

# **CSC4202-5: DESIGN AND ANALYSIS OF ALGORITHMS**

## **GROUP PROJECT LANDSLIDE RISK REDUCTION IN A HILLY TERRAIN**

**LECTURER NAME:**

**DR. NUR ARZILAWATI BINTI MD YUNUS**

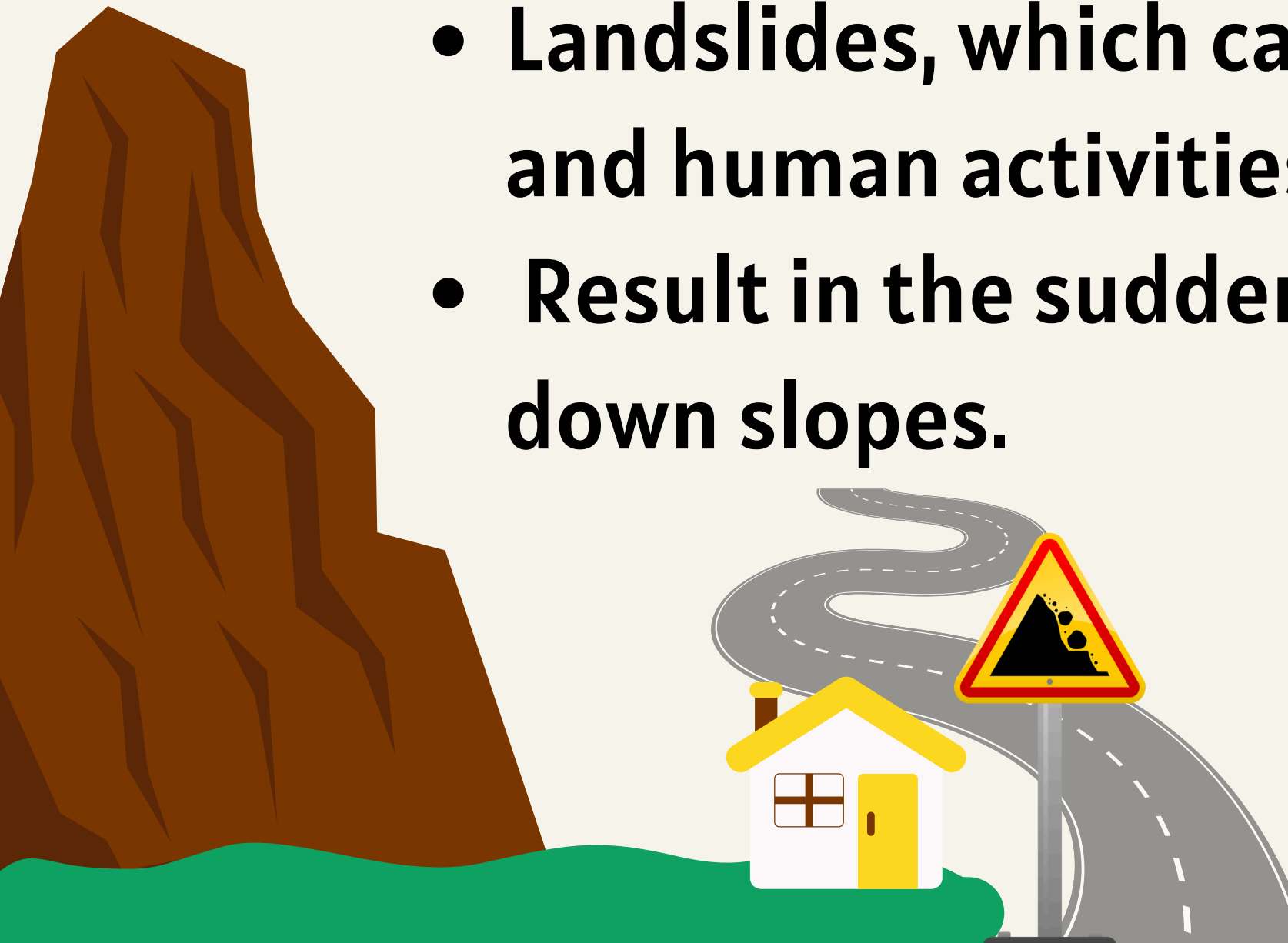
HASRINAH BINTI KURONG 213110

MOONNA ZULKAFLY 212137

ANIS NADIRA BINTI NOOR MAISAM 210423

# SCENARIO: LANDSLIDE

- Region with steep slopes, hilly terrain, and unstable geological formations, prone to landslides.
- Home to several communities and infrastructure
- Landslides, which can be triggered by heavy rainfall, seismic and human activities.
- Result in the sudden movement of rock, soil, and debris down slopes.





