DESIGN AND ANALYSIS OF ALGORITHMS

CSC4202- 4

SEMESTER 2 2024/2025

GROUP ASSIGNMENT

LECTURER : Dr. Nur Arzilawati Md Yunus

NUM.	STUDENT'S NAME	MATRIC NUM
1.	VISAGAN RAO A/L J VENKATA RAMANA	217138
2.	RAVINRAJ A/L MATHIALAHAN	216807
3.	MAYNON A/L SELVAM	215548
4.	XIAO FENG	213836

Table of Contents

1.0 Introduction

As the modern world continues to grapple with global health challenges like the COVID-19 pandemic, the efficient and equitable distribution of limited resources—especially vaccines—has become one of the greatest challenges to governments and health organizations across the globe. While the situation is dire, the greater question is: how will vaccines be distributed? Because the consequences of little or inefficient distribution may be preventable death, persistent outbreaks, and stretched healthcare infrastructure.

Vaccines are produced in batches and are limited for many reasons during the distribution phase. During the early phases of a pandemic or an outbreak, authorities have to quickly deliver vaccines to disparate areas with differing levels of urgency, needs, and ability to accommodate the many vaccines being delivered.

Meanwhile, 'manual' decision-making or equal distribution (first come, first served or some kind of per-capita distribution system) are frequently uneven and ineffective, and distribution decisions are based on consideration of infection severity, population size, level of efficacy for vaccine(s) in an at-risk population and so on. Therefore, computer- based and data-based decisions through the implementation of the algorithmic optimization of delivery assessments and criteria are critical; from the production of vaccines to demand and eventual illness.

This study investigates the use of Dynamic Programming (DP) as a means to solve a constrained vaccine allocation problem. The problem is formulated as a variant of the 0/1 Knapsack Problem, resulting in a model that ensures:

- Global optimality in lives saved;
- Adequate consideration of logistical constraints such as fixed dose sizes;
- Scalability so decisions can be made in real-time.

We provide a rigorous, mathematically sound solution that can assist policy makers in maximizing public health impact when faced with constrained resources.

2.0 Problem Statement

In a pandemic situation, assume a central government or public health authority is responsible for distributing a limited number of vaccine doses to a set of cities or regions. Each city has its own set of influencing factors, such as:

- A different population size
- A unique infection rate, indicating the proportion of people at risk
- A vaccine efficacy score, which may vary due to factors like demographics, climate, or logistical capacity
- A non-negotiable dose requirement; partial allocation is ineffective or impractical

The goal is to determine which cities should receive vaccine allocations such that:

1. The total number of lives saved is maximized, and
2. The total doses distributed do not exceed the central supply.

The problem becomes even more important when considering:

- Real-time distribution across changing outbreak zones
- Fair distribution among under-resourced regions
- Ensuring no vaccine wastage through incomplete allocations

Therefore, solving this problem with a robust algorithm like Dynamic Programming is both a technical necessity and a moral imperative in public health management.

3.0 Scenario

We consider a scenario where a central authority is tasked with distributing a limited supply of 100,000 vaccine doses across multiple cities. Each city has unique characteristics affecting how impactful the vaccines would be there.

Each city has:

- **Varying population sizes** (ranging from hundreds of thousands to millions)
- **Different infection rates** (reflecting the severity of the outbreak)
- **Different vaccine efficacy** (depending on logistics, climate, and demographic fit)
- **Fixed vaccine dose requirements** (no partial allocation allowed)

Example Scenario

City	Population	Infection Rate	Efficacy	Doses Needed
City A	**1,000,000**	**10%**	**90%**	**50,000**
City B	**500,000**	**20%**	**70%**	**30,000**

4.0 Objective

The primary objective of this project is to develop an algorithm that enables the optimal allocation of a limited supply of vaccine doses across multiple cities, with the aim of maximizing the total number of lives saved.

To achieve this, the solution must:

- Select a subset of cities that receive complete vaccine allocations based on a fixed dose requirement. Partial vaccinations are not allowed, meaning a city must receive all required doses or none.
- Respect the global constraint of a total vaccine dose limit (e.g., 100,000 doses).

