```

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.