FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
DEPARTMENT OF COMMUNICATION TECHNOLOGY AND NETWORK

GROUP PROJECT

SECOND SEMESTER 2023/2024

COURSE NAME : DESIGN AND ANALYSIS OF ALGORITHMS

COURSE CODE : CSC4202-5

MEMBERS :

1. **CHENG KE XI** **212228**
2. **THANESH RAO A/L SIMMATHIRI** **210887**
3. **WU LIQIANG** **209759**

LECTURER : DR NUR ARZILAWATI BINTI MD YUNUS

Table of Content

Original Scenario: Efficient Cargo Loading for Humanitarian Aid Delivery

Context

Consider a humanitarian aid organization responsible for dispatching relief shipments to areas affected by natural disasters such as earthquakes, floods, and hurricanes. The mission is to deliver essential supplies quickly and efficiently to mitigate suffering and save lives.

One of the key assets available is a cargo plane, which plays a crucial role in transporting aid to remote and inaccessible regions. However, the plane has a maximum weight capacity that it can safely carry, which must be adhered to in order to ensure safe and efficient operations.

Problem Description

Each piece of aid cargo has a specific weight and an associated priority value. The priority value represents the urgency and importance of delivering that particular item, as certain supplies may be more critical for immediate survival and relief than others. The goal is to load the plane in such a way that maximizes the total priority value of the cargo, ensuring that the plane does not exceed its maximum weight capacity.

Weight Capacity

The maximum weight that the cargo plane can carry without exceeding its operational limits is 4000 kilograms.

Cargo Items: A diverse list of aid items is available, each with its own weight and priority value. These items represent essential supplies like food, water, medical kits, blankets, and tents, which are crucial for disaster relief efforts.

Here is an example of the aid items :

Item	Weight (kg)	Priority Value
Non-perishable food	700	60
Bottled Water	400	50
First Aid Kit	300	70
Insulated Blankets	500	40
Emergency Shelter	800	80
Baby Formula and Food	200	55
Personal Hygiene Items	150	65
Water Purification System	300	75
Portable Toilet	250	70
Medical Supplies	350	65
Communication Devices	200	60
Fire-starting Supplies	100	45
Tools	400	40
Sanitation Supplies	200	55
Pet Supplies	300	50
Entertainment and Comfort Items	150	45
Safety Gear	250	60

Importance of finding an optimal solution

1. **Maximizing Aid Effectiveness**

Ensuring that the highest priority items are delivered first maximizes the effectiveness of the aid provided. In disaster scenarios, high-priority items like medical kits and food are crucial for immediate relief and can literally save lives. Medical kits can treat injuries and prevent infections, which are common in disaster-affected areas. Providing food prevents starvation and malnutrition, which can exacerbate health problems among the affected population. Prioritizing essential items helps address the most urgent needs of survivors, reducing their suffering significantly.

The prompt delivery of life-saving supplies can stabilize critical health conditions, preventing fatalities. Effective aid distribution ensures that resources are used where they are most needed, making the overall response more efficient. It also enables aid workers to plan subsequent deliveries better, knowing that the immediate critical needs have been met. By focusing on the most critical needs first, the aid organization can create a positive impact quickly, which is vital in the chaotic aftermath of a disaster. Maximizing aid effectiveness ultimately translates into a more humane and responsive relief effort, underscoring the importance of prioritizing high-importance items.

2. **Efficient Resource Utilization**

An optimal solution ensures the plane's weight capacity is used most efficiently, avoiding wastage of valuable space and weight that could have been used for critical supplies. In the context of humanitarian aid, every kilogram of capacity on a plane is precious. By utilizing the plane's full capacity, the organization can maximize the amount of aid delivered in a single trip. This efficiency is crucial because it allows more supplies to reach those in need faster, especially when transportation resources are limited. Avoiding underutilization ensures that no potential aid is left behind due to poor planning. This is particularly important in remote or hard-to-reach areas where supply deliveries are infrequent.

Efficient loading also means that items that might not seem critical on their own but collectively make a significant difference, such as hygiene products or clothing, can be included. Moreover, optimizing space and weight can help in balancing the load, which is crucial for the safety of the flight. Efficient resource utilization is not just about filling the plane but doing so in a way that maximizes the overall impact of the relief mission. Thus, it contributes to a more effective and comprehensive aid response.