- Account for city-specific parameters, including:
    - Population size
    - Infection rate (indicating outbreak severity)
    - Vaccine efficacy (contextual effectiveness in that region)
- Calculate the potential number of lives saved for each city based on the formula:
  livesSaved[i]= (population[i]×infectionRate[i]×efficacy[i]) ÷ 10,000
- Ensure fairness and efficiency in vaccine distribution, especially when dealing with diverse regional demands and outbreak intensities.

Secondary Objectives:

- Model the problem as a 0/1 Knapsack Problem, leveraging a proven dynamic programming approach to guarantee optimal results.
- Minimize vaccine wastage by avoiding partial allocations and optimizing dose utilization.
- Support policy-making decisions through a transparent and traceable algorithmic solution that explains why specific cities were selected.
- Lay the groundwork for future enhancements such as:
    - Regional priority weighting
    - Time-sensitive distribution
    - Budget or cost constraints
    - Support for real-time data input

This objective ensures that the solution is not only technically optimal but also ethically responsible and realistically applicable in critical public health scenarios.

5.0 Algorithm

5.1 Algorithm comparison

Paradigm	Strengths	Weaknesses	Suitability
Sorting	Easy to implement; can prioritize by "lives per dose"	Ignores constraints; not globally optimal	❌ Not suitable
Divide and Conquer (DAC)	Works for independent subproblems	Cannot handle constraints across subproblems	❌ Not suitable
Greedy	Fast; prioritizes high-efficiency cities	May skip combinations that yield higher total lives	⚠️ Suboptimal
Graph (Max Flow)	Great for network flow and capacities	Too complex for this setting; requires flow structures	❌ Overkill
Dynamic Programming (DP)	Guarantees global optimality; handles constraints	Slightly higher time/memory cost	✅ **Best fit**

5.2 Dynamic Programming

The vaccine allocation problem is a variation of the 0/1 Knapsack Problem, where:

- Cities = Items
- Doses = Weights
- Lives Saved = Values

Dynamic Programming is ideal because:

- It guarantees optimality by exploring all valid dose combinations.
- It respects constraints (no city can be partially vaccinated).
- It allows memoization of overlapping subproblems, improving efficiency.

5.3 Algorithm Explanation

We define a 2D DP table:

dp[d][c] = Maximum lives saved using **d doses** and the first **c cities.**

Recurrence Relation :

```
dp[d][c] = max(
   dp[d][c - 1],                          // Skip city c
   dp[d - doses[c]][c - 1] + livesSaved[c]        // Include city c
)
```

Base case

dp[0][c] = 0 → No doses = 0 lives saved

dp[d][0] = 0 → No cities = 0 lives saved

Optimization goal

Maximize:

- maximize ∑ livesSaved[i]

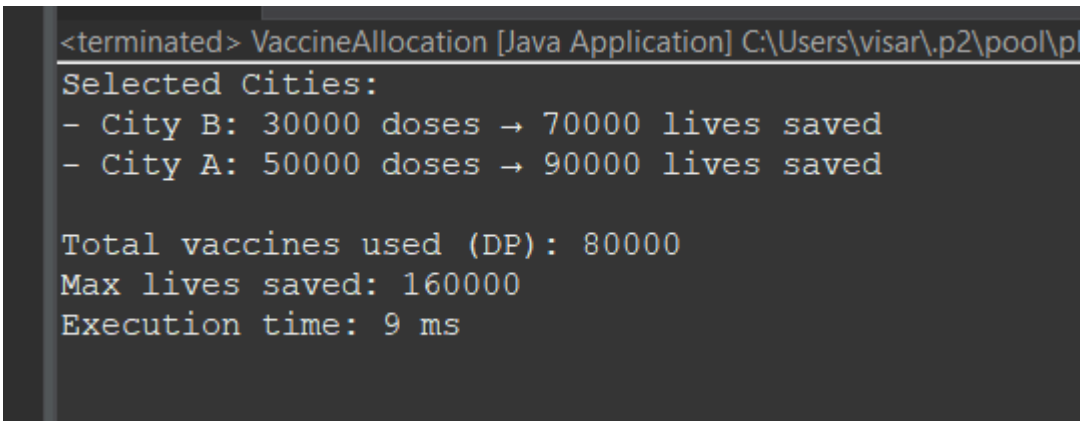
Subject to:

- subject to ∑ doses[i] ≤ totalDoses

Where:

- livesSaved[i] = (population × infectionRate × efficacy) / 10000

6.0 Output



7.0 Algorithm Correctness Analysis

An established dynamic programming technique based on the 0/1 Knapsack Problem serves as the foundation for the algorithm used to solve this vaccine allocation problem. The following features show that it is correct through the following aspects :

- **Base Case Initialization**
  The DP table dp[d][c] is initialized such that :
  - dp[0][c] = 0 → No lives can be saved with 0 availabe doses,regardless of how many cities exist.

  - Dp[d][0] = 0 → No lives can be saved with 0 cities to consider ,regardless of the number of avaiable doses .

    These base cases ensure a proper starting point for the DP recurrence to function correctly.

- **Recurrence Relation Validity**
  The recurrence formula:

  ```
  dp[d][c] = max(
     dp[d][c - 1],                        // Exclude current city
     dp[d - doses[c]][c - 1] + livesSaved[c] // Include current city
  )
  ```

  ensures that for each city, we explore both possibilities:

  - Skipping the city

  - Including the city, if enough doses remain

  This guarantees that all valid combinations are explored and the maximum value (lives saved) is retained at each state.

- **Subproblem Overlap**
  Using DP prevents needless recomputation, preserving accuracy and efficiency because decisions depend on previously calculated values (i.e., overlapping subproblems).

- **Backtracking**
  The cities that contribute to the best solution are tracked in a selected[][] table.

By listing precise city allocations, this helps confirm the accuracy and offers a traceable explanation of the outcome.

- **Proven Optimality**
  The DP solution guarantees global optimality, as it completely considers all feasible subsets under the constraints. This makes it ideal for critical real-world problems like vaccine allocation, where accuracy and fairness are essential.

8.0 Performance Analysis

Time Complexity Analysis

Let :

- n = number of cities
- d = total number of available vaccine doses

The time and space complexity of the algorithm are analyzed as follows:

Case	Time Complexity	Explanation
Best	Ω(n × d)	To assess potential combinations, every entry in the DP table needs to be taken into account.
Average	Θ(n × d)	Regardless of the data distribution, the DP table is always completely filled.
Worst	O(n × d)	Every city and dose combination requires an evaluation of every cell in the DP table.

- Space Complexity :

  To store intermediate results for every city and every possible dose value, the DP table needs O(n × d) space.
  However, if only the maximum value is needed (i.e., when tracking the chosen cities is not required), space can be optimized to O(d) using a 1D array.

9.0 Conclusion

This paper suggests that the best algorithmic paradigm to solve the vaccine allocation problem in a constrained environment is dynamic programming. Dynamic programming can explore every unique combination of cities selected while not violating a limit on doses of vaccine. Thus, dynamic programming guarantees optimality, unlike greedy or divide and conquer.

The algorithm allows for the maximization of the number of lives saved by calculating the optimal combination of cities receiving vaccines based on the city's population, infection rate, efficacy, and dose required. Additionally, the functioning of the algorithm is efficient for larger input sizes, guarantees correctness by following a structured recurrence, and logs output for cities selected which can be traced back.

Additionally, the model may be scaled and adjusted to account for requests in the real world features, such as weighting for risk of mortality, prioritization by region, partial vaccination, or budgeting. Dynamic programming is a powerful technique to address public health resource allocation problems because of the flexibility and guaranteed optimality.

10.0 References

1. WHO. (2021). *Fair Allocation Mechanism for COVID-19 Vaccines through the COVAX Facility*. https://www.who.int
   ➤ Explains how vaccine allocation principles were applied globally, relevant to your project scenario.

2. Paltiel, A. D., Schwartz, J. L., Zheng, A., & Walensky, R. P. (2020). *Clinical outcomes of a COVID-19 vaccine: implementation over efficacy*. Health Affairs, 40(1), 42–52.
   ➤ Highlights the importance of targeted vaccine distribution, similar to your optimization approach.

