FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

CSC4202 Design and Analysis of Algorithms

Semester 2 2024/2025

Project: Priority-Based Food Bank Allocation During Disaster Relief

GROUP 6

NAME	MATRIC NO.
CHAN CI EN	215035
KHOO BOO JING	215382
LOO HUAI YUAN	215516
TAN YONG JIN	217086

Table of Contents

1.0 Introduction

1.1 Scenario Overview

In the aftermath of a severe natural disaster, a central food bank must deliver and distribute emergency supplies, especially food, to multiple temporary relief stations established across affected urban and suburban districts. Each station represents a community with specific needs that are defined by population size and urgency, and incurs different delivery costs due to distance, terrain, and logistics.

The food bank faces two critical constraints that significantly shape its disaster relief strategy. First, it faces a **limited delivery budget**, which encompasses essential operational components such as fuel, manpower, and transportation logistics. Second, it contends with an **insufficient food supply**, making it impossible to assist every affected location.

Given these limitations, the food bank must adopt a systematic approach to resource allocation that is both fair and effective. Specifically, it must determine which relief stations will receive full assistance, as partial deliveries are not permitted under the distribution policy. The goal is to **maximise the total number of people served**, while giving higher priority to more urgent cases **without exceeding the overall delivery budget**. This requires careful selection and prioritisation to ensure that the humanitarian impact is maximised under stringent resource constraints.

1.2 Problem Definition

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. Hence, the problem constitutes a constrained resource allocation and selection task by requiring a balance between impact and feasibility. The decision-making components include:

- **District Selection**: Identifying which relief stations to serve.
- **Resource Allocation**: Distributing enough food to fully serve selected districts.
- **Priority Management**: Factoring for urgency levels (e.g., critical vs. moderate need) to prioritise aid delivery.
- **Cost Efficiency**: Ensuring that the total delivery cost does not exceed the available budget.

2.0 Importance of Optimal Solution

2.1 Necessity of Efficient Decision-Making

Effective optimisation is essential in disaster relief scenarios, where every resource, such as time, fuel, labour, and supplies, is scarce. An unstructured or ad-hoc allocation method may

lead to significant inefficiencies, including overlooking high-need or high-priority areas. An optimal, algorithmically guided strategy offers several advantages:

- **Maximising Humanitarian Impact**: Ensures that the largest possible number of people receive aid, with weighted consideration given to urgency.
- **Efficient Use of Scarce Resources**: Supports cost-effective decision-making, enhancing the reach and effectiveness of each delivery.
- **Transparent and Fair Allocation**: Promotes accountability and public trust by demonstrating that distribution decisions are based on systematic and equitable criteria.
- **Scalability and Adaptability**: Enables rapid adaptation to dynamic circumstances, such as shifting needs, changing budgets, or expanding geographic coverage.

2.2 Real-World and Technical Value

This model is highly applicable to real-world disaster response efforts and offers valuable technical and societal benefits:

- **Social Impact**: Supports equitable and efficient distribution of emergency aid under urgent and resource-constrained conditions.
- **Operational Readiness**: Addresses logistical challenges in allocating and delivering supplies to multiple high-need locations.
- **Visualisation and Communication**: Integrates with GIS tools to clearly map aid coverage and affected areas, improving transparency and coordination.
- **Technical Demonstration**: Serves as a practical case study for applying constrained optimisation and multi-criteria decision-making techniques in algorithm design.
- **Extensibility and Flexibility**: The framework is adaptable to include routing logic, real-time updates, and other enhancements, and is well-suited for implementation in languages like Java or Python.

3.0 Algorithm Suitability Review

3.1 Sorting Algorithms

Strengths:
Sorting algorithms remain essential for data preparation and prioritisation. Notably, Sample Sort is widely recognised for its scalability and performance in parallel and distributed environments. In a 2022 survey, Yu and Li found it highly effective on manycore processors, improving efficiency by dividing data into buckets processed in parallel, making it ideal for sorting large datasets efficiently (Yu & Li, 2022).

Weaknesses:
While sorting provides an ordered structure, it does not evaluate combinations necessary to satisfy capacity constraints. Therefore, although useful for heuristic ordering, it cannot guarantee the optimal selection of districts in the food bank allocation problem.

Suitability for the Current Problem:
Sorting is useful in the pre-processing stage. For instance, ranking districts based on the need-to-cost ratio, but it lacks the ability to enforce constraints or consider interdependencies between choices. As a result, it cannot be used as a standalone solution for constrained optimisation tasks such as food aid allocation, where the selection of one district affects the feasibility of selecting others.

3.2 Divide-and-Conquer (DAC)

Strengths:
The divide-and-conquer approach breaks a problem into independent sub-problems, solves them individually, and then combines their solutions to form the final result. As Alsalmi (2020) explains, this method is well-suited for multicore systems because sub-problems can be distributed and solved in parallel. Common applications include sorting algorithms like quicksort, routing algorithms, and scheduling tasks.

A strong application of this paradigm can be found in the work of Ren et al. (2018), who developed a DAC-based algorithm, which is a complex variant of the satisfiability (SAT) problem. Their method divides the SAT formula into two sub-formulas, solves them independently, and merges the results, handling variable overlaps carefully to maintain solution validity. By introducing a segmentation strategy, they reduced variable overlap and improved merge efficiency. Experimental results showed that this parallel DAC method outperformed traditional solvers in both the number of instances solved and the number of solutions generated, particularly in large-scale scenarios.

Weaknesses:
While DAC is efficient for problems that can be clearly decomposed, it is less suitable when sub-problems overlap or when global constraints must be enforced, such as in resource-limited optimisation tasks. Although Alsalmi (2020) mentions various dynamic programming techniques built on top of divide-and-conquer (e.g., bottom-up and top-down DP), DAC in its pure form lacks the memory mechanism needed for handling overlapping subproblems, making it less effective for problems like food bank allocation.

Suitability for the Current Problem:
In the context of food bank allocation during disaster relief, DAC's limitations become apparent. The selection of aid distribution zones involves maximising total priority-weighted population served within a fixed delivery budget. Since DAC cannot inherently track the impact of one selection on the feasibility of others, it risks suboptimal solutions. Therefore, despite its strengths in scalability and parallelisation, DAC is not suitable for this problem. Instead, dynamic programming provides the necessary structure to handle state dependencies and ensure global optimality.

