GROUP PROJECT

Design Algorithm and Analysis

Presented by :

KERENA

HAVITRA

PROBLEM STATEMENT

A coastal region frequently experiences flooding due to various factors such as weather patterns, geographical features, and historical data of water levels. The local government aims to enhance flood detection capabilities and optimize resource allocation to minimize damage, ensure public safety, and maximize the efficiency of emergency response efforts.

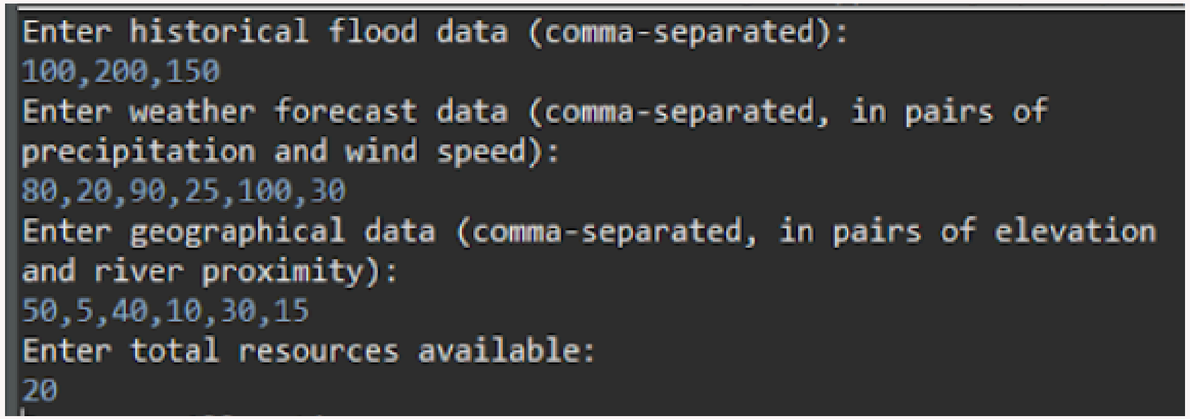
Objectives

1. Predicts flood levels accurately across different zones based on historical data and real-time weather forecasts.

2. Allocates resources (e.g., sandbags, rescue teams, medical supplies) optimally to mitigate flood impact and ensure rapid response.

INPUT DATA

- Historical records of flood occurrences, including severity levels (e.g., water levels).
- Current and forecasted weather conditions relevant to flood prediction, such as precipitation, and wind speed.
- Records of geographical data, such as elevation and river proximity.
- Current record of total resources available.



Algorithm Suitability

Algorithm	Strengths	Weakness
Sorting	Helps in prioritizing zones based on flood severity.	Insufficient for handling dynamic and dependent variables like road conditions and real-time data.
Divide and Conquer (DAC)	Breaks the problem into manageable sub-problems.	May not effectively address dependencies between zones and real-time changes.
Dynamic Programming (DP)	Suitable for problems with overlapping subproblems and optimal substructure. Efficient in handling state-based predictions.	Can be memory-intensive and complex to implement.
Greedy Algorithms	Simple and fast, useful for problems where local optimization leads to global optimum.	May not always produce the optimal solution for complex problems with dependencies.
Graph Algorithms	Effective for modeling and solving network-based problems, ideal for representing zones and their connections.	Computationally intensive for large graphs, especially with real-time data.