3. **Operational Cost Management**

Loading the plane optimally reduces the number of trips needed, saving fuel and operational costs, which is especially important for non-profit organizations with limited budgets. Humanitarian organizations often operate under tight financial constraints and rely heavily on donations and grants. Reducing the number of trips means that resources spent on fuel, maintenance, and personnel can be significantly

lowered. Each flight incurs substantial costs, so minimizing the number of flights can lead to considerable savings.

These savings can then be redirected to other critical areas such as purchasing more supplies, hiring additional aid workers, or improving logistics on the ground. Fewer trips also mean less wear and tear on aircraft, reducing maintenance costs and prolonging the lifespan of the planes. Additionally, minimizing operational costs allows the organization to respond more quickly and flexibly to emerging needs. Efficient cost management through optimal loading ensures that the organization can maintain its operations sustainably over the long term. This financial efficiency is vital for maintaining the trust and support of donors, who expect their contributions to be used wisely. Therefore, managing operational costs effectively is a key component of a successful humanitarian aid strategy.

4. Timely Delivery

Optimal loading ensures that the most important items reach the disaster-stricken area promptly, reducing suffering and helping stabilize the situation more quickly. In the immediate aftermath of a disaster, time is of the essence. Quick delivery of essential supplies like medical kits, food, and water can prevent further casualties and reduce the overall impact of the disaster. By prioritizing and efficiently loading high-importance items, aid organizations can ensure that the most critical needs are met as soon as possible. This prompt response can help prevent the outbreak of diseases, reduce hunger and dehydration, and provide shelter to those who have lost their homes.

Timely delivery also helps in stabilizing the situation by restoring some degree of normalcy and security for the affected population. It allows aid workers to set up operations and distribute supplies more effectively. Furthermore, quick and efficient aid delivery can help in rebuilding trust and morale among the survivors, showing them that they are not alone and that help is on the way. This immediate relief can significantly reduce panic and despair, providing a foundation for longer-term recovery efforts. Therefore, ensuring timely delivery is a crucial aspect of disaster response that can save lives and alleviate suffering.

5. Transparency and Accountability

Demonstrating that aid is distributed efficiently helps maintain the trust of donors and stakeholders, ensuring continued support and funding for humanitarian efforts. Transparency in how aid is distributed reassures donors that their contributions are being used effectively and for their intended purpose. Accountability in operations shows that the organization is responsible and trustworthy, which is essential for maintaining and growing donor support. Clear and efficient aid distribution practices help in providing accurate reports and audits, demonstrating the organization's commitment to ethical standards. This transparency can also attract new donors who are looking for reliable and effective channels to contribute to.

Moreover, stakeholders, including governments and partner organizations, need to see that the aid is being used efficiently to justify their support and collaboration. Efficient distribution also means that the organization can respond to

donor inquiries and media scrutiny with confidence, showcasing their operational effectiveness. Maintaining a high level of transparency and accountability helps build a positive reputation, which is crucial for long-term sustainability. This trust is essential not only for funding but also for mobilizing volunteers and securing partnerships. Thus, transparency and accountability are fundamental to the ongoing success and impact of humanitarian aid efforts.

Review of Solution Paradigms

1. **Sorting**

   Strengths: The sorting approach is simple to implement and understand, making it accessible even to those without extensive programming knowledge. It can quickly prioritize items based on their value-to-weight ratio, providing a heuristic solution that is easy to comprehend.

   Weaknesses: Despite its simplicity, sorting does not guarantee an optimal solution. This method may overlook high-priority items if they do not fit well with the remaining capacity after sorting, leading to suboptimal outcomes in some cases.

2. **Divide and Conquer (DAC)**

   Strengths: DAC can effectively break down complex problems into smaller, more manageable subproblems. This approach can simplify the overall solution process and make it easier to tackle challenging scenarios.

   Weaknesses: However, DAC may not be well-suited for the cargo loading problem as it does not inherently provide a way to combine solutions from subproblems into an optimal global solution. This limitation could hinder its effectiveness in this context.

3. **Dynamic Programming (DP)**

   Strengths: DP guarantees an optimal solution by considering all possible combinations of items up to the weight limit. By using memoization, DP avoids redundant calculations, making it efficient in finding the best solution.

   Weaknesses: However, DP can be memory-intensive and complex to implement, particularly when dealing with a large number of items. The need to store and update values for every combination can lead to increased computational overhead.

4. **Greedy Algorithm**

   Strengths: The greedy algorithm is fast and easy to implement, making it a practical choice for simple scenarios. It works well when items are sorted by their value-to-weight ratio, as it consistently selects the most immediately beneficial option at each step.

