CSC4202 - Design and Analysis of Algorithms

PRIORITY-BASED FOOD BANK ALLOCATION DURING DISASTER RELIEF

Bringing Hope With Limited Resources

Presented by
Group 6

Chan Ci En (215035)
Khoo Boo Jing (215382)
Loo Huai Yuan (215516)
Tan Yong Jin (217086)

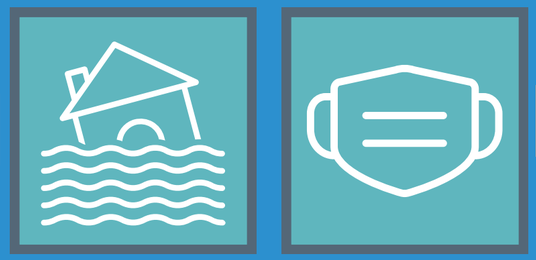
Table of Content

Introduction

Disaster

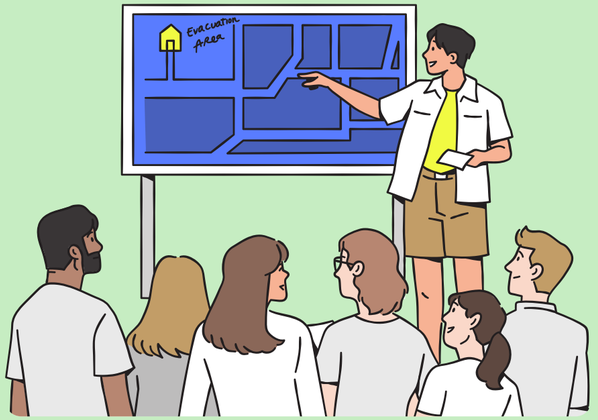
A disaster happens when a hazard seriously affects a community or area, causing widespread damage, loss of life, or disruption that overwhelms the local capacity to cope.

FLOODING



PANDEMIC

SCENARIO



- After a severe natural disaster, **emergency food supplies must be delivered to relief stations** across affected urban and suburban districts.
- Policy: Only full deliveries are allowed – **no partial aid**.

Each relief station:

- Represents a local community.
- Has varying levels of **urgency** and **population size**.
- Incurs **different delivery costs** based on location and terrain.

Major constraints:

The central food bank faces:

Limited delivery budget (fuel, labour, logistics).

Insufficient food supply (not all districts can be helped).

Problem Definition & Project Objective

🧩 Problem Statement:

In the wake of a natural disaster, a **central food bank must allocate limited resources to supply food to various affected districts** due to insufficient food and a strict delivery budget.

This creates a **constrained resource allocation problem**, requiring strategic decisions that balance impact and feasibility based on:

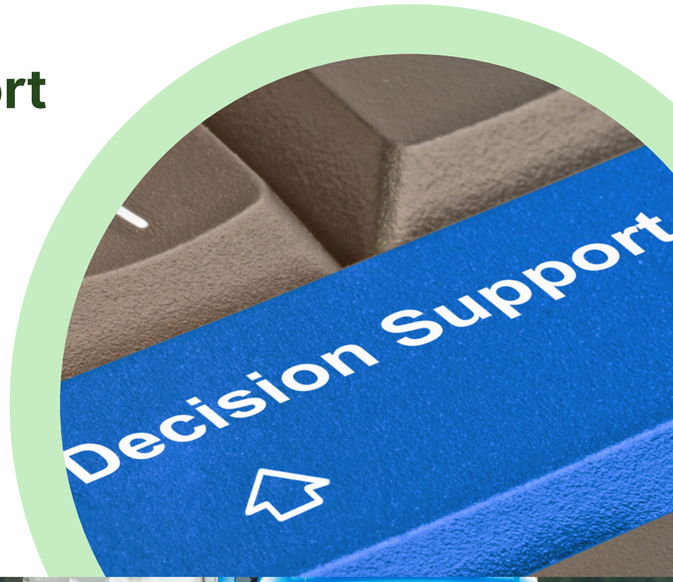
- **District Selection**: Identify which relief stations to serve.
- **Resource Allocation**: Ensure full aid delivery.
- **Priority Management**: Prioritise based on urgency levels.
- **Cost Efficiency**: Stay within a strict delivery budget.