CHOSEN ALGORITHMS- GREEDY

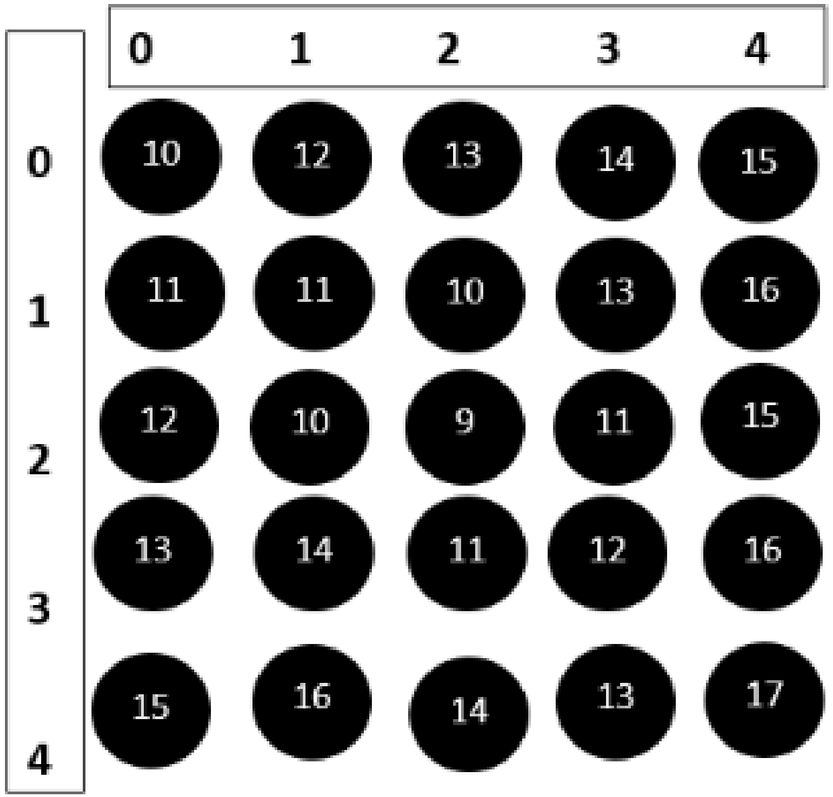
Greedy Algorithm

- Used to find the lowest elevation and mark cells as flooded if they are below the water level where the water moving to neighboring cells recursively.

- Simplicity: Greedy algorithms are often simpler to implement and understand.
- Speed: Greedy algorithms can be faster because they make decisions based on local optima without exploring all possibilities.

HOW IT WORKS: GREEDY ALGORITHM

1. Initialize the Map and Elevation Data: We have a 5x5 grid representing a geographical area. Each cell has an elevation value.
2. Set Initial Water Levels: Define an initial water level that will be compared against the elevation.
3. Apply Greedy Algorithm: Start from the lowest elevation and mark cells as flooded if they are below the water level, moving to neighboring cells recursively.



HOW IT WORKS: GREEDY ALGORITHM

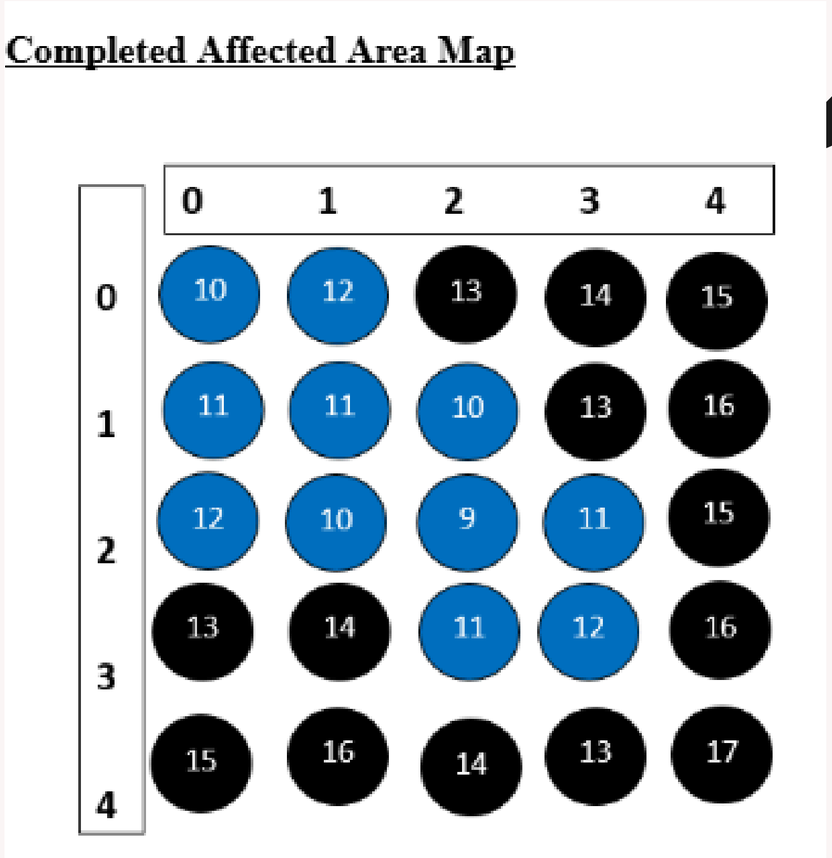
1. Water Level:

   Assume the water level is set at 12.