# DAMAGE IMPACT

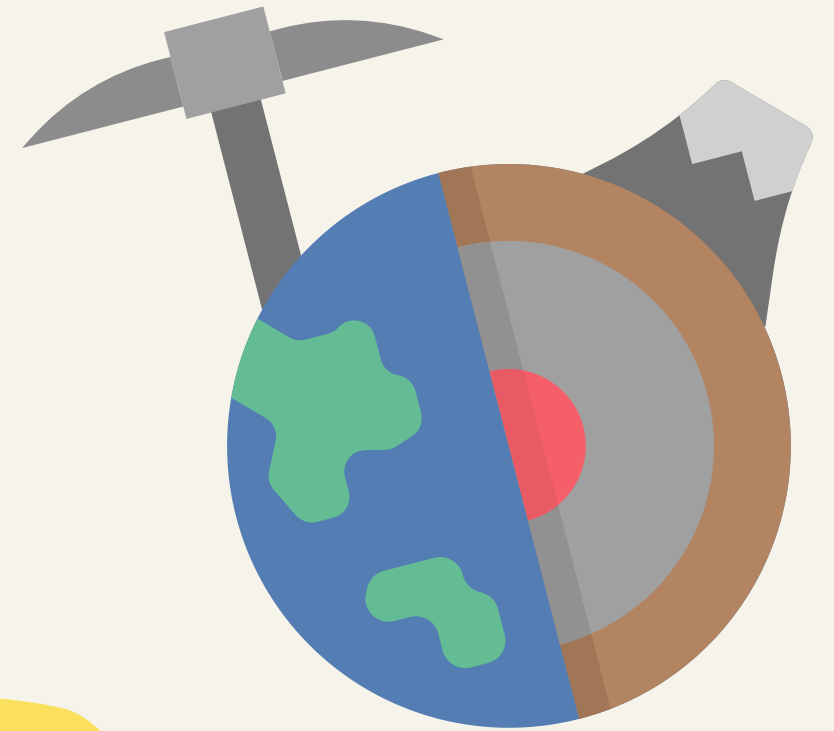
- **Significant damage:**
  - **loss of life**
  - **displacement of communities**
  - **disruption of transportation networks**
  - **destruction of homes, agricultural lands, and natural ecosystems**



# IMPORTANCE OF AAD

## Landslide risk reduction

- optimizing landslide monitoring, early warning systems, land use planning and emergency response measures
- **automated identification of evacuation routes during landslide.**
- analyze geological data, rainfall patterns, terrain mapping, and land use changes
- identify areas at risk and develop strategies for landslide mitigation, preparedness and evacuation.



## GOAL

Identify least risky evacuation paths for landslide victims, during the emergency situations.



# WHY FINDING OPTIMAL SOLUTION IS IMPORTANT?

**MINIMIZE DAMAGE & ECONOMIC LOSS**

**prioritize cost-effective mitigation strategies**

**ENHANCING RESILIENCE**

**help communities recover quickly from landslides**



**SAVE LIVES**

**ensures communities are prepared to evacuate**



**SUSTAINABLE DEVELOPMENT**

**identify safe areas for construction**





# ALGORITHMS SUITABILITY



Sorting	DAC	DP	Greedy	Graph
Organize and retrieve data	Breaking down problems into smaller	Stores solutions to subproblems, avoid redundant calculations	Make decision based on local optima	Modelling relationships between data points
Useful for data organization	Parallel processing for risk assessments	Suitable for finding the least risk path	Helpful for quick identification of high-risk areas	Can be used to model landslide terrain



# DYNAMIC PROGRAMMING

## 1. Optimal Substructure

solving complex problems by breaking them down into smaller subproblems.

## 2. Scalability


can handle large-scale problems with many overlapping subproblems

## 3. Efficient Use of Resources

DP avoids redundant calculations by storing intermediate results.