3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
   ➤ Standard textbook covering dynamic programming and algorithm correctness proofs.

4. Khan Academy. *Dynamic Programming – Knapsack Problem*.

   https://www.khanacademy.org/computing/computer-science/algorithms/dynamic-programming/a/dynamic-programming
   ➤ Educational explanation of the knapsack problem and how dynamic programming solves it.

Appendix

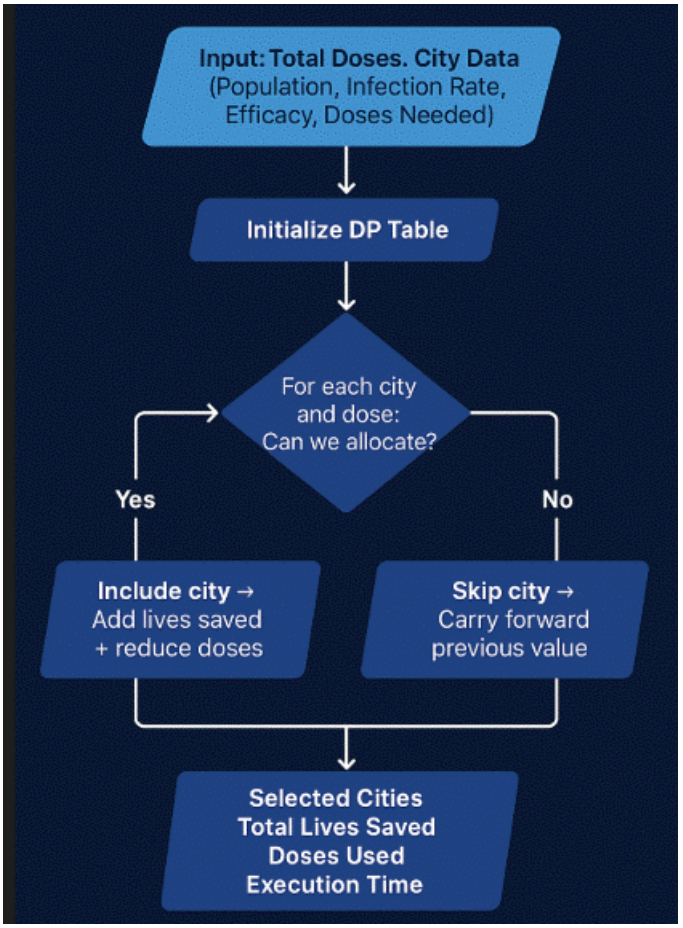
Code

```
public class VaccineAllocation {
    static class City {
        String name;
        int population, infectionRate, efficacy, doses;

        public City(String name, int population, int infectionRate, int efficacy, int doses) {
            this.name = name;
            this.population = population;
            this.infectionRate = infectionRate;
            this.efficacy = efficacy;
            this.doses = doses;
        }

        public int livesSaved() {
            return (population * infectionRate * efficacy) / 10000;
        }
    }

    public static int allocateVaccines(City[] cities, int totalDoses) {
        int n = cities.length;
        int[][] dp = new int[totalDoses + 1][n + 1];
        boolean[][] selected = new boolean[totalDoses + 1][n + 1];

        long startTime = System.currentTimeMillis();

        for (int d = 0; d <= totalDoses; d++) {
            for (int c = 1; c <= n; c++) {
                City city = cities[c - 1];
                if (city.doses <= d) {
                    int include = dp[d - city.doses][c - 1] + city.livesSaved();
                    int exclude = dp[d][c - 1];
                    if (include > exclude) {
                        dp[d][c] = include;
                        selected[d][c] = true;
                    } else {
                        dp[d][c] = exclude;
                    }
                } else {
                    dp[d][c] = dp[d][c - 1];
                }
            }
        }

        int d = totalDoses, c = n, totalUsed = 0;
```

```
        System.out.println("Selected Cities:");
        while (c > 0) {
            if (selected[d][c]) {
                City city = cities[c - 1];
                System.out.println("- " + city.name + ": " + city.doses + " doses → " +
city.livesSaved() + " lives saved");
                d -= city.doses;
                totalUsed += city.doses;
            }
            c--;
        }

        int maxLivesSaved = dp[totalDoses][n];
        long endTime = System.currentTimeMillis();

        System.out.println("\nTotal vaccines used (DP): " + totalUsed);
        System.out.println("Max lives saved: " + maxLivesSaved);
        System.out.println("Execution time: " + (endTime - startTime) + " ms");

        return maxLivesSaved;
    }

    public static void main(String[] args) {
        City[] cities = {
            new City("City A", 1000000, 10, 90, 50000),
            new City("City B", 500000, 20, 70, 30000)
        };

        int totalDoses = 100000;
        allocateVaccines(cities, totalDoses);
    }
}
```