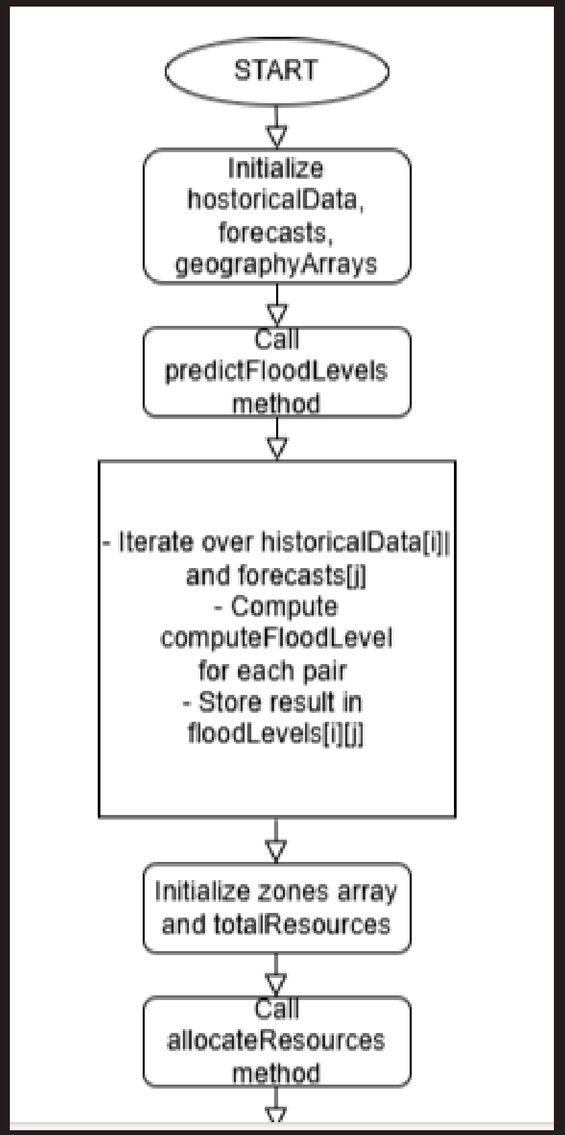
2. Greedy Algorithm for Flood Prediction:

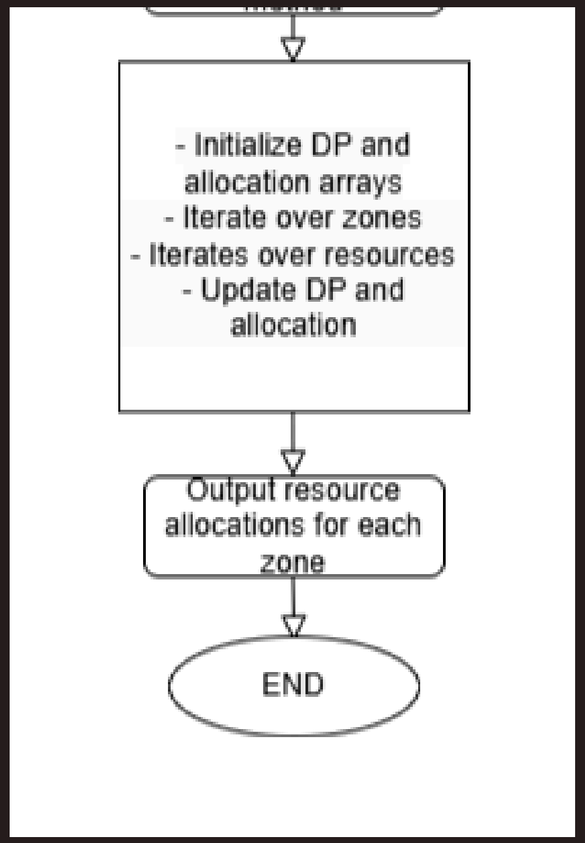
- Find the lowest elevation cell below the water level (start at 9).
- Mark the cell as flooded (cell (2, 2)).
- Check neighboring cells and repeat the process if they are also below the water level.

3. Elevation Value: Height of the ground above sea level.
4. Flooding Prediction: Areas with elevation values at or below the water level will be flooded.
5. Purpose: Helps in planning and disaster management by predicting flood-prone areas based on elevation data.