Problem Definition & Project Objective

🎯 Objective:

To design and implement an **optimisation-based decision-support model** that:

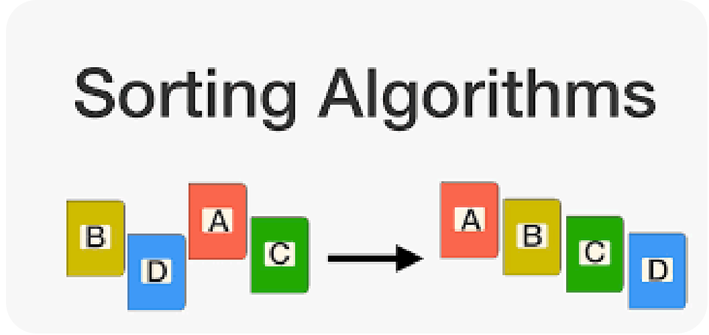
- Selects the most **impactful subset of districts** to receive full aid.
- **Balances** urgency, population served, and delivery cost in decision-making.
- **Maximises total weighted aid impact** under resource constraints.
- Provides a **scalable and adaptable framework** for disaster scenarios.

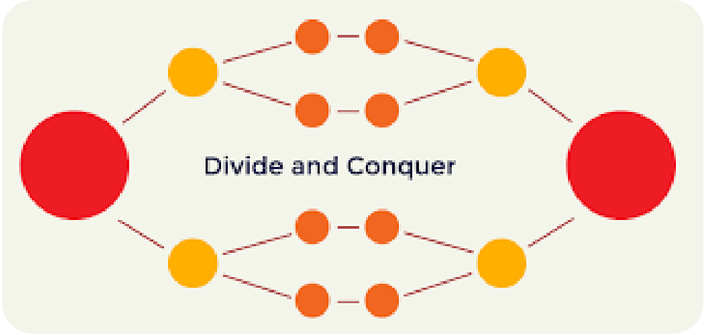


Algorithm Selection

Sorting Algorithm

It is great for organizing data like ranking districts based on need or cost. According to Yu and Li (2022), Sample Sort performs well on modern multi-core systems. However, sorting only provides an order. It does not evaluate combinations of districts or enforce resource limits. So while it is useful for pre-processing, it cannot solve the core optimization problem on its own.