   Weaknesses: However, the greedy algorithm does not always produce an optimal solution. This is because it makes decisions based on the current best choice without

considering the potential impact on future selections. As a result, it may overlook better overall solutions that require sacrificing immediate gains.

5. **Graph Algorithms**

   Strengths: Graph algorithms can represent complex problems like the cargo loading scenario in a structured and visual way. They can find paths that optimize both weight and value, providing a comprehensive approach to the problem.

   Weaknesses: However, graph algorithms are generally more complex and may not be as intuitive for solving knapsack-type problems. Implementing and understanding these algorithms may require more expertise and computational resources compared to simpler approaches.

Summary of Solution Paradigm

Paradigm	Strength	Weakness
Sorting	Simple implementation and understanding.	Does not guarantee optimal solution. May overlook high-priority items.
Divide and Conquer (DAC)	Breaks down complex problems into manageable subproblems.	Doesn't naturally combine subproblems into an optimal global solution. Limited effectiveness for cargo loading.
Dynamic Programming (DP)	Guarantees optimal solution by considering all combinations. Avoids redundant calculations through memoization.	Memory-intensive and complex. Increased computational overhead.
Greedy Algorithm	Fast and easy to implement. Works well for sorted items by value-to-weight ratio.	Doesn't always produce optimal solution. May overlook better solutions by focusing on immediate gains.
Graph Algorithm	Represents problems visually. Can find paths optimizing both weight and value.	More complex and less intuitive for knapsack-type problems. Requires more expertise and computational resources.

Algorithm Design

- **Dynamic Programming**
  - The algorithm utilizes dynamic programming to efficiently solve the problem by breaking it down into smaller subproblems and solving them recursively.
  - It constructs a 2D array dp[ ][ ] where dp [i] [w] represents the maximum priority value that can be achieved considering the first i items and a weight limit of w.

- **Recurrence Relation**
  - At each step of the dynamic programming process, the algorithm considers whether to include the current cargo item or exclude it based on its weight and priority value.
  - If the weight of the current item is less than or equal to the current weight capacity, the algorithm computes the maximum priority value achievable by either including or excluding the item:
    - If included: dp [i] [w] = dp [i - 1] [w - item.weight] + item.priorityValue
    - If excluded: dp [i] [w] = dp [i - 1] [w]
  - If the weight of the current item exceeds the current weight capacity, the item cannot be included, so the value remains the same as excluding the item: dp [i] [w] = dp [i - 1] [w]

- **Optimization**
  - The algorithm optimizes by computing and storing the maximum priority value for each subproblem in the dp array.
  - By storing these values, the algorithm avoids recalculating solutions for the same subproblems multiple times, leading to improved efficiency.

Program Java language

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class CargoItem {
    String name;
    int weight;
    int priorityValue;
    int index; // Index to keep track of original order

    public CargoItem(String name, int weight, int priorityValue, int index) {
        this.name = name;
        this.weight = weight;
        this.priorityValue = priorityValue;
        this.index = index;
    }

    @Override
    public String toString() {
        return "name=" + name + ", weight=" + weight + ", priorityValue=" + priorityValue;
    }
}

public class CargoLoading {
    public static void main(String[] args) {
        String csvFile = "C:\\Users\\HP\\eclipse24-workspace\\Asg1\\src\\cargoitems.csv"; //
Path to your CSV file
        int maxWeight = 4000; // Maximum weight capacity of the ship
        List<CargoItem> items = loadItemsFromCsv(csvFile);

        Result result = knapsackDynamicProgramming(items, maxWeight);

        System.out.println("Total Priority Value: " + result.totalPriorityValue);
        System.out.println("Total Weight: " + result.totalWeight);
        System.out.println("Items Chosen:");