3.3 Greedy Algorithms

Strengths:
Greedy algorithms operate by making the locally optimal choice at each step in hopes of finding a global optimum. As Alsalmi (2020) notes, they are fast and simple to implement, often producing solutions close to optimal in practical time. Common applications include Kruskal’s and Prim’s algorithms for minimum spanning trees, and Huffman coding for data

compression.

Weaknesses:
However, greedy algorithms do not backtrack and make irreversible decisions, which can lead to suboptimal solutions in complex problems. Alsalmi (2020) emphasises that unless a problem is proven to be optimally solvable by a greedy strategy, this method may result in only a locally optimal solution. For tasks like equitable food distribution, which require fairness, constraint satisfaction, and evaluating trade-offs between competing districts, greedy approaches may fall short. This limitation is further supported by Alkaabneh et al. (2021), who compared greedy allocation strategies with a dynamic programming-based model in real-world food bank operations. Their findings showed that the dynamic programming approach resulted in a 7.73% increase in overall utility and a 3.0% improvement in nutritional effectiveness, clearly demonstrating that greedy algorithms underperform in constrained, fairness-critical environments like food distribution during crises.

Suitability:
Greedy algorithms may serve as a fast, initial approximation method or heuristic, particularly useful when execution time is a priority and problem constraints are minimal. However, in scenarios like food bank allocation, where multiple constraints, fairness, and optimality are critical, greedy algorithms are not suitable as the primary solution method, as they cannot guarantee the best global outcome under resource limits.

3.4 Dynamic Programming (DP)

Strengths:
Dynamic programming solves problems by breaking them into overlapping subproblems and storing intermediate results, preventing redundant computations. According to Alsalmi (2020), DP is computationally efficient, parallelizable, and adaptable to changing requirements. It's especially suitable for complex optimisation problems with constraints, such as sequence alignment or pathfinding.

Furthermore, Posypkin and Thant Sin (2016) conducted an in-depth experimental study comparing different DP methods for the classic 0/1 Knapsack problem, which is structurally similar to the Priority-Based Food Bank Allocation scenario. Their findings reveal that table-based DP, list-based DP, and enhanced list-based DP (with dominance rules and upper bounds) can solve large-scale instances with remarkable accuracy. The study confirmed that, despite higher space complexity, DP methods consistently provide exact and optimal solutions, significantly outperforming heuristic approaches in both correctness and fairness key considerations in disaster aid allocation.

Weaknesses:
DP may have higher space complexity and needs a well-structured formulation. Despite this, the trade-off is worthwhile due to its guaranteed correctness and flexibility.

Suitability:
For the Priority-Based Food Bank Allocation problem, dynamic programming is the most suitable approach. The problem can be modelled as a 0/1 Knapsack problem:

- Each district has a cost (resources needed) and value (priority/impact).
- The objective is to maximise total impact under a total resource constraint.
- The DP approach evaluates all valid combinations efficiently using memoisation.

Unlike greedy algorithms that may overlook better combinations due to local decisions or divide-and-conquer methods that do not guarantee global optimality, DP delivers a globally optimal solution, critical in disaster relief, where equity, transparency, and impact maximisation are paramount. As shown by Posypkin and Thant Sin (2016), even large instances of the Knapsack problem (e.g., 500 items) were efficiently solved using DP variants, proving its robustness and practical utility in real-world scenarios.

3.5 Graph Algorithms (Dijkstra's Algorithm)

Strengths:
Dijkstra's algorithm is a robust solution for finding the shortest path from a single source to all other nodes in a weighted graph with non-negative edge values. It runs in $O((E + V) \log V)$ time when implemented with a binary heap, and its correctness and efficiency have been well-established (GeeksforGeeks, 2024). This makes it ideal for planning individual delivery routes from a central food bank to specific districts.

Weaknesses:
However, Dijkstra's algorithm only optimises each path independently. It does not create an efficient multi-stop route or cover all selected districts in a single trip. For that purpose, algorithms like the Travelling Salesman Problem (TSP) or spanning tree models would be necessary. Thus, while it is excellent for individual route planning, it is not sufficient for optimising complex delivery tours.

Suitability:
Dijkstra's algorithm is well-suited for situations where food deliveries are made to one district at a time from a central hub, especially when the goal is to minimise the delivery distance or cost to each location. However, it is not suitable for route optimisation when a single trip must cover multiple districts efficiently, as it does not account for total route cost across multiple stops. In such cases, more advanced route planning algorithms are required.

3.6 Final Algorithm Selection

After evaluating multiple algorithmic paradigms, including Sorting, Divide-and-Conquer (DAC), Greedy Algorithms, Dynamic Programming (DP), and Graph Algorithms, it is evident that Dynamic Programming (DP) is the most suitable and effective solution for the Priority-Based Food Bank Allocation problem.

Sorting methods, while useful for prioritisation, cannot handle the interdependent constraints necessary for fair and optimal resource distribution. Divide-and-conquer techniques offer strong performance in independent sub-problem scenarios but fail to retain the necessary state information across overlapping decision paths. Greedy algorithms, though fast and easy to implement, make irreversible local decisions and often produce suboptimal results when fairness and capacity constraints are involved. Graph-based algorithms like Dijkstra's

are powerful in routing problems but are not applicable to the core allocation logic, which involves selection under resource limits rather than pathfinding.

In contrast, Dynamic Programming offers a globally optimal solution, efficiently handling overlapping subproblems and ensuring that all feasible combinations of districts are evaluated. By modelling the problem as a 0/1 Knapsack scenario, DP allows for a structured approach to maximise the total impact of food aid distribution under fixed supply constraints. As supported by Posypkin and Thant Sin (2016) and Alkaabneh et al. (2021), DP consistently outperforms heuristic approaches in correctness, fairness, and real-world applicability, even in large-scale, resource-limited environments. Therefore, Dynamic Programming is chosen as the final solution paradigm due to its proven ability to meet the complex demands of humanitarian resource allocation, delivering both accuracy and accountability in crisis response efforts.