FLOWCHART


START
Initialize hostoricalData, forecasts, geographyArrays
Call predictFloodLevels method
- Iterate over historicalData[i]| and forecasts[j]
- Compute computeFloodLevel for each pair
- Store result in floodLevels[i][j]
Initialize zones array and totalResources
Call allocateResources method


- Initialize DP and allocation arrays
- Iterate over zones
- Iterates over resources
- Update DP and allocation
Output resource allocations for each zone
END

CODE

```
import java.util.*;

public class FloodPredictionAndResourceAllocation {
static class WeatherForecast {
    // Placeholder class for weather forecast data (expand as
needed)
}


static class Geography {
    // Placeholder class for geographical data (expand as
needed)
}

static class Zone {
    int id;
    int predictedFloodLevel;
    int requiredResources;
    Zone(int id) {
        this.id = id;
    }
}
static class ResourceAllocation {
    int zoneId;
    int allocatedResources;

    ResourceAllocation(int zoneId, int allocatedResources) {
        this.zoneId = zoneId;
        this.allocatedResources = allocatedResources;
    }
}
public static int[][] predictFloodLevels(int[] historicalData,
WeatherForecast[] forecasts, Geography[] geography) {
    int n = historicalData.length;
    int m = forecasts.length;
    int[][] floodLevels = new int[n][m];

  // Dummy calculation for flood levels (replace with actual
logic)
    Random random = new Random();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            floodLevels[i][j] = (int) (historicalData[i] * 0.6 +
random.nextInt(100) * 0.3 + random.nextInt(100) * 0.1);
```

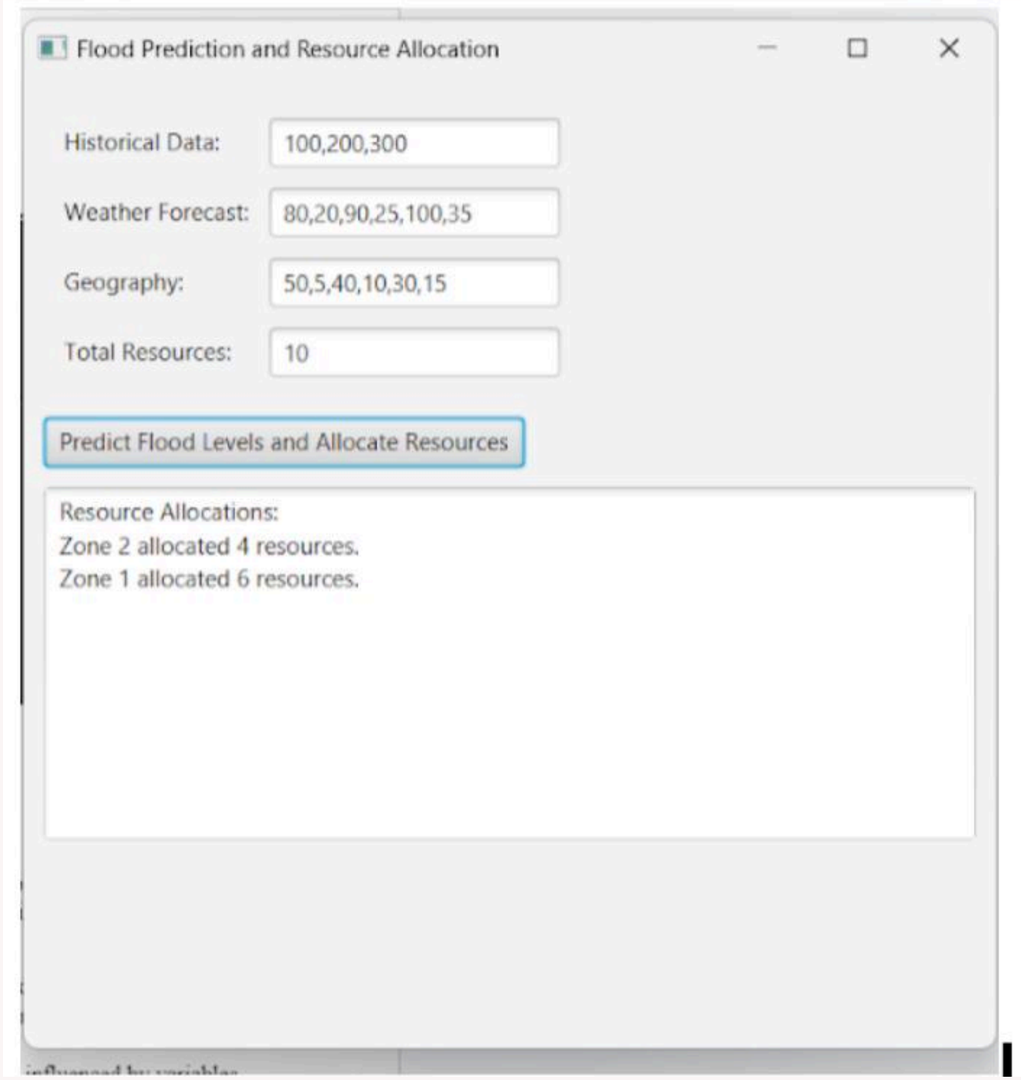
```
    }
    }
    return floodLevels;
}

public static List<ResourceAllocation>
allocateResources(Zone[] zones, int totalResources) {
    int n = zones.length;
    int[][] dp = new int[n + 1][totalResources + 1];



   int[][] allocation = new int[n][totalResources + 1];

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= totalResources; j++) {
            dp[i][j] = dp[i - 1][j]; // Initialize with previous values
            if (j >= zones[i - 1].requiredResources) {
                int value = dp[i - 1][j - zones[i - 1].requiredResources]
+ zones[i - 1].predictedFloodLevel;
                if (value > dp[i][j]) {
                    dp[i][j] = value;
                    allocation[i - 1][j] = zones[i - 1].requiredResources;
                }
            }
        }
    }
```

```
    List<ResourceAllocation> result = new ArrayList<>();
    int remainingResources = totalResources;

    for (int i = n - 1; i >= 0; i--) {
        if (allocation[i][remainingResources] > 0) {
            result.add(new ResourceAllocation(zones[i].id,
allocation[i][remainingResources]));
            remainingResources -= allocation[i]
[remainingResources];
        }
    }
    return result;
}

public static void main(String[] args) {
    // Example usage
    int[] historicalData = {100, 200, 150}; // Dummy historical
flood data
    WeatherForecast[] forecasts = new WeatherForecast[3]; //
Dummy weather forecasts
    Geography[] geography = new Geography[3]; // Dummy
geographical data

    // Predict flood levels for each zone based on historical
data and forecasts
    int[][] floodLevels = predictFloodLevels(historicalData,
forecasts, geography);
```

```
    // Create zones with predicted flood levels and randomly
assigned resource requirements

    Zone[] zones = new Zone[3];
    Random random = new Random();
    for (int i = 0; i < zones.length; i++) {
        zones[i] = new Zone(i);
        zones[i].predictedFloodLevel = floodLevels[i][0]; //
Example: Using the first forecast
        zones[i].requiredResources = random.nextInt(10) + 1; //
Random resource requirement (1 to 10)
    }

    int totalResources = 20; // Total resources available

    // Allocate resources optimally using dynamic
programming approach

    List<ResourceAllocation> allocations =
allocateResources(zones, totalResources);

    // Print resource allocations

    System.out.println("Resource Allocations:");
    for (ResourceAllocation allocation : allocations) {
        System.out.println("Zone " + allocation.zoneId + "
allocated " + allocation.allocatedResources + " resources.");
    }
}}
```