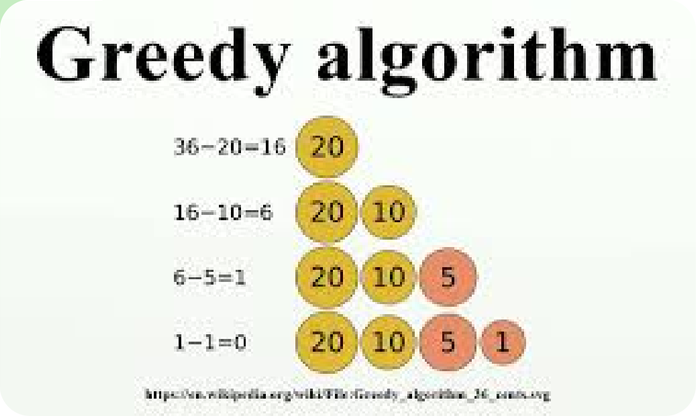


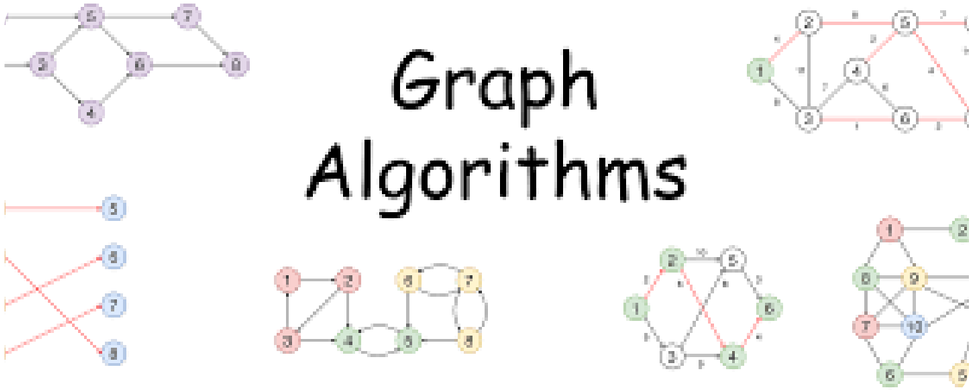


Divide and Conquer

It is effective in problems like sorting and routing, especially when parallelism is needed. Ren et al. (2018) showed how it works well in large SAT problems. However, DAC struggles with overlapping subproblems and doesn't track state. Since district selection in food aid is highly interdependent, DAC cannot guarantee global optimality.

Algorithm Selection





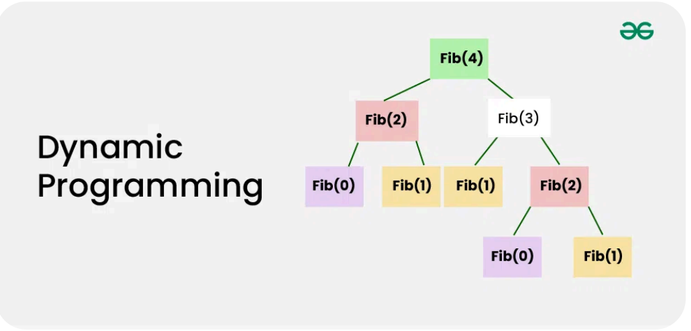
Greedy algorithms are fast and simple. As Alsalmi (2020) explains, they work well in algorithms like Huffman coding or Prim's MST. But in our case, greedy methods risk selecting locally best districts while missing better combinations. Alkaabneh et al. (2021) showed that greedy allocation underperformed dynamic programming in a real food bank case, where fairness and constraint handling were critical.

Dijkstra's algorithm is great for route planning. It finds the shortest path from a central hub to each district. AbuSalim et al. (2020) confirmed its efficiency in real networks. However, Dijkstra's works one destination at a time. It does not optimize multi-stop delivery routes or help decide which districts should be served. So it's helpful for logistics, but not suitable for our core allocation logic.

Algorithm Selection

Dynamic Programming

It’s ideal for resource-constrained optimization problems like ours. Posypkin and Thant Sin (2016) showed that DP solves large-scale 0/1 Knapsack problems with high accuracy. Each district in our problem is like an item in the knapsack—with a cost and a value. DP evaluates all combinations, tracks state, and guarantees the optimal selection of districts, making it ideal for fair, efficient food allocation



Final Decision

In conclusion, after reviewing all algorithm options, we selected Dynamic Programming because it aligns perfectly with our problem's structure. It ensures fairness, respects constraints, and delivers optimal results in crisis situations

Scenario

Inputs:

- **budget (int)**: Total available delivery cost capacity.
- **districts (List)**: A list of districts, each with:
  - id (String): Unique district identifier
  - cost (int): Delivery cost
  - population (int): People in need
  - urgency (String): Priority level ("Critical", "High", "Moderate")
  - weightedValue (double): Calculated as population × urgency weight

Outputs:

- **maxValue (double):** Maximum total priority-weighted population served.
- **selectedDdistricts (List)**: Districts chosen for aid delivery.

Objective Function

Maximize total weighted value:

$$\text{Maximize: } Z = \sum_{i \in S} (\text{population}_i \times \text{urgencyWeight}_i)$$

$$\text{Subject to: } \sum_{i \in S} \text{cost}_i \leq \text{budget}$$

Where:

- S is the set of selected districts.
- $\text{urgencyWeight}_i \in \{2.0, 1.5, 1.0\}$ based on urgency tier.