4.0 Model Development of the Scenario

4.1 Overview

The model developed for this project is based on the 0/1 Knapsack Dynamic Programming approach. The key objective is to allocate limited food delivery resources to selected districts affected by a disaster, in a way that maximises the overall humanitarian impact, which is quantified as the total priority-weighted population served, without exceeding the predefined delivery budget.

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.
- **selectedDistricts** (List): Districts chosen for aid delivery.

4.2 Objective Function and Constraints

Objective Function:
Maximise 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.

4.3 Constraints

- Delivery cost for all selected districts must not exceed the total budget.
- No partial deliveries: each district is either fully included or excluded (0/1 selection)
- Urgency weights are fixed and non-negative

4.4 Modelling Challenges

- Need to balance impact (population served) with feasibility (cost).
- High-priority districts may be expensive, requiring trade-offs.
- The binary nature of aid (0/1 inclusion) increases decision complexity.

5.0 Algorithm Design

5.1 Overview of Chosen Paradigm

Why Dynamic Programming(DP) Fit This Problem

Problem Requirement	How DP Addresses It
Binary Selection	0/1 Knapsack naturally models yes/no decisions for each district.
Urgency prioritization	Priority weights multiply population values, embedding urgency into the objective.

Budget constraint	DP table tracks the maximum value achievable for every possible sub-budget.
Optimality guarantee	Exhaustively evaluates combinations without greedy shortcuts.

Key DP Components

1. **State Definition**

- dp[i][w]: Maximum value (weighted population) achievable with the first i districts and budget w.

2. **Recurrence Relation**

- Example:
  if cost[i] ≤ w:
    dp[i][w] = max(dp[i-1][w], dp[i-1][w - cost[i]] + value[i])
  else:
    dp[i][w] = dp[i-1][w]

3. **Backtracking**

- Trace back from dp[n][B] to identify selected districts.

5.2 Pseudocode and Recurrence

```
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]

    // 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

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

The process begins with an input list of districts. Each district includes two key pieces of information: the delivery cost and the priority-weighted population, which reflects both the size of the population and how urgently they need help.

To find the best combination of districts to serve, a Dynamic Programming (DP) table is created. This table has rows for each district and columns for every possible budget amount, from 0 up to the total available budget. The table is filled with zeros at the start.

Next, the algorithm goes through each district, one by one, and for each possible budget value. If the current district's delivery cost is less than or equal to the current budget, the algorithm has two choices: skip the district or include it. It picks the option that gives the highest total priority-weighted population. If the cost is more than the budget, the district is automatically skipped for that budget level.

Next, the algorithm goes backwards through the table to find out which districts were actually selected to reach the best result. This is called backtracking. The algorithm outputs two things: the maximum total priority-weighted population that can be served within the budget, and the list of selected districts that make up this optimal solution.

5.3 Flowchart and Explanation

The Dynamic Programming (DP) approach for the food bank allocation problem follows a structured sequence of steps to ensure optimal resource distribution under budget constraints. The process begins with input preparation, where each district's delivery cost and priority-weighted population (urgency × population) are compiled.

Next, a DP table is initialised with dimensions (n+1) × (budget+1), where n is the number of districts. This table stores the maximum achievable weighted population for every possible sub-budget, starting from zero up to the total available budget. The table is initially filled with zeros, representing no districts served with a zero budget.

The core of the algorithm involves iteratively filling the DP table. For each district, the algorithm evaluates whether including it (if the cost permits) yields a higher weighted population than excluding it. This decision is made by comparing:

- The value of not serving the district (retaining the previous best solution for the same budget).
- The value of serving the district (adding its weighted population to the best solution for the remaining budget after deducting its cost).

Once the table is fully populated, backtracking identifies the selected districts. Starting from the final cell (dp[n][budget]), the algorithm traces backwards to determine which districts contributed to the optimal solution. If the value in a cell differs from the cell above, the corresponding district was included.

This method guarantees an optimal, fair, and transparent allocation, ensuring the highest humanitarian impact given the constraints. The flowchart visually captures this logical progression from input to output, with clear decision points for inclusion/exclusion of districts.

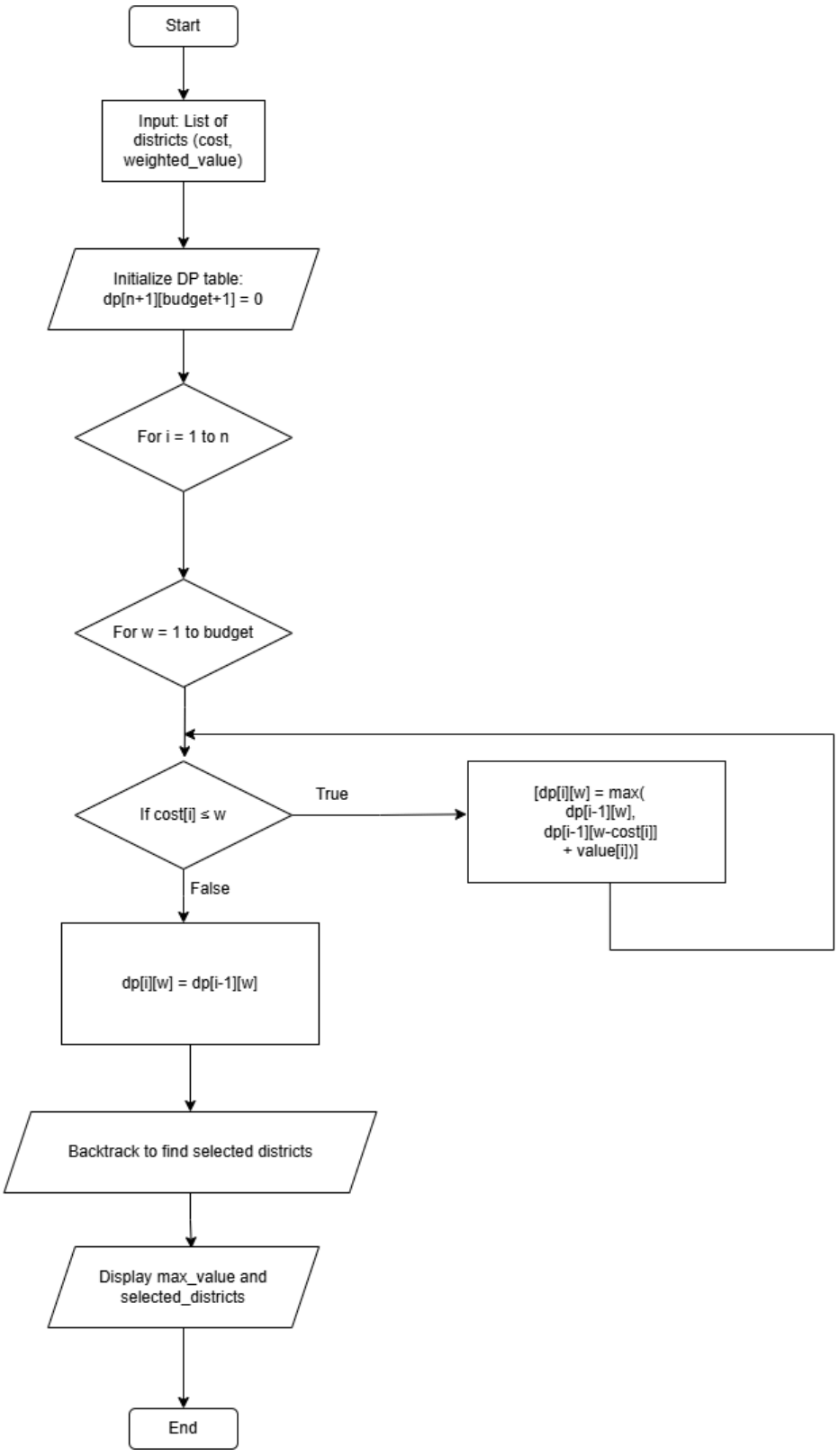


Figure 1: Flowchart of the Dynamic Programming Approach for Food Bank Allocation

6.0 Algorithm Specification

6.1 Input format

The input is provided as a JSON file structured as follows:

```
{
  "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"}
  ]
}
```

Table 1: Urgency Weight Mapping

Tier	Weight
Critical	2.0
High	1.5
Moderate	1.0

6.2 Output format

The algorithm returns an AllocationResult object containing two values:

- **maxValue:** A double representing the maximum total weighted value achieved, i.e., the sum of (population × urgency weight) for selected districts within the given budget.
- **selectedDistricts**: A List<District> object representing the districts that were selected for food allocation.

6.3 Constraints

1. Resource Limits:
    - Total selected costs ≤ Budget (B)
    - 1 ≤ B ≤ 10,000 (practical upper bound)
    - 1 ≤ Number of districts ≤ 100

2. Computational:
    - Time: O(n×B) (DP solution)
    - Space: O(n×B)

3. Logical:
    - No district splitting (0/1 selection)
    - Urgency weights are fixed and non-negative

6.4 Assumptions

1. Problem Scope:
    - Delivery costs are deterministic and known in advance
    - Urgency tiers are pre-classified and immutable

2. Behavioral:
    - Ties in priority are resolved by selecting the lower-cost district
    - All input data is valid (no negative costs/populations)

3. Environmental:
    - The budget is fixed during allocation
    - No real-time updates (static problem)

Why This Specification Matters

This detailed specification acts like an instruction manual for developers working on the food bank allocation system. By defining exactly what inputs are needed (like budget numbers and district details) and what outputs to expect (such as which districts get aid), it removes any ambiguity. The included example lets programmers test their code quickly to see if it works as intended. Without this clarity, different developers might interpret the problem differently, leading to inconsistent or buggy results.

The constraints and limits listed in the document serve as guidelines. They prevent the algorithm from being used in ways it wasn't designed for, for example, trying to handle a budget of zero or too many districts. The computational limits, like time and space complexity, also ensure the solution runs efficiently, even for larger problems. Without these boundaries, the program might become too slow or crash when faced with real-world data.

Finally, the fixed urgency weights and documented assumptions make the system fair and predictable. Everyone can agree that "Critical" districts always get priority because the rules are clear upfront. The assumptions also explain hidden design choices, like why the budget can't change mid-calculation. This transparency helps avoid arguments later and ensures the system behaves the same way every time it runs.

7.0 Coding

We built the proof of concept of our Food Bank Allocation Algorithm using Java.

7.1 Java Code Structure

The Java project consists of:

- **Main.java**: Entry point; reads JSON input and runs the algorithm
- **District.java**: Represents district data and computes weightedValue
- **FoodBankAllocator.java**: Contains the dynamic programming algorithm
- **TestRunner.java**: Executes all test scenarios from the resources folder

7.2 Key Functions and Their Purpose