Progress Report

<table>
<tr><td>Group Name</td><td>GROUP 10 (VACCINE ALLOCATION)</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Email</th><th>Phone number</th></tr>
<tr><td>VISAGAN RAO J VENKATA RAMANA</td><td>217138@student.upm.edu.my</td><td>010-2684173</td></tr>
<tr><td>RAVINRAJ MATHIALAHAN</td><td>216807@student.upm.edu.my</td><td>018-955 2884</td></tr>
<tr><td>MAYNON SELVAM</td><td>215548@student.upm.edu.my</td><td>017-252 1593</td></tr>
<tr><td>XIAO FENG</td><td>213836@student.upm.edu.my</td><td>011-4055 0025</td></tr>
</table>
</td></tr>
<tr><td>Problem scenario description</td><td>In a pandemic situation such as COVID-19, a government has limited vaccine doses available and must decide how to distribute them across multiple cities</td></tr>
<tr><td>Why it is important</td><td>Faster vaccination → lower infection rates → quicker economic recovery.</td></tr>
<tr><td>Problem specification</td><td>Maximize the total number of lives saved given a fixed number of vaccine doses, with no partial allocations allowed.</td></tr>
<tr><td>Potential solutions</td><td>Chosen Method: Dynamic Programming
Guarantees optimal lives saved
Adapts to any number of cities
Handles exact dose constraints
Proven solution strategy for bounded allocation problems</td></tr>
</table>

Sketch (framework, flow, interface)	

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

<table>
<tr><td>Group Name</td><td>GROUP 10 (VACCINE ALLOCATION)</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Role</th></tr>
<tr><td>VISAGAN RAO J VENKATA RAMANA</td><td>Code</td></tr>
<tr><td>RAVINRAJ MATHIALAHAN</td><td>Algorithm correctness</td></tr>
<tr><td>MAYNON SELVAM</td><td>Problem statement and objectives</td></tr>
<tr><td>XIAO FENG</td><td>Performance analysis</td></tr>
</table>
</td></tr>
<tr><td>Problem statement</td><td>To allocate a limited number of vaccine doses across multiple cities such that the total number of lives saved is maximized, considering each city’s population, infection rate, vaccine efficacy, and dose requirement.</td></tr>
<tr><td>Objectives</td><td>Design an optimal vaccine distribution algorithm.

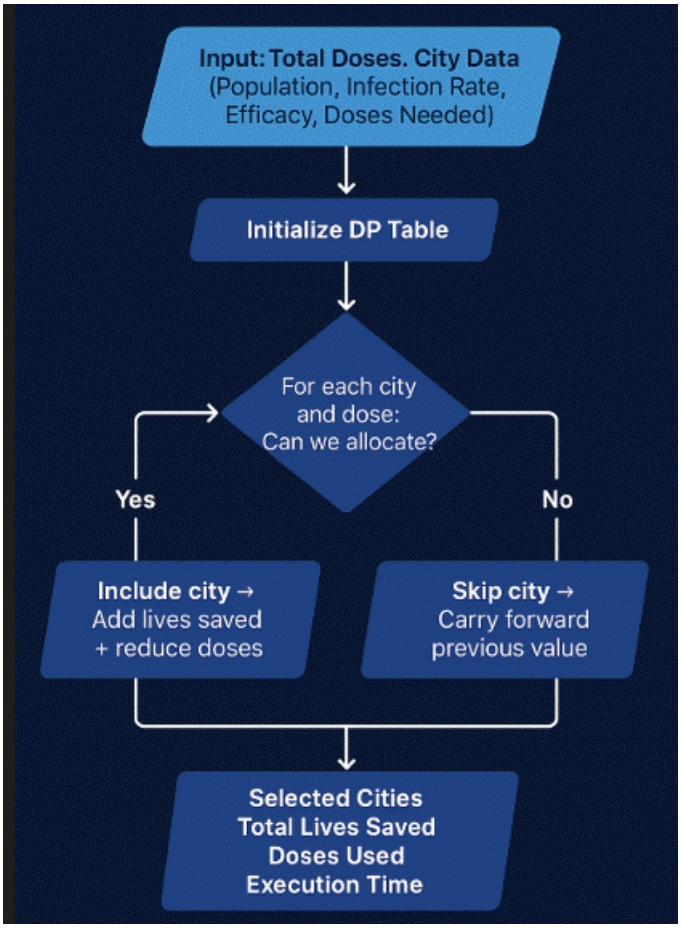
Maximize total lives saved with constrained resources.