Pseudocode of the main DP algorithm

```
FUNCTION allocateFoodBanks(budget, districts):

    n ← number of districts

    CREATE 2D array dp[0...n][0...budget] initialized to 0

  FOR i from 1 to n:

      cost ← districts[i-1].cost

      value ← districts[i-1].value


      FOR w from 1 to budget:

          IF cost ≤ w THEN

              dp[i][w] ← MAX(dp[i-1][w], dp[i-1][w - cost] + value)

          ELSE

              dp[i][w] ← dp[i-1][w]
```

1. Initialize n = number of districts
2. Initialize DP table to all 0
3. Recurrence: choose to take or skip district i

Pseudocode of the main DP algorithm

4.Backtrack to find selected districts

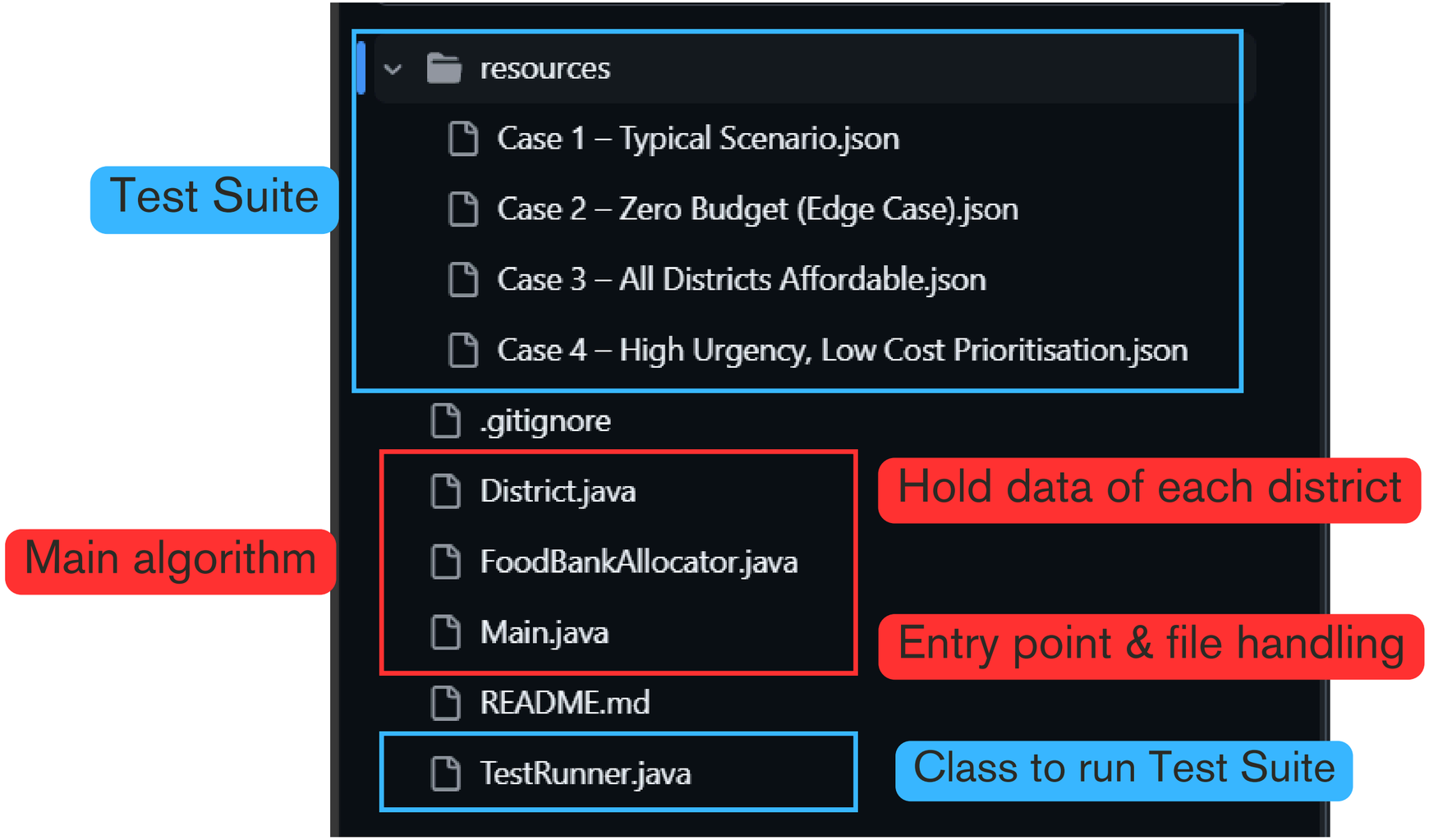
```
// Backtracking to find selected districts
selected ← empty list
w ← budget

FOR i from n down to 1:
    IF dp[i][w] ≠ dp[i-1][w] THEN
        ADD districts(i-1) to selected
        w ← w - districts[i-1].cost
```