        // Sort the itemsChosen list by priority value in descending order
        Collections.sort(result.itemsChosen, new Comparator<CargoItem>() {
            @Override
            public int compare(CargoItem item1, CargoItem item2) {
                return Integer.compare(item2.priorityValue, item1.priorityValue);
            }
        });

        for (int i = 0; i < result.itemsChosen.size(); i++) {
```

```
            CargoItem item = result.itemsChosen.get(i);
            System.out.println((i + 1) + ". Name: " + item.name + ", Priority Value: " +
item.priorityValue + ", Weight: " + item.weight);
        }
    }

    private static List<CargoItem> loadItemsFromCsv(String csvFile) {
        List<CargoItem> items = new ArrayList<>();
        String line;
        String csvSplitBy = ",";

        try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {
            br.readLine(); // skip header
            int index = 1;
            while ((line = br.readLine()) != null) {
                String[] data = line.split(csvSplitBy);
                String name = data[0];
                int weight = Integer.parseInt(data[1]);
                int priorityValue = Integer.parseInt(data[2]);
                items.add(new CargoItem(name, weight, priorityValue, index++));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Sort items by priority value in descending order
        Collections.sort(items, new Comparator<CargoItem>() {
            @Override
            public int compare(CargoItem item1, CargoItem item2) {
                return Integer.compare(item2.priorityValue, item1.priorityValue);
            }
        });

        return items;
    }

    private static Result knapsackDynamicProgramming(List<CargoItem> items, int
maxWeight) {
        int n = items.size();
        int[][] dp = new int[n + 1][maxWeight + 1];

        for (int i = 1; i <= n; i++) {
            CargoItem item = items.get(i - 1);
            for (int w = 1; w <= maxWeight; w++) {
                if (item.weight <= w) {
                    dp[i][w] = Math.max(dp[i - 1][w], dp[i - 1][w - item.weight] + item.priorityValue);
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }
```

```
        int totalPriorityValue = dp[n][maxWeight];
        int totalWeight = 0;
        List<CargoItem> itemsChosen = new ArrayList<>();

        for (int i = n, w = maxWeight; i > 0; i--) {
            if (dp[i][w] != dp[i - 1][w]) {
                CargoItem item = items.get(i - 1);
                itemsChosen.add(item);
                totalWeight += item.weight;
                w -= item.weight;
            }
        }

        return new Result(totalPriorityValue, totalWeight, itemsChosen);
    }
}

class Result {
    int totalPriorityValue;
    int totalWeight;
    List<CargoItem> itemsChosen;