Implement and test the algorithm in Java.

Evaluate algorithm correctness and time complexity.</td></tr>
<tr><td>Expected output</td><td>● List of selected cities for vaccine allocation.
● Number of lives saved in each selected city.
● Total vaccine doses used.
● Execution time.
● Optimality guarantee using Dynamic Programming.</td></tr>
</table>

Problem scenario description	During a pandemic (e.g., COVID-19), the government has only **100,000 vaccine doses** available. Different cities have: ● Different population sizes ● Different infection rates ● Different vaccine efficacies ● Specific dose needs Due to limited resources, it's not possible to vaccinate every city. The system must determine how to allocate vaccines to **maximize lives saved**.
Why it is important	**Health Impact**: Saves the maximum number of lives. **Resource Efficiency**: Ensures vaccines are not wasted. **Ethical Responsibility**: Prioritizes high-risk populations. **Economic Recovery**: Vaccinating critical areas can speed up reopening. **Government Planning**: Supports data-driven decision making in real-world emergencies.
Problem specification	**Input:** ● Total vaccine doses: e.g., 100,000 ● Cities: ○ Population (e.g., 1,000,000) ○ Infection rate (e.g., 10%) ○ Vaccine efficacy (e.g., 90%) ○ Doses needed (e.g., 50,000) **Output:** ● Optimal subset of cities to vaccinate ● Total lives saved ● Total doses used ● Performance statistics **Constraint:** ● No partial vaccination allowed for any city

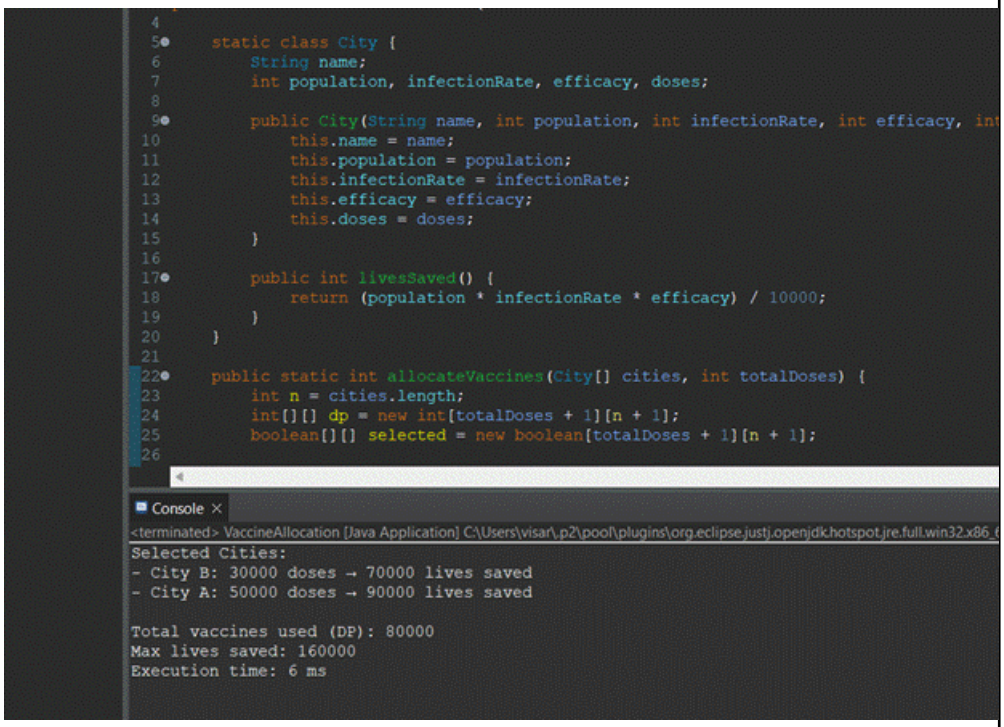
Potential solutions	LivesSaved = (Population × InfectionRate × Efficacy) / 10,000 **Dynamic Programming** Optimal, handles constraints well Requires more memory/time

