Session 1: Homework Recap

1 Data Structure for the Maze:

- **Grid Representation**: The maze can be represented as a 2D array (or matrix) where each cell can either be a wall or a passage.
    - Example: A grid of dimensions n×mn \times mn×m, where each cell (i,j)(i, j)(i,j) could be marked as:
        - 0 for a wall.
        - 1 for an open path.
- **Cell Properties**: Each cell might have properties like:
    - Walls on the north, south, east, and west sides.
    - Whether it has been visited or not (for the maze generation algorithm).

2 Maze Generation Algorithm:

- **Recursive Division**: A recursive algorithm that divides the maze into smaller sections by adding walls and then creating openings.
    - Start with a grid filled with walls.
    - Recursively divide the grid by adding horizontal or vertical walls.
    - Add one door (opening) in the wall at random, except at the borders.
    - Repeat the process for the new sub-regions until each section is sufficiently small.
- **Depth-First Search (DFS)** for Maze Generation (Alternative):
    - Start at a random cell.
    - Mark the current cell as visited.
    - Randomly select a neighboring cell and check if it has been visited.
    - If not visited, remove the wall between the current cell and the chosen cell, then move to the neighboring cell and repeat.
    - If all neighbors are visited, backtrack to the previous cell.

Session 2:

Host: Lucas
Scribe: Anita
Secretary: Karim

4)

Axioms

1. Maze Boundaries:

- The maze must have boundary walls on all four sides:
  - Top row's top walls and bottom row's bottom walls must be set to 1.
  - Left column's left walls and right column's right walls must be set to

2. Unique Exit:

- The cell at the bottom-right corner ([width-1][height-1]) must have its exit attribute set to 1, and all other cells must have exit set to 0.

3. One Door Per Wall in Recursive Division:

- When a wall is placed between two subregions during division, it should have exactly one door.

4. Each Cell has 4 Walls:

- Each cell in the maze has four walls (left, right, top, bottom), and each wall is either "open" (0) or "closed" (1).

Pre-Conditions

1. Maze Initialization:

- The initializeMaze function must be called with valid dimensions (width > 0 and height > 0).

2. Memory Allocation:

◦ Dynamic memory for the maze and its cells must be successfully allocated before accessing or modifying any cells.

3. Recursive Division:

◦ Subregions must have a minimum width and height of 2 for further division.

4. Shortest Path:

◦ The maze must be fully initialized with a valid exit

(maze->cells[width-1] [height-1]. exit == 1) before attempting to find the shortest path.

5. Freeing the Maze:

◦ The maze and its substructures must not be NULL before calling freeMaze to avoid memory leaks.

1. **Memory Leak Test: Tingting**

Ensures that all dynamic allocations are properly freed. Use tools like Valgrind to detect leaks.

2. **Dimension Limit Test: Tingting**

Ensures that the generation works with very small or very large mazes. Also tests asymmetrical dimensions.

3. **Performance and Scalability Test: Sarah**

Measures the execution time on large mazes. Verifies if the program remains smooth and efficient.

4. **Maze Integrity Test: Anita**

Ensures that the outer and inner walls are consistent. Verifies the validity of the maze structure.

5. **Randomness Test: Karim**

Ensures that each generation is unique and random. Also tests reproducibility with a fixed seed.

6. **Solvability Test: Lucas**

Ensures that the maze can be solved from point A to point B. Verifies that there are no inaccessible areas.

7. **Correct Division Test: Daniel**

Ensures that each division creates valid sub-mazes. Also tests the depth of the divisions.

8. **Pointer Validity Test (NULL): Daniel**

Ensures that the pointers for the sub-mazes are always valid or NULL. Makes sure to avoid memory access errors.