    public Result(int totalPriorityValue, int totalWeight, List<CargoItem> itemsChosen) {
        this.totalPriorityValue = totalPriorityValue;
        this.totalWeight = totalWeight;
        this.itemsChosen = itemsChosen;
    }
}
```

Algorithm Specification

Algorithm Name: **Cargo Loading with Dynamic Programming**

Problem Statement

Given a set of cargo items, each with a weight and a priority value, and a maximum weight capacity for the cargo ship, the objective is to maximize the total priority value of the items loaded onto the ship without exceeding its weight capacity.

Input :

- cargoItems: A list of cargo items, each containing the following attributes:
    - name: Name of the cargo item (String)
    - weight: Weight of the cargo item in kilograms (Integer)
    - priorityValue: Priority value of the cargo item (Integer)
- maxWeight: Maximum weight capacity of the cargo ship in kilograms (Integer)

Output :

- totalPriorityValue: The maximum total priority value achievable by loading cargo items onto the ship (Integer)
- totalWeight: The total weight of the cargo items loaded onto the ship (Integer)
- itemsChosen: A list of cargo items chosen for loading onto the ship, along with their details (List of CargoItem objects)

Algorithm Steps

1. Load Cargo Items from CSV
    - Read cargo items data from a CSV file.
    - Create CargoItem objects for each item, including its name, weight, priority value, and index.
2. Sort Cargo Items:
    - Sort the cargo items by priority value in descending order.
3. Knapsack Dynamic Programming
    - Initialize a 2D array dp[][] of size (n + 1) x (maxWeight + 1), where n is the number of cargo items.
    - Iterate over each cargo item and weight capacity, filling the dp array according to the knapsack dynamic programming algorithm.
    - Compute the maximum total priority value achievable (totalPriorityValue) and the total weight of the chosen items (totalWeight).
4. Retrieve Items Chosen
    - Trace back through the dp array to identify the cargo items chosen for loading onto the ship.
    - Construct a list itemsChosen containing the chosen cargo items.
5. Output Results
    - Return the totalPriorityValue, totalWeight, and itemsChosen as the output of the algorithm.

Analysis of Algorithm Correctness and Time Complexity

Correctness

The implemented algorithm is based on the dynamic programming (DP) approach for solving the 0/1 Knapsack Problem. The correctness of the algorithm can be verified through Base Case Initialization, Recursive Relation, and Final Solution Construction. The algorithm ensures that the maximum priority value is achieved for the given weight capacity, adhering to the constraints of the 0/1 Knapsack Problem. The principles of dynamic programming, which avoid redundant calculations and ensure optimal substructure, guarantee the correctness of this solution.

1. **Base Case Initialization**
    - The DP table is initialized with zeros, which means that if there are no items or the weight capacity is zero, the total priority value is zero. This base case is correct because, with no items or zero capacity, the maximum achievable priority value is indeed zero.

2. **Recursive Relation**
    - The algorithm correctly implements the recurrence relation for the 0/1 Knapsack Problem. If the current item's weight is less than or equal to the current capacity, the algorithm decides whether to include the item or not by comparing the maximum value obtained from including the item versus excluding it.

      $$dp[i][w] = max(dp[i-1][w], dp[i-1][w - item.weight] + item.priorityValue)$$

    - If the current item's weight exceeds the current capacity, the item is excluded with the algorithm.

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

    - This ensures that for each item and weight capacity, the DP table stores the maximum achievable priority value.

3. **Final Solution Construction**
    - After filling the DP table, the algorithm reconstructs the optimal set of items by tracing back through the table. This backtracking step ensures that the selected items provide the maximum total priority value without exceeding the weight capacity.
    - During backtracking, if $dp[i][w] = dp[i-1][w]$, it indicates that the iii-th item was included in the optimal solution. The item is then added to the chosen items list, and the weight capacity is reduced by the item's weight.

Time Complexity

The time complexity of the dynamic programming solution for the 0/1 Knapsack Problem is analyzed based on three main phases: initialization, filling the DP table, and reconstructing the solution.

1. **Initialization**
    - Initializing the DP table requires creating a 2D array of dimensions $(n + 1)\times(W + 1)$, where n is the number of items and W is the maximum weight capacity. This initialization process takes $O(n\times W)$ time, as each cell in the DP table is set to zero initially.

2. **Filling the DP Table**