5. Return the maximum value found and selected districts

```
RETURN dp[n][budget], selected
```

Implementation



Implementation

District.java

```java
public class District {
    private final String id;
    private final int cost;
    private final int population;
    private final String urgency;
    private final double weightedValue;

    public District(String id, int cost, int population, String urgency) {
        this.id = id;
        this.cost = cost;
        this.population = population;
        this.urgency = urgency;
        this.weightedValue = calculateWeightedValue();
    }
```

```java
    private double calculateWeightedValue() {
        double weight = switch (urgency.toLowerCase()) {
            case "critical" -> 2.0;
            case "high" -> 1.5;
            case "moderate" -> 1.0;
            default -> 1.0;
        };
        return population * weight;
    }
```

```java
    // Getters
    public String getId() { return id; }
    public int getCost() { return cost; }
    public int getPopulation() { return population; }
    public String getUrgency() { return urgency; }
    public double getWeightedValue() { return weightedValue; }

    @Override
    public String toString() {
        return String.format("District %s: Cost=%d, Population=%d, Urgency=%s, WeightedValue=%.1f",
                id, cost, population, urgency, weightedValue);
    }
}
```

FoodBankAllocator.java

```java
import java.util.ArrayList;
import java.util.List;

public class FoodBankAllocator {
    private final int budget;
    private final List<District> districts;
    private double[][] dp;

    public FoodBankAllocator(int budget, List<District> districts) {
        this.budget = budget;
        this.districts = districts;
    }

    public AllocationResult allocateFoodBanks() {
        int n = districts.size();

        // Initialize dp table to 0
        this.dp = new double[n + 1][budget + 1];

        // Fill dp table
        for (int i = 1; i <= n; i++) {
            District district = districts.get(i - 1);
            int cost = district.getCost();
            double value = district.getWeightedValue();

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

        // Backtrack to find selected districts
        List<District> selectedDistricts = new ArrayList<>();
        int w = budget;

        for (int i = n; i > 0; i--) {
            if (dp[i][w] > dp[i - 1][w]) {
                selectedDistricts.add(districts.get(i - 1));
                w -= districts.get(i - 1).getCost();
            }
        }

        return new AllocationResult(dp[n][budget], selectedDistricts);
    }

    public static class AllocationResult {
        public final double maxValue;
        public final List<District> selectedDistricts;

        public AllocationResult(double maxValue, List<District> selectedDistricts) {
            this.maxValue = maxValue;
            this.selectedDistricts = selectedDistricts;
        }
    }
}
```