```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());
        }
    }
}
```

Figure 2: Main.java

main(String[] args)

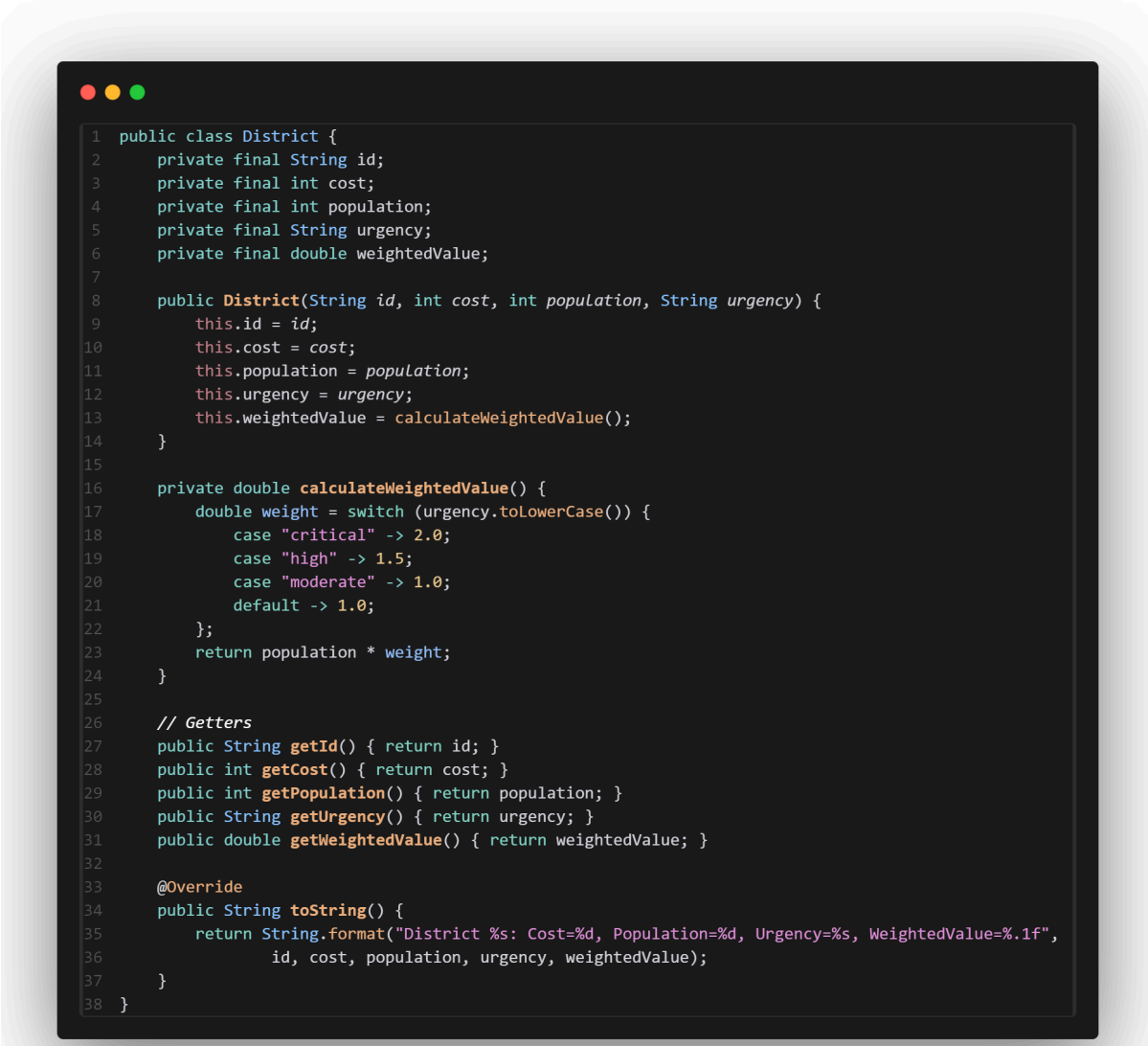
- Purpose: Application entry point that processes command-line arguments and executes the algorithm
- Input Validation: Checks for file existence and valid arguments
- Flow Control: Orchestrates JSON parsing, algorithm execution, and result display

readFile(String filename)

- Purpose: Reads file content using Java NIO Files API
- Error Handling: Returns null and prints error message if file cannot be read
- Implementation: Uses Files.readAllBytes(Paths.get(filename)) for efficient file reading

parseJSON(String json, int[] budget, List<District> districts)

- Purpose: Parses JSON input files containing budget and district data
- Implementation: Uses string manipulation to extract budget and district information
- Parsing Logic:
    - Removes whitespace from JSON string
    - Extracts budget using string splitting on "budget":
    - Extracts districts array and parses each district object
    - Creates District objects with parsed data



```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();
    }

    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;
    }

    // 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);
    }
}
```

Figure 3: District.java

calculateWeightedValue()

- Purpose: Converts urgency levels to priority weights and calculates weighted population value
- Weight Mapping (using switch expression):
    - Critical: 2.0x multiplier
    - High: 1.5x multiplier
    - Moderate: 1.0x multiplier
    - Default: 1.0x multiplier (for unknown urgency levels)
    - Formula: weightedValue = population × urgency_weight

toString()

- Purpose: Provides formatted string representation of district data
- Format: "District {id}: Cost={cost}, Population={population}, Urgency={urgency}, WeightedValue={weightedValue}"

```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;
        }
    }
}
```

Figure 4: FoodBankAllocator.java

allocateFoodBanks()

- Purpose: Executes the dynamic programming solution for the 0/1 knapsack problem
- Algorithm Steps:

    - Initialize DP table with dimensions (n+1) × (budget+1)
    - Fill DP table using recurrence relation
    - Backtrack to identify selected districts

Backtracking Logic (within allocateFoodBanks())

- Purpose: Identifies which districts were selected in the optimal solution
- Method: Traces backward through DP table from dp[n][budget]
- Criterion: District i is selected if dp[i][w] > dp[i-1][w]
- Implementation: Decrements budget (w) by selected district cost

AllocationResult Inner Class

- Purpose: Encapsulates algorithm results
- Fields:
- maxValue (double): Maximum weighted value achieved
- selected (List<District>): List of selected districts

```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];
    }
}
```

Figure 5: TestRunner.java

main(String[] args)

- Purpose: Executes all test cases automatically
- Flow: Discovers JSON files, runs each through Main.main(), displays results

getAvailableTestCases()

- Purpose: Automatically discovers all JSON test files in resources directory
- Filter: Only processes files with .json extension
- Sorting: Returns files in alphabetical order for consistent execution
- Error Handling: Returns empty array if resources directory doesn't exist

8.0 Demonstration

8.1 Sample Input and Output

This section demonstrates the functionality of the proposed Food Bank Allocation program using a sample test case.

Input:

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 program will then select districts in a manner that maximises total weighted value without exceeding the budget constraint. The sample output is presented below:

```
===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
```

Figure 6: Sample Output

The result shows that the algorithm successfully selected Districts A, B, and D, achieving a combined cost of exactly 5 units and a total weighted value of 445.0. This demonstrates the algorithm's ability to:

- Prioritise high-impact districts (B and A were the top-weighted).
- Optimise aid distribution within tight budget constraints.
- Ensure fairness and strategic selection, taking into account both population and urgency.

This output reflects the algorithm’s ability to prioritise districts with high urgency and large populations, respect budget limitations, and make optimal selections under the constraints.

9.0 Results and Evaluation

9.1 Algorithm Efficiency

The primary objective of the algorithm is to ensure that limited resources are allocated in a way that maximises social impact. The efficiency is measured by maximising population impact and urgency within a constrained budget. In the tested scenario, the algorithm successfully identified the most cost-effective districts based on urgency and population.

Table 2: Allocation Summary Metrics for Sample Test Case

Metric	Value
Budget Limit	5 units
Total Cost Utilised	5 units
Total Weighted Value Achieved	445.0

This outcome indicates high efficiency in resource usage. The algorithm strategically avoided low-impact or high-cost districts and instead selected the most value-dense option(s). Despite the budget not being fully consumed, the result demonstrates optimal decision-making with prioritising impact over full utilisation.

Such behaviour reinforces the strength of the algorithm, especially in real-world situations where over-allocation could be detrimental and strategic underspending may yield higher social benefit.

9.2 Algorithm Correctness

To ensure the correctness and reliability of the implemented dynamic programming algorithm, we employed two complementary 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

Initialisation:
The algorithm initialises a two-dimensional table dp[i][w], where i represents the number of districts considered (from 0 to n), and w corresponds to the remaining budget (from 0 to W).

Initially, all entries are set to 0 with dp[0][w]=0 for all $w \in [0,W]$ because this reflects the base case for the recurrence. When no districts are selected, the total value achieved is zero, regardless of the budget.

Loop Invariant:
At the beginning of each iteration of the outer loop (for district i) and the inner loop (for budget w), 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:
During each iteration, the algorithm evaluates two possibilities for each district i:

- **Exclusion of district *i*:** The value is inherited from dp[i−1][w], indicating the best achievable value without including the current district.
- **Inclusion of district *i*:** If the cost of district *i* does not exceed the current budget w, then the value becomes dp[i−1][w−$cost_i$] + $value_i$.

The recurrence relation is thus defined as:

$$dp[i][w] = \begin{cases} max(dp[i-1][w],\ dp[i-1][w-cost_i] + value_i), & if\ cost_i \leq w \\ dp[i-1][w], & otherwise \end{cases}$$

This ensures that the algorithm selects the optimal decision whether to include or exclude the current district at each subproblem without violating the budget constraint.

Termination:
After completing all iterations, the entry dp[i][w] contains the maximum total value that can be obtained without exceeding the budget. A backtracking process through the table can then identify the subset of districts contributing to this optimal value.

Hence, by satisfying the four conditions of the loop invariant method (Initialisation, Invariant, Maintenance, and Termination), the algorithm is provably correct.

Empirical Validation: Input-Output Testing

In addition to the formal correctness proof via the loop invariant method, empirical validation was carried out to demonstrate the algorithm's practical reliability under various input conditions. A series of structured test scenarios was designed to evaluate the correctness of both the computed optimal value and the corresponding subset of selected districts.

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.

The results of each test are summarised in **Table 3**.

Table 3: Summary of Empirical Test Cases and Observed Results

Case	Description	Expected Behavior	Observed Result
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.	Maximum Value: 445.0 Selected Districts: A (150.0), B (160.0), D (135.0) Selection and value matched expectations.
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.	Maximum Value: 0.0 Selected Districts: None The output matched the expected result.
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.	Maximum Value: 572.5 Selected Districts: A, B, C, D, E All districts were selected, as expected.
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.	Maximum Value: 237.5 Selected Districts: A (140.0), C (97.5) Correct prioritisation observed.

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, optimality, and robustness in a range of realistic disaster-relief allocation scenarios.

9.3 Complexity Analysis and Growth Function

The proposed algorithm is based on the classical 0/1 Knapsack dynamic programming (DP) approach. In this context, the problem domain involves selecting a subset of districts within a limited budget to maximise overall value. While analysing, consider n = number of districts, and W = budget.

Time Complexity Analysis

The algorithm employs a bottom-up dynamic programming approach using a two-dimensional table of size $(n + 1) \times (W + 1)$. The time complexity analysis is shown in **Table 4**.

Table 4: Time Complexity Analysis

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), as every subproblem must be solved regardless of the data distribution or values. The approach is deterministic and exhaustive within the defined state space.

Space Complexity Analysis

The current implementation maintains a complete 2D table of dimensions (n + 1) × (W + 1), resulting in a space complexity of:

$$\text{Space Complexity} = O(nW)$$

This allocation allows the algorithm to store the computed maximum value for each subproblem and supports a backtracking step to identify the contributing districts. While effective for small to moderate input sizes, this approach becomes memory-intensive for large datasets.

Growth Function

The algorithm's growth function is polynomial in both input parameters n and W, and can be formally expressed as:

$$T(n, W) \in \theta(nW)T(n, W)$$

This indicates that the runtime increases proportionally with the number of districts and the size of the budget. The algorithm thus scales efficiently for small to moderate problem sizes, making it suitable for real-world decision-making scenarios where constraints are manageable. However, for significantly larger instances (e.g., thousands of districts or very high budget values), performance optimisation strategies, such as space reduction, heuristic approximation, or problem decomposition, may be required to maintain acceptable computational efficiency.

10.0 Summary and Reflection

