# Maze Generation and Resolution Algorithms

## Introduction

This report provides a brief description of the algorithms used for maze generation and resolution in our project. We explain our choices and present the algorithms in both English and pseudo-code for clarity.

## Maze Generation Algorithm

### Algorithm Description

We implemented a recursive division algorithm to generate the maze. This method involves dividing the maze into smaller sections by adding walls and creating openings (doors) until the sections are indivisible. At each recursive call, the algorithm chooses to divide either horizontally or vertically based on the dimensions of the current section.

### Pseudo-Code

```
function divide(maze, x, y, width, height):
    if width <= 1 or height <= 1:
        return
    if width > height:
        orient = VERTICAL
    else:
        orient = HORIZONTAL
    if orient == VERTICAL:
        wall_x = random integer between x + 1 and x + width - 2
        open_y = random integer between y and y + height - 1
        for i from y to y + height - 1:
            if i != open_y:
                maze[i][wall_x] += WALL
        divide(maze, x, y, wall_x - x, height)
        divide(maze, wall_x + 1, y, x + width - wall_x - 1, height)
    else:
        wall_y = random integer between y + 1 and y + height - 2
        open_x = random integer between x to x + width - 1
        for i from x to x + width - 1:
            if i != open_x:
                maze[wall_y][i] += WALL
        divide(maze, x, y, width, wall_y - y)
        divide(maze, x, wall_y + 1, width, y + height - wall_y - 1)
```

### Explanation of Choice

We chose the recursive division algorithm because it creates mazes with a good balance of complexity and solvability. It ensures that the maze is fully connected and that there is a unique path between any two points, which is desirable for our application. Additionally, the algorithm is relatively simple to implement and performs efficiently for mazes of various sizes.

## Maze Resolution Algorithm

### Algorithm Description

For maze resolution, we implemented the Breadth-First Search (BFS) algorithm. BFS systematically explores the maze level by level, ensuring that the shortest path from the entrance to the exit is found. The algorithm uses a queue to keep track of cells to visit next and marks visited cells to avoid redundant exploration.

### Pseudo-Code

```
function bfs_solve(maze, start):
    create a queue Q
    mark start cell as visited and enqueue it
    while Q is not empty:
        current = dequeue Q
        if current is exit:
            reconstruct path from start to exit
            return path
        for each neighbor in adjacent cells of current:
            if neighbor is accessible and not visited:
                mark neighbor as visited
                set neighbor's parent to current
                enqueue neighbor
    return no path found
```

### Explanation of Choice

We selected BFS because it guarantees finding the shortest path in an unweighted graph, which is essential for maze solving. BFS is straightforward to implement and has a time complexity of $O(V + E)$, where $V$ is the number of vertices (cells) and $E$ is the number of edges (connections between cells). This efficiency makes it suitable for mazes of any size.

## Conclusion

The combination of the recursive division algorithm for maze generation and the BFS algorithm for maze resolution provides an effective solution for creating and solving mazes. The generation algorithm produces mazes with interesting patterns and ensures connectivity, while the resolution algorithm efficiently finds the shortest path from entrance to exit.