OUTPUT

```
Enter historical flood data (comma-separated):
100,200,150
Enter weather forecast data (comma-separated, in pairs of
precipitation and wind speed):
80,20,90,25,100,30
Enter geographical data (comma-separated, in pairs of elevation
and river proximity):
50,5,40,10,30,15
Enter total resources available:
20
Resource Allocations:
Zone 2 allocated 9 resources.
Zone 1 allocated 5 resources.
Zone 0 allocated 5 resources.
```



- Historical Data: Offers a foundational comprehension of previous flood levels.
- Weather forecasts: Affect forecasts by taking into account the amount of precipitation and wind speed at the moment.
- Geographical Information: Each zone's susceptibility to flooding is influenced by variables including elevation and the proximity of rivers.

ALGORITHM ANALYSIS

TIme Complexity:

Flood Prediction: $O(n \times m)$, where n is the number of zones and m is the number of time steps.

Resource Allocation: $O(n \times R)$, where n is the number of zones and R is the total resources.

- **Best Case: $O(n \times m + n \times R)$**
- **Average Case: $O(n \times m + n \times R)$**
- **Worst Case: $O(n \times m + n \times R)$**

This complexity is efficient for the problem constraints, ensuring timely and effective flood prediction and resource allocation.

CONCLUSION

Integration of Techniques

- Greedy Algorithm: Optimizes immediate relief distribution decisions based on current data.

Real-Time Adaptation

- Adapts to changing data inputs for accurate flood forecasts and distribution plans.

Impact

- Enhances disaster preparedness and response capabilities.
- Facilitates efficient allocation of resources in crisis situations.

THANK YOU !