## 4. Handling Multiple Variables

DP can efficiently handle and combine multiple variables to provide comprehensive solutions





# ALGORITHM DESIGN



- 1. Collect terrain data (elevation, slope, geology) and create a risk grid.**
- 2. Use dynamic programming to calculate the minimum risk value for each location on the grid.**
- 3. Iterate through the grid, considering neighboring cell risks to find the overall safest path.**
- 4. Identify the minimum risk value at the end point (bottom-right corner), representing the safest route across the terrain.**





# THE ALGORITHM PARADIGM

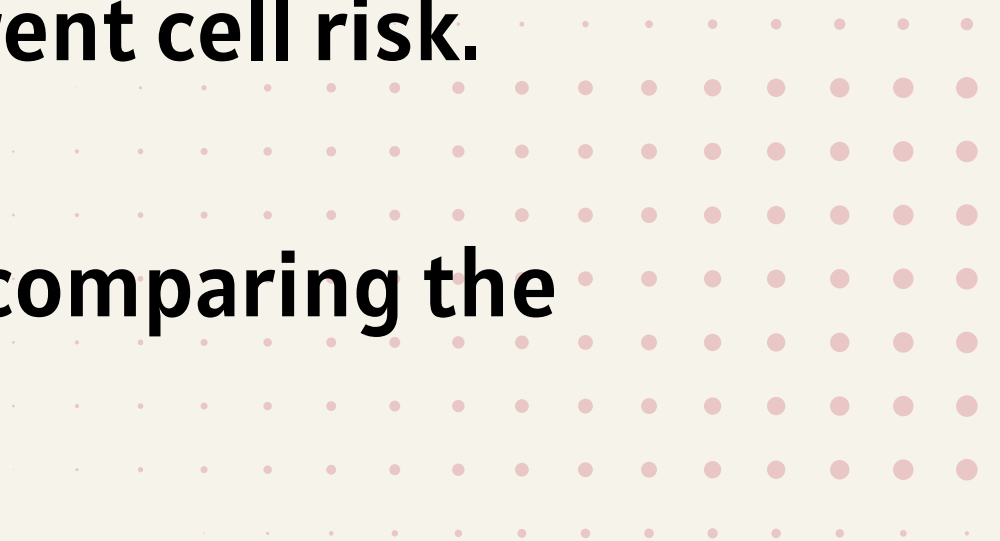
## Paradigm: Dynamic Programming (DP)


- Breaks down complex problems into simpler subproblems.
- Stores solutions to subproblems to avoid redundant calculations.

## Recurrence Relation:

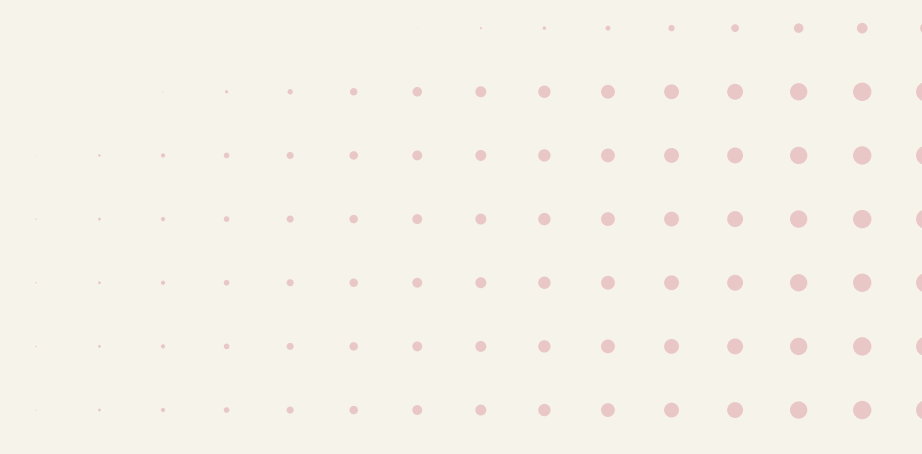
- In dynamic programming, a recurrence relation defines how the solution to a larger problem is built from the solutions to smaller subproblems.
- Represents minimum risk from top or left cell, plus current cell risk.

## Optimization Function:

- Ensures that the best (optimal) solution is selected by comparing the outcomes of different choices
  - Output: Minimum risk path across the entire terrain.
- 



# PSEUDOCODE

1. Start
  2. Initialize rows and cols from terrain dimensions
  3. Create a 2D array dp for dynamic programming
  4. Initialize dp with maximum values
  5. Set `dp[0][0]` as the starting point of the terrain grid
  6. Iterate through each cell of the terrain grid
    - Update `dp[i][j]` with the minimum risk to reach the current cell
  7. Return `dp[rows-1][cols-1]` as the minimum risk path
  8. End
- 