    - Filling the DP table involves iterating over each item and each possible weight capacity. For each item i and weight capacity W, the algorithm performs a constant amount of work to decide whether to include the item in the knapsack or not. The decision is based on the maximum priority value that can be achieved:
        - If the item's weight is less than or equal to the current capacity W, the algorithm compares the value of including the item versus not including it.
        - If the item's weight exceeds the current capacity W, the item is excluded, and the value from the previous item is carried forward.
    - Since this operation is performed for each item and each weight capacity, the time complexity for filling the DP table is $O(n\times W)$.

3. **Reconstruction of Solution**
    - After filling the DP table, the algorithm traces back through the table to determine which items were included in the optimal solution. This step involves iterating through the items and checking whether each item was included based on the differences in the DP table values. The tracing process takes $O(n)$ time, as it involves a single pass through the items.

4. **Overall Time Complexity**
    - Combining the three phases, the overall time complexity of the algorithm is:

      $O(n\times W) + O(n\times W) + O(n) = O(2n\times W + n)$
    - Thus, the overall time complexity of the algorithm is $O(n\times W)$.

      $O(2n\times W + n) = O(n\times W)$

Best, Average, and Worst Case Time Complexity

Best Case

Time Complexity: $O(n \times W)$

Even in the best-case scenario, where the problem may seem simpler or the weight capacity W is relatively small, the algorithm still needs to initialize and fill the entire DP table. This initialization involves iterating through all items and weight capacities up to W, resulting in a time complexity of $O(n \times W)$.

Average Case

Time Complexity: $O(n \times W)$

Typically, in the average case, the algorithm explores most of the DP table to find the optimal solution. The dynamic programming approach involves filling the DP table based on subproblems, and this process inherently requires $O(n \times W)$ time, where nnn is the number of items and W is the weight capacity.

Worst Case

Time Complexity: $O(n \times W)$

In the worst-case scenario, where both the number of items nnn and the weight capacity W are large, the algorithm must still fill the entire DP table. This means that every possible subproblem is evaluated, resulting in a time complexity of $O(n \times W)$.

Summary of Time Complexity

The time complexity remains $O(n \times W)$ in all cases because the algorithm always needs to initialize and fill the DP table and perform the same number of operations regardless of the input configuration. The number of operations depends on the total number of items n and the weight capacity W.

Space Complexity

The space complexity is primarily determined by the size of the DP table, which is $O(n \times W)$. This is because the DP table dp requires space proportional to the product of the number of items (n) and the weight capacity (W).

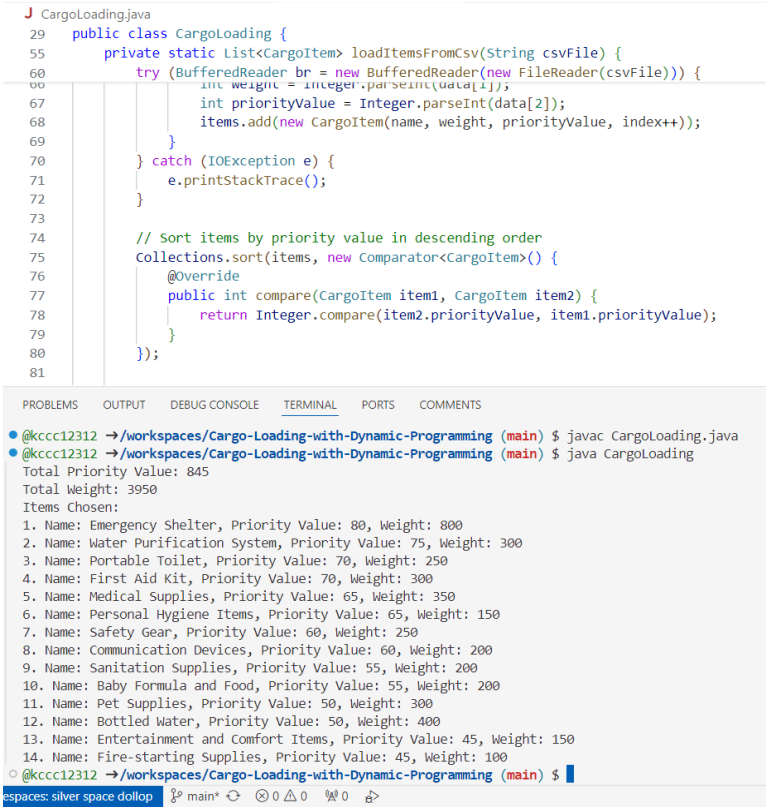
Space Complexity : $O(n \times W)$

The dynamic programming algorithm solves the 0/1 Knapsack Problem efficiently, ensuring an optimal solution by considering all possible item combinations and weights. The time and space complexities of the algorithm are both $O(n \times W)$, making it suitable for the given problem constraints.

GitHub Implementation

Cargo-Loading-with-Dynamic-Programming

https://github.com/kccc12312/Cargo-Loading-with-Dynamic-Programming



Summary

This project successfully addresses the efficient cargo loading problem for humanitarian aid delivery, providing a robust and optimal solution using dynamic programming. The developed algorithm and its implementation ensure that the highest priority items are delivered within the weight constraints of the cargo plane, thereby enhancing the efficiency and impact of the relief efforts.

Initial Project Plan (week 10, submission date: 31 May 2024)

<table>
<tr><td>Group Name</td><td>Relief Rapid</td></tr>
<tr><td>Members</td><td>
<table>
<tr><td>Name</td><td>Email</td><td>Phone number</td></tr>
<tr><td>THANESH RAO A/L SIMMATHIRI</td><td>210887@student.upm.edu.my</td><td>+60 10-392 9754</td></tr>
<tr><td>CHENG KE XI</td><td>212228@student.upm.edu.my</td><td>+60 11-5961 5501</td></tr>
<tr><td>WU LIQIANG</td><td>209759@student.upm.edu.my</td><td>+60 17-619 0155</td></tr>
</table>
</td></tr>
<tr><td>Problem scenario description</td><td>The scenario involves a humanitarian aid organization responsible for delivering relief shipments to disaster-affected areas using a cargo plane. The plane has a maximum weight capacity of 4000 kilograms. Each piece of cargo has a specific weight and priority value. The objective is to maximize the total priority value of the cargo loaded onto the plane without exceeding the weight limit.</td></tr>