Sketch (framework, flow, interface)	

Methodology	

Milestone	Time
Refine problem scenario and constraints	wk10
Explore example problems & compare algorithms (Greedy vs DP)	wk11
Develop and debug final Java code	wk12
Analyse algorithm correctness and time complexity	wk13
Prepare online documentation, portfolio, and final presentation slides	wk14

Project Progress (Week 10 – Week 14)

Milestone 1	**Completed code and expected output received**
Date (week)	**12/6/2025(week13)**
Description/ sketch	

```
    static class City {
        String name;
        int population, infectionRate, efficacy, doses;

        public City(String name, int population, int infectionRate, int efficacy, int
            this.name = name;
            this.population = population;
            this.infectionRate = infectionRate;
            this.efficacy = efficacy;
            this.doses = doses;
        }

        public int livesSaved() {
            return (population * infectionRate * efficacy) / 10000;
        }
    }

    public static int allocateVaccines(City[] cities, int totalDoses) {
        int n = cities.length;
        int[][] dp = new int[totalDoses + 1][n + 1];
        boolean[][] selected = new boolean[totalDoses + 1][n + 1];
```

```
Console ×
<terminated> VaccineAllocation [Java Application] C:\Users\visar\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_
Selected Cities:
- City B: 30000 doses → 70000 lives saved
- City A: 50000 doses → 90000 lives saved

Total vaccines used (DP): 80000
Max lives saved: 160000
Execution time: 6 ms
```

<table>
<tr><td>Role</td><td>
<table>
<tr><td>Member 1</td><td>Member 2</td><td>Member 3</td><td>Member 4</td></tr>
<tr><td>Visagan-code</td><td>Maynon – Objectives and input</td><td>Ravinraj – Algorithm Correctness</td><td>Xiao Feng –

Performance Analysis</td></tr>
</table>
</td></tr>
</table>

Milestone 2	**Algortithm correctness and performance analysis**
Date (Wk)	**14/6/2025(week13)**
Description/ sketch	**Base Case Validity:** **- dp[0][c] = 0 → 0 doses save 0 lives** **- dp[d][0] = 0 → 0 cities save 0 lives** ↓ **Recurrence Relation:** **dp[d][c] = max(dp[d][c-1], dp[d - doses[c]][c-1] + livesSaved[c])** **- Chooses the better of:** **• Skipping the current city** **• Including the current city if doses allow** ↓ **Optimal Substructure:** **- The problem is solved by solving subproblems:** **• e.g., max lives saved with d-k doses and c-1 cities** ↓ **Overlapping Subproblems:**

<table>
<tr><td></td><td>

- Same subproblems (dose-city combinations) are reused

- Enables efficient DP with memoization

↓

Performance:

- Time Complexity: O(n × d)

- Space Complexity: O(n × d) (can be optimized to O(d))

</td></tr>
<tr><td>Role</td><td>

<table>
<tr><td>Member 1</td><td>Member 2</td><td>Member 3</td><td>Member 4</td></tr>
<tr><td>Visagan-code</td><td>Maynon – Objectives and input</td><td>Ravinraj – Algorithm Correctness</td><td>Xiao Feng –

Performance Analysis</td></tr>
</table>

</td></tr>
</table>