# CODE

```
package algo;
import java.util.Arrays;
```

```
public class LandslideRiskReduction {
```

```
    // Function to find the minimum risk path
```

```
    public static int findMinRisk(int[][] terrain) {
```

```
        int rows = terrain.length;
```

```
        int cols = terrain[0].length;
```

```
        // Create a dp array to store the minimum risk at each cell
```

```
        int[][] dp = new int[rows][cols];
```

```
        // Initialize the dp array with maximum possible values
```

```
        for (int[] row : dp) {
```

```
            Arrays.fill(row, Integer.MAX_VALUE);
```

```
        }
```

```
        // Start from the top-left corner
```

```
        dp[0][0] = terrain[0][0];
```

```
        // Fill the dp array with the minimum risk values
```

```
        for (int i = 0; i < rows; i++) {
```

```
            for (int j = 0; j < cols; j++) {
```

```
                if (i > 0) {
```

```
                    dp[i][j] = Math.min(dp[i][j], dp[i-1][j] + terrain[i][j]);
```

```
                }
```

```
                if (j > 0) {
```

```
                    dp[i][j] = Math.min(dp[i][j], dp[i][j-1] + terrain[i][j]);
```

```
                }
```

```
            }
```

```
        }
```

```
    // Return the minimum risk to reach the bottom-right corner
```

```
        return dp[rows-1][cols-1];
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        // Example terrain grid
```

```
        int[][] terrain = {
```

```
            {1, 3, 1},
```

```
            {1, 5, 1},
```

```
            {4, 2, 1}
```

```
        };
```

```
        // Find and print the minimum risk
```

```
        int minRisk = findMinRisk(terrain);
```

```
        System.out.println("The minimum risk path has a risk of: " + minRisk);
```

```
    }
```

```
}
```

```
Console ×
```

```
<terminated> LandslideRiskReduction [Java Application] C:\Users\moonnn\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64\jre\bin\java.exe -Djava.library.path=C:\Users\moonnn\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64\jre\bin\plugin2 -jar C:\Users\moonnn\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64\jre\bin\plugin2\plugin2.jar
```

```
The minimum risk path has a risk of: 7
```

# ALGORITHM'S CORRECTNESS ANALYSIS

The algorithm implements a dynamic programming approach to find the minimum risk path in a 2D grid.

## 1. Initialization

- The `dp` array is initialized to store the minimum risk at each cell. The initial cell `dp[0][0]` is set to the risk value of the starting cell `terrain[0][0]`, ensuring the starting point is correctly represented.

## 2. Filling the DP Array

- The algorithm iterates over each cell in the grid and updates the `dp` array with the minimum risk value by considering the minimum risk from the top cell (`i-1, j`) and the left cell (`i, j-1`). This ensures that each cell in the `dp` array holds the minimum risk required to reach that cell.

## 3. Final Value

- The algorithm effectively uses dynamic programming to compute the minimum risk path from the top-left to the bottom-right corner of the grid. It ensures that at each step, the minimum cumulative risk is updated, leveraging optimal solutions to subproblems. This approach is crucial for safely evacuating areas prone to landslides, as it identifies the safest routes based on the calculated risk landscape.

# TIME COMPLEXITY ANALYSIS

Each cell in the grid represents a terrain segment with a risk value. The algorithm processes each cell exactly once, so the time complexity is always the same, regardless of the input values.

## I. Best Case

The best case occurs when the terrain values are such that the minimum risk path is straightforward, either going directly right or down, but the algorithm still needs to compute the risk values for all cells.

Time complexity is  $O(\text{rows} * \text{cols})$ . Example terrain grid:

```
int[ ][ ] terrain = {  
    {1, 1, 1},  
    {1, 1, 1},  
    {1, 1, 1}  
};
```

# TIME COMPLEXITY ANALYSIS

Each cell in the grid represents a terrain segment with a risk value. The algorithm processes each cell exactly once, so the time complexity is always the same, regardless of the input values.

## 2. Average Case

For an average terrain configuration, the algorithm processes each cell once. If the terrain grid has varying risk values, neither extremely high or extremely low, the algorithm needs to explore various paths to find the one with the least cumulative risk by processing each cell exactly once. Time complexity is  $O(\text{rows} * \text{cols})$ . Example terrain grid:

```
int[ ][ ] terrain = {  
    {1, 3, 1},  
    {2, 5, 1},  
    {4, 2, 1}  
};
```



# TIME COMPLEXITY ANALYSIS

Each cell in the grid represents a terrain segment with a risk value. The algorithm processes each cell exactly once, so the time complexity is always the same, regardless of the input values.

## 3. Worst Case

Even in the worst-case scenario, where the risk values are high and require maximum computation, the algorithm still processes each cell once to find the minimum cumulative risk. Time complexity is  $O(\text{rows} * \text{cols})$ . Example of terrain grid:

```
int[ ][ ] terrain = {  
    {5, 8, 9},  
    {7, 10, 6},  
    {12, 11, 4}  
};
```

**Larana University | 2024**

**THANK YOU**

**Presented By : Adeline Palmerston**