<tr><td>Why it is important</td><td>Efficient cargo loading is critical in disaster relief operations to ensure the most urgent and necessary supplies reach affected areas promptly. Properly prioritizing and maximizing the value of the cargo can save lives and improve the effectiveness of the aid mission.</td></tr>
<tr><td>Problem specification</td><td>Maximum Weight Capacity: 4000 kg
Cargo Items: Various aid supplies with specific weights and priority values
Objective: Maximize the total priority value without exceeding the weight limit</td></tr>
<tr><td>Potential solutions</td><td>4. Sorting : Initially sort items based on priority or weight and apply a heuristic method to select items.
5. Divide and Conquer (DAC) :Break the problem into smaller sub-problems and combine their solutions.
6. Dynamic Programming (DP) : Use a DP approach to find the optimal combination of items that maximize the priority value without exceeding the weight capacity.
7. Greedy Algorithm : Select items based on the highest priority value-to-weight ratio until the weight limit is reached.
8. Graph Algorithms : Model the problem as a graph and use algorithms to find the optimal path.</td></tr>
<tr><td>Sketch (framework, flow, interface)</td><td>Framework : Java-based application
Flow:
Input cargo items with weights and priority values
Apply the selected algorithm to determine the optimal cargo load
Output the selected items and their total priority value
Interface: Simple command-line interface for input and output</td></tr>
</table>

Project Proposal Refinement (week 11, submission date: 7 June 2023)

<table>
<tr><td>Group Name</td><td>Relief Rapid</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Role</th></tr>
<tr><td>THANESH RAO A/L SIMMATHIRI</td><td>Algorithm Design and Implementation</td></tr>
<tr><td>CHENG KE XI</td><td>Program Development and Testing</td></tr>
<tr><td>WU LIQIANG</td><td>Analysis and Documentation</td></tr>
</table>
</td></tr>
<tr><td>Problem statement</td><td>Optimize the loading of a humanitarian aid cargo plane to maximize the priority value of the cargo without exceeding a weight limit of 4000 kg.</td></tr>
<tr><td>Objectives</td><td>Design an algorithm to maximize the priority value of the loaded cargo.
Implement the algorithm in Java.
Analyze the correctness and time complexity of the algorithm.
Develop an online portfolio to illustrate the project and its results.</td></tr>
<tr><td>Expected output</td><td>1. An optimized list of cargo items
2. Total priority value of the loaded cargo
3. An online portfolio detailing the problem, solution, and results</td></tr>
<tr><td>Problem scenario description</td><td>The scenario involves a humanitarian aid organization responsible for delivering relief shipments to disaster-affected areas using a cargo plane. The plane has a maximum weight capacity of 4000 kilograms. Each piece of cargo has a specific weight and priority value. The objective is to maximize the total priority value of the cargo loaded onto the plane without exceeding the weight limit.</td></tr>
<tr><td>Why it is important</td><td>a. Maximizing Aid Effectiveness
Efficient cargo loading helps ensure that the most critical supplies, such as medical equipment, food, and water, are prioritized and delivered swiftly to those in need. This prioritization can significantly enhance the overall impact of the aid mission.

b. Efficient Resource Utilization
By optimizing the use of available space and resources, efficient cargo loading allows more supplies to be transported in fewer trips. This not only maximizes the value of each shipment but also ensures that resources are used judiciously.

c. Operational Cost Management
Proper cargo loading can reduce transportation costs by minimizing the number of trips and the amount of fuel required. This cost efficiency allows more funds to be allocated to acquiring and delivering essential supplies.

d. Timely Delivery
Speed is of the essence in disaster relief operations. Efficient cargo loading ensures that aid reaches the affected areas as quickly as possible, reducing delays and helping to stabilize the situation sooner.

e. Transparency and Accountability
An organized and efficient cargo loading process promotes transparency and accountability in aid distribution. It ensures that all stakeholders are aware of what supplies are being sent, how they are prioritized, and</td></tr>
</table>

	when they will arrive, fostering trust and cooperation among all parties involved.
Problem specification	**Maximum Weight Capacity**: 4000 kg **Cargo Items**: Various aid supplies with specific weights and priority values **Objective**: Maximize the total priority value without exceeding the weight limit
Potential solutions	a. Sorting : Initially sort items based on priority or weight and apply a heuristic method to select items. b. Divide and Conquer (DAC) :Break the problem into smaller sub-problems and combine their solutions. c. Dynamic Programming (DP) : Use a DP approach to find the optimal combination of items that maximize the priority value without exceeding the weight capacity. d. Greedy Algorithm : Select items based on the highest priority value-to-weight ratio until the weight limit is reached. e. Graph Algorithms : Model the problem as a graph and use algorithms to find the optimal path.