In this project, we successfully designed and implemented a decision-support model for priority-based food bank allocation during disaster relief using the 0/1 Knapsack Dynamic Programming algorithm. We model the solution to reflect real-world constraints, including limited delivery budgets, urgency levels, and binary (all-or-nothing) aid policies. Through a well-structured algorithmic approach, we ensured that the limited resources were allocated to the most impactful combination of districts, which maximises the total priority-weighted population served without exceeding the available budget.

We tested our implementation using structured input scenarios that reflect real-world disaster relief conditions. The program was validated using multiple sets of district-level data with varying costs, populations, and urgency levels. These tests confirmed that the dynamic programming approach consistently produced optimal results within the defined budget constraints. The result clearly shows the algorithm has the ability to make precise, high-impact decisions while honouring critical limitations and ensuring practical applicability for humanitarian resource allocation.

From a reflective perspective, this project has not only deepened our technical knowledge of algorithm design and optimisation but also enhanced our problem-solving and critical thinking skills. We learned how to translate the theoretical concepts from our lectures into real-world applications with life-impacting consequences. While we acknowledge that real-world situations are often more complex, involving additional constraints and inconsistencies, we have successfully demonstrated a proof of concept. Each of us contributed to the project, whether in research, development, testing, or documentation, and this collaboration enabled us to combine our individual strengths toward a unified goal. Throughout the process, we also learned the value of clear communication, division of responsibilities, and continuous iteration.

11.0 References

AbuSalim, T., Ayyash, M., & Mismar, F. (2020). An efficient Dijkstra's algorithm for finding the shortest path in real road networks. *International Journal of Advanced Computer Science and Applications (IJACSA), 11*(7), 263–269. https://doi.org/10.14569/IJACSA.2020.0110734

Alkaabneh, F., Diabat, A., & Gao, H. O. (2020). A unified framework for efficient, effective, and fair resource allocation by food banks using an Approximate Dynamic Programming approach. *Omega*, *100*, 102300. https://doi.org/10.1016/j.omega.2020.102300

Alsalmi, A. A. (2020). A comparative analysis of searching algorithms. Journal of University Studies for inclusive Research, 1(1), 1-6. https://usrij.com/wp-content/uploads/2020/04/A-comparative-analysis-of-searching-algorithms.pdf

Deng, J., Yang, R., Liang, J., Wu, Y., & Wang, H. (2024). Research on models based on greedy algorithms, dynamic programming algorithms and grid discretization algorithms. *2024 IEEE 2nd International Conference on Control, Electronics and Computer Technology (ICCECT)*, 847–852. https://doi.org/10.1109/iccect60629.2024.10545782

Luna-Garcia, R., Candano, T. M., & Caga-Anan, R. (2024). Performance analysis of classical algorithms for the traveling Salesman Problem. *journals.msuiit.edu.ph*. https://journals.msuiit.edu.ph/tmjm/article/view/716/599

Posypkin, M., & Thant Sin, S. T. (2016). Comparative analysis of the efficiency of various dynamic programming algorithms for the Knapsack problem. *2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW)*, 313–316. https://doi.org/10.1109/eiconrusnw.2016.7448182

Posypkin, M., & Thant Sin, S. T. (2020). Comparative performance study of shared and distributed memory dynamic programming algorithms. *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, 2010–2013. https://doi.org/10.1109/eiconrus49466.2020.9039470

Qu, X., Zhang, R., & Han, J. (2024). Flight attendant route planning and resource allocation algorithm based on reinforcement learning and Dynamic Programming. *2024 3rd International Conference on Artificial Intelligence and Autonomous Robot Systems (AIARS)*, 449–454. https://doi.org/10.1109/aiars63200.2024.00089

Ren, X., Guo, W., Mo, Z., & Tian, W. (2018). A divide and conquer approach to all solutions satisfiability problem. *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, 2590–2595. https://doi.org/10.1109/compcomm.2018.8780746

12.0 Appendix

Initial Project Plan

<table>
<tr><td>Group Name</td><td>Group 6</td></tr>
<tr><td>Members</td><td>
<table>
<tr><td>Name</td><td>Email</td><td>Phone number</td></tr>
<tr><td>CHAN CI EN</td><td>215035@student.upm.edu.my</td><td>010-2939109</td></tr>
<tr><td>KHOO BOO JING</td><td>215382@student.upm.edu.my</td><td>012-4290341</td></tr>
<tr><td>LOO HUAI YUAN</td><td>215516@student.upm.edu.my</td><td>011-10758597</td></tr>
<tr><td>TAN YONG JIN</td><td>217086@student.upm.edu.my</td><td>013-9190343</td></tr>
</table>
</td></tr>
<tr><td>Problem scenario description</td><td>During times of economic crisis or natural disaster, food banks face the challenge of distributing limited aid to districts with varying levels of urgency, population, and delivery cost. Each district has different needs, and aid cannot be partially distributed. it is either delivered in full or skipped. The goal is to distribute available food resources to the most impactful combination of districts.</td></tr>
<tr><td>Why it is important</td><td>Efficient and fair food distribution is critical during crisis situations to avoid neglecting high-need communities and ensure that available resources are used in the most impactful way. Poor allocation can result in wasted resources or missed opportunities to help the most vulnerable groups.</td></tr>
<tr><td>Problem specification</td><td>Each district has:
● A cost (e.g., delivery cost or amount of food needed)
● A population size
● An urgency level (Moderate, High, Critical)
A weighted value is calculated from population × urgency weight.
Total available budget is fixed.
The problem is modeled as a 0/1 Knapsack problem.</td></tr>
<tr><td>Potential solutions</td><td>Dynamic Programming (0/1 Knapsack) to find the optimal combination of districts. Sorting and greedy approaches may be used for comparison or initial approximation, but are not reliable for fairness or optimality.</td></tr>
<tr><td>Sketch (framework, flow, interface)</td><td>Framework:
● Input from json file (cost, population, urgency)
● Calculate weighted value per district
● Apply 0/1 Knapsack DP algorithm
● Output selected districts and total value
Flow:
json → Value calculation → Dynamic Programming → Output report

Interface:
Command-line Java program reading json input and printing selected districts with total value</td></tr>
</table>

Project Proposal Refinement

<table>
<tr><td>Group Name</td><td>Group 6</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Role</th></tr>
<tr><td>Chan Ci En</td><td>Team Leader & Developer</td></tr>
<tr><td>Khoo Boo Jing</td><td>Tester</td></tr>
<tr><td>Loo Huai Yuan</td><td>Researcher</td></tr>
<tr><td>Tan Yong Jin</td><td>Algorithm Designer</td></tr>
</table>
</td></tr>
<tr><td>Problem statement</td><td>How can we allocate limited food bank resources to the most high-priority districts during a crisis, ensuring that total impact is maximized while staying within delivery budget constraints?</td></tr>
<tr><td>Objectives</td><td>● Design and implement an algorithm to maximize impact under budget constraints