Inner class to return multiple values at once

Exactly the same as pseudocode

Main.java

File handling: read and parse JSON files

```java
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage: java Main <filename>");
            System.out.println("Example: java Main \"resources/Case 1 - Typical Scenario.json\"");
            return;
        }

        String filename = args[0];
        File file = new File(filename);
        if (!file.exists()) {
            System.out.println("Error: File '" + filename + "' not found.");
            return;
        }

        System.out.println("=== Food Bank Allocation Algorithm ===");
        System.out.println("Processing: " + filename);
        System.out.println("-".repeat(50));

        // Read JSON and parse data
        String content = readFile(filename);
        if (content == null) return;

        int[] budget = {0};
        List<District> districts = new ArrayList<>();
        parseJSON(content, budget, districts);

        if (districts.isEmpty()) {
            System.out.println("Error: No districts found in " + filename);
            return;
        }

        // Run algorithm
        FoodBankAllocator allocation = new FoodBankAllocator(budget[0], districts);

        System.out.println("Input Details:");
        System.out.println("   Budget: $" + budget[0]);
        System.out.println("   Districts: " + districts.size());
        for (District district : districts) {
            System.out.println("   " + district);
        }

        FoodBankAllocator.AllocationResult result = allocation.allocateFoodBanks();
        System.out.println("\nOutput Result:");
        System.out.printf("   Maximum Value: %.1f%n", result.maxValue);
        System.out.println("   Selected Districts: " + result.selectedDistricts.size());
        for (District selectedDistrict : result.selectedDistricts) {
            System.out.println("   " + selectedDistrict);
        }

        int totalCost = result.selectedDistricts.stream().mapToInt(District::getCost).sum();
        System.out.printf("   Total Cost: $%d (Budget: $%d)%n", totalCost, budget[0]);
    }

    private static String readFile(String filename) {
        try {
            return new String(Files.readAllBytes(Paths.get(filename)));
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
            return null;
        }
    }

    private static void parseJSON(String json, int[] budget, List<District> districts) {
        try {
            json = json.replaceAll("\\s+", "");

            // Extract budget
            budget[0] = Integer.parseInt(json.split("\"budget\":")[1].split(",")[0]);

            // Extract districts
            String districtsPart = json.split("\"districts\":\\[")[1].split("\\]")[0];

            for (String district : districtsPart.split("\\{")) {
                if (district.isEmpty()) continue;
                district = district.replaceAll("\\},?$", "");

                String[] fields = district.split(",");
                String name = fields[0].split(":")[1].replace("\"", "");
                int cost = Integer.parseInt(fields[1].split(":")[1]);
                int value = Integer.parseInt(fields[2].split(":")[1]);
                String urgency = fields[3].split(":")[1].replace("\"", "");

                districts.add(new District(name, cost, value, urgency));
            }
        } catch (Exception e) {
            System.err.println("Error parsing JSON: " + e.getMessage());
        }
    }
}
```

**CREATED FOR TESTING PURPOSE **

TestRunner.java

2. Loop all test cases and run main function for all

1. Get all test cases from the “resources” directory

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

public class TestRunner {
    private static final String RESOURCES_DIR = "resources/";

    public static void main(String[] args) {
        System.out.println("=== Food Bank Allocation Algorithm Test Suite ===\n");

        File[] testFiles = getAvailableTestCases();
        if (testFiles.length == 0) {
            System.out.println("No test case files found in resources directory.");
            return;
        }

        for (File testFile : testFiles) {
            System.out.println("Running Test Case: " + testFile.getName());
            System.out.println("-".repeat(40));

            // Call Main.java's main function for this file
            String[] mainArgs = {testFile.getPath()};
            Main.main(mainArgs);

            System.out.println("\n" + "=".repeat(60) + "\n");
        }

        System.out.println("=== All Test Cases Completed ===");
    }

    private static File[] getAvailableTestCases() {
        File resourcesDir = new File(RESOURCES_DIR);
        if (resourcesDir.exists() && resourcesDir.isDirectory()) {
            File[] testFiles = resourcesDir.listFiles((dir, name) -> name.endsWith(".json"));
            if (testFiles != null) {
                Arrays.sort(testFiles);
                return testFiles;
            }
        }
        return new File[0];
    }
}
```

Sample Input and Output

To validate the functionality of the proposed food bank allocation system, the system reads input from a JSON file that contains each district-level data in the following attributes:

- Cost
- District Name
- Urgency Level
- Weighted Value

To validate the system, we use the following JSON input file:


```
{
  "budget": 5,
  "districts": [
    {"name": "A", "cost": 2, "value": 100, "urgency": "High"},
    {"name": "B", "cost": 1, "value": 80, "urgency": "Critical"},
    {"name": "C", "cost": 3, "value": 120, "urgency": "Moderate"},
    {"name": "D", "cost": 2, "value": 90, "urgency": "High"},
    {"name": "E", "cost": 1, "value": 60, "urgency": "Moderate"}
  ]
}
```

OUTPUT

After processing the input data, the sample output is presented below:

Justification:

- Prioritise high-impact districts.
- Optimise aid distribution within tight budget constraints
- Ensure fairness and strategic selection.

Screenshot

```
===Output Result===
Maximum Value: 445.0
Selected Districts:
District D: Cost=2, Population=90, Urgency=High, WeightedValue=135.0
District B: Cost=1, Population=80, Urgency=Critical, WeightedValue=160.0
District A: Cost=2, Population=100, Urgency=High, WeightedValue=150.0
```

Correctness Analysis

To ensure the correctness and reliability of algorithm, we employed two validation strategies:

Theoretical Validation

- **Using the loop invariant method, a formal approach to proving algorithm correctness.**

Empirical Validation

- **Through a series of controlled test scenarios, using both manual and programmatic result verification.**

THEORETICAL VALIDATION:

Loop Invariant Method

Loop Invariant:
At the beginning of each iteration, the value stored in dp[i][w] correctly represents the maximum achievable value using the first i districts with a total budget of w.

Maintenance:

Each iteration, the algorithm evaluates:

- Exclusion of district i
- Inclusion of district i

To ensure that the algorithm selects the optimal decision whether to include or exclude the current district.

Termination:

After completing all iterations, the entry dp[i][w] contains the maximum total value without exceeding the budget. The algorithm is correct.

EMPIRICAL VALIDATION:

Input-Output Testing

The testing process involved three main steps for each case:

1. Manual Calculation: The expected result was computed manually based on the problem formulation.

2. Program Execution: The implemented Java algorithm was executed using the same input parameters.

3. Result Verification: The output values and selected districts were compared with the manually derived expectations to confirm accuracy.

In all test cases, the algorithm’s outputs precisely matched the manually calculated results. These consistent outcomes provide strong empirical evidence of the algorithm's correctness.

Case	Description	Expected Behavior
Typical Scenario	Districts have varied costs and urgency levels under a moderately constrained budget. (e.g. budget = 5).	Selects the optimal subset of districts to maximise total priority-weighted value.
Zero Budget (Edge Case)	The budget was set to zero, with districts present but unaffordable. (e.g. budget = 0).	Returns a total value of 0 with no districts selected.
All Districts Affordable	The combined cost of all districts does not exceed the available budget (e.g. budget = 10).	All districts are included in the final selection.
High Urgency, Low Cost	Tight budget with some high-urgency, low-cost districts included. (e.g. budget = 3).	Prioritises districts with the highest urgency-to-cost ratio.

Time Complexity Analysis

The algorithm employs a bottom-up dynamic programming approach using a two-dimensional table of size (n + 1) × (W + 1). The time complexity analysis is as follows:

Case	Time Complexity	Justification
Best Case	O(n × W)	All table entries are filled, typically for small inputs or bounded budget values
Average Case	O(n × W)	Represents typical scenarios where all subproblems are evaluated
Worst Case	O(n × W)	All entries must be computed due to the absence of shortcutting or pruning

In all cases, the time complexity remains O(nW), the approach is within the defined state space.

Conclusion

Project Summary

- Designed and implemented a decision-support model for food bank allocation during disaster relief.
- Utilised the 0/1 Knapsack Dynamic Programming algorithm to optimise aid delivery under:
  - 🚚 Budget constraints
  - ⚠️ Urgency levels
  - 👭 Population size
  - ❌ No partial delivery policy
- The model successfully maximised its impact by selecting the most effective combination of districts within the budget.

Reflection

- Strengthened our understanding of algorithm design and real-world optimisation.
- Learned to transform theory into a life-impacting application.
- Gained valuable experience in collaboration, problem-solving, and iterative development.

THANK YOU

**Optimisation isn't about choosing who matters more,
it's about maximising the impact of what we have.**