Sketch (framework, flow, interface)	**Framework** The application will be built using Java, leveraging its robust libraries and tools to handle the various aspects of cargo loading optimization. Java's object-oriented nature and extensive support for algorithms and data structures make it a suitable choice for this application. **Flow** 1. Input Cargo Items ● User Input: The user will provide details of each cargo item, including its weight and priority value. ● Data Structure: Each cargo item will be represented as an object containing these attributes. 2. Algorithm Application ● Selection of Algorithm: The user can choose from different algorithms to determine the optimal cargo load. For instance, a greedy algorithm for a simpler heuristic approach or dynamic programming for an exact solution. ● Processing: The selected algorithm will process the list of cargo items to determine the optimal subset that maximizes the total priority value without exceeding weight limits. 3. Output ● Selected Items: The program will output the list of selected cargo items that form the optimal load. ● Total Priority Value: It will also display the total priority value of the selected items. ● Feedback: Additional information, such as the total weight of the selected items and the remaining capacity, will be provided for better understanding. **Interface** Simple Command-Line Interface (CLI) 1. Prompt for Input a. The application will prompt the user to enter the number of cargo items.

<table>
<tr><td></td><td>
b. For each cargo item, the user will be asked to input the weight and priority value.
2. Algorithm Selection
a. The user will be prompted to select the algorithm they want to use for determining the optimal cargo load.
3. Display Output
a. The application will display the selected cargo items, their total priority value, and any additional relevant information.

Implementation
Step 1: Defining Cargo Item Class
Step 2: Input Handling
Step 3: Algorithm Implementation
Step 4: Integration and Execution
</td></tr>
<tr><td>Methodology</td><td>
<table>
<tr><td>Milestone</td><td>Time</td></tr>
<tr><td>Scenario Refinement</td><td>wk10</td></tr>
<tr><td>Algorithm Selection</td><td>wk11</td></tr>
<tr><td>Code Implementation</td><td>wk12</td></tr>
<tr><td>Correctness and Complexity Analysis</td><td>wk13</td></tr>
<tr><td>Online Portfolio and Presentation</td><td>wk14</td></tr>
</table>
</td></tr>
</table>

Project Progress (Week 10 – Week 14)

Milestone 1	**Scenario Refinement**
Date (week)	**27 May 2024 (Week 10)**
Description/ sketch	Refine the problem scenario and initial problem statement.
Role	

Member 1	Member 2	Member 3
THANESH RAO A/L SIMMATHIRI	CHENG KE XI	WU LIQIANG
Prepare initial sketches and interface design.	Lead the scenario refinement.	Document the problem statement and objectives.

Milestone 2	**Algorithm Selection and Initial Coding**
Date (Wk)	**3 June 2024 (Week 11)**
Description/ sketch	Select the algorithm and begin initial coding.
Role	

Member 1	Member 2	Member 3
THANESH RAO A/L SIMMATHIRI	CHENG KE XI	WU LIQIANG
Begin coding the algorithm.	Assist with coding and document the methodology.	Research and select suitable algorithms.

Milestone 3	**Complete Coding and Debugging**
Date (week)	**10 June 2024 (Week 12)**
Description/ sketch	Finalize the coding and conduct thorough testing and debugging.
Role	

Member 1	Member 2	Member 3
THANESH RAO A/L SIMMATHIRI	CHENG KE XI	WU LIQIANG
Finalize algorithm implementation.	Conduct thorough testing and debugging.	Support testing and start analysis documentation.

<table>
<tr><td>Milestone 4</td><td>Conduct Analysis of Correctness and Time Complexity</td></tr>
<tr><td>Date (week)</td><td>17 June 2024 (Week 13)</td></tr>
<tr><td>Description/ sketch</td><td>Analyze the correctness and time complexity of the algorithm.</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Member 1</td><td>Member 2</td><td>Member 3</td></tr>
<tr><td>THANESH RAO A/L SIMMATHIRI</td><td>CHENG KE XI</td><td>WU LIQIANG</td></tr>
<tr><td>Compile and document analysis results.</td><td>Lead the analysis of correctness.</td><td>Analyze time complexity.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>Milestone 5</td><td>Prepare Online Portfolio and Presentation</td></tr>
<tr><td>Date (week)</td><td>24 June 2024 (Week 14)</td></tr>
<tr><td>Description/ sketch</td><td>Prepare the online portfolio and presentation materials.</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Member 1</td><td>Member 2</td><td>Member 3</td></tr>
<tr><td>THANESH RAO A/L SIMMATHIRI</td><td>CHENG KE XI</td><td>WU LIQIANG</td></tr>
<tr><td>Review and finalize the portfolio and presentation.</td><td>Create the online portfolio.</td><td>Develop the presentation materials.</td></tr>
</table>
</td></tr>
</table>