● Ensure fairness through urgency-weighted prioritization
● Provide a system that can support data input and produce reasonable decisions</td></tr>
<tr><td>Expected output</td><td>● A working program that reads district-level data and selects the best set of districts to support
● Online portfolio with algorithm explanation, analysis, and demonstration</td></tr>
<tr><td>Problem scenario description</td><td>A food bank must distribute limited resources to districts during a disaster. Each district has a different level of urgency, population, and resource cost. Aid cannot be split—districts are either fully supported or not at all. The goal is to choose the combination of districts that maximizes total weighted priority value without exceeding the overall budget.</td></tr>
<tr><td>Why it is important</td><td>Efficient and fair food distribution is critical during crisis situations to avoid neglecting high-need communities and ensure that available resources are used in the most impactful way. Poor allocation can result in wasted resources or missed opportunities to help the most vulnerable groups.</td></tr>
<tr><td>Problem specification</td><td>Each district has:

● A cost (e.g., delivery cost or amount of food needed)
● A population size
● An urgency level (Moderate, High, Critical)

A weighted value is calculated from population × urgency weight.
Total available budget is fixed.
The problem is modeled as a 0/1 Knapsack problem.</td></tr>
<tr><td>Potential solutions</td><td>Dynamic Programming (0/1 Knapsack) to find the optimal combination of districts.</td></tr>
</table>

<table>
<tr><td></td><td>Sorting and greedy approaches may be used for comparison or initial approximation, but are not reliable for fairness or optimality.</td></tr>
<tr><td>Sketch (framework, flow, interface)</td><td>Framework:

● Input from json file (cost, population, urgency
● Calculate weighted value per district
● Apply 0/1 Knapsack DP algorithm
● Output selected districts and total value

Flow:
json → Value calculation → Dynamic Programming → Output report

Interface:
Command-line Java program reading json input and printing selected districts with total value</td></tr>
<tr><td>Methodology</td><td>
<table>
<tr><td>Milestone</td><td>Time</td></tr>
<tr><td>Finalized the problem scenario and budget scope, reviewed related literature, and selected 0/1 Knapsack as the algorithm approach.</td><td>Week10</td></tr>
<tr><td>Completed key documentation sections including model development, algorithm suitability review, and pseudocode structure.</td><td>Week11</td></tr>
<tr><td>Developed and tested Java implementation, documented demonstration, finalized algorithm flow and I/O, and compiled supporting appendices.</td><td>Week12</td></tr>
<tr><td>Finalized documentation and proofreading, completed results analysis and summary, built the GitHub portfolio site, designed presentation slides, and submitted all required materials.</td><td>Week 13-14</td></tr>
</table>
</td></tr>
</table>

Project Progress

Project Progress (Week 10 – Week 14)

<table>
<tr><td>Milestone 1</td><td>Finalized the problem scenario and budget scope, reviewed related literature, and selected 0/1 Knapsack as the algorithm approach.</td></tr>
<tr><td>Date (week)</td><td>Week 10</td></tr>
<tr><td>Description/ sketch</td><td>Task: Scenario finalization and algorithm selection.

Details: Our team finalized the disaster relief food allocation scenario based on real-world constraints and urgency levels. Potential algorithms (Greedy and 0/1 Knapsack) were discussed and evaluated. Dynamic Programming using 0/1 Knapsack was chosen for its optimality under budget constraints.</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Member 1 (Chan Ci En)</td><td>Member 2 (Khoo Boo Jing)</td><td>Member 3 (Loo Huai Yuan)</td><td>Member 4 (Tan Yong Jin)</td></tr>
<tr><td>Defined and finalized the overall problem scenario and delivery budget scope.</td><td>Drafted the scenario background and documented logistical needs.</td><td>Conducted literature review on food distribution models in disaster response.</td><td>Evaluated Greedy vs DP and justified 0/1 Knapsack with priority weights.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>Milestone 2</td><td>Completed key documentation sections including model development, algorithm suitability review, and pseudocode structure.</td></tr>
<tr><td>Date (Wk)</td><td>Week 11</td></tr>
<tr><td>Description/ sketch</td><td>Task: Initial documentation and algorithm structure.

Details: Our team developed key documentation sections for the report and portfolio, including problem overview, algorithm suitability, and pseudocode. Work was distributed to ensure technical and written progress.</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Member 1 (Chan Ci En)</td><td>Member 2 (Khoo Boo Jing)</td><td>Member 3 (Loo Huai Yuan)</td><td>Member 4 (Tan Yong Jin)</td></tr>
<tr><td>Drafted Section 4.0: Model Development of the Scenario.</td><td>Completed Sections 1.0, 2.0, and prepared structure for testing documentation.</td><td>Wrote Section 3.0: Algorithm Suitability Review.</td><td>Created Section 5.2 pseudocode and sketched algorithm structure.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>Milestone 3</td><td>Developed and tested Java implementation, documented demonstration, finalized algorithm flow and I/O, and compiled supporting appendices.</td></tr>
<tr><td>Date (week)</td><td>Week 12</td></tr>
<tr><td>Description/ sketch</td><td>Task: Algorithm specification, coding, and early testing.

Details: Our team implemented the Java-based DP algorithm, completed algorithm specification, and ran multiple test cases. Appendices and technical documentation were also produced.</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Member 1 (Chan Ci En)</td><td>Member 2 (Khoo Boo Jing)</td><td>Member 3 (Loo Huai Yuan)</td><td>Member 4 (Tan Yong Jin)</td></tr>
<tr><td>Developed Java code (Section 7.0) and validated algorithm with test inputs.</td><td>Wrote Section 8.0 Demonstration and assisted testing workflow.</td><td>Compiled appendices: Initial Plan & Proposal Refinement.</td><td>Completed Sections 5.0, 6.0 and finalized algorithm flowchart and I/O format.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>Milestone 4</td><td>Finalized documentation and proofreading, completed results analysis and summary, built the GitHub portfolio site, designed presentation slides, and submitted all required materials.</td></tr>
<tr><td>Date (Wk)</td><td>Week 13 - 14</td></tr>
<tr><td>Description/ sketch</td><td>Task: Analysis, online portfolio, and final preparation.

Details: Our team conducted result evaluation, built a portfolio website, finalized the report and slide deck, and submitted all deliverables. The solution's optimality, complexity, and impact were highlighted in the final presentation.</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Member 1 (Chan Ci En)</td><td>Member 2 (Khoo Boo Jing)</td><td>Member 3 (Loo Huai Yuan)</td><td>Member 4 (Tan Yong Jin)</td></tr>
<tr><td>Finalized compilation and uploaded files to PutraBlast.</td><td>Analyzed Section 9.0: Results and handled test case explanation.</td><td>Wrote Section 10.0 Summary and conducted final proofread of all content.</td><td>Built a GitHub Site portfolio and designed final presentation slides.</td></tr>
</table>
</td></tr>
</table>