INTERNATIONAL UNIVERSITY
VIETNAM NATIONAL UNIVERSITY, HCM CITY

School of Computer Science & Engineering

2048 GAME PROJECT

Advisor: Tran Thanh Tung

Course: Algorithms and Data Structures

Student:

Tran Khanh Tai
ITITIU21300

Contents

List of Tables

List of Code Examples

1 Introduction

The 2048 game is a popular single-player sliding tile puzzle game developed by Gabriele Cirulli in March 2014. The game involves sliding numbered tiles on a grid to combine them, with the goal of creating a tile with the number 2048. This project implements the game using Java, focusing on applying data structures and algorithms concepts to create an efficient and functional game.

This report details the implementation of the 2048 game with specific emphasis on the data structures used, algorithms implemented, and their complexity analysis. It also explores how object-oriented programming principles have been applied to make the code modular, reusable, and maintainable.

1.1 Game Overview and History

The 2048 game was inspired by earlier games like Threes! and 1024. Gabriele Cirulli created it as a weekend project but it quickly went viral, with millions of players worldwide within weeks of its release. The game's appeal comes from its simple mechanics combined with strategic depth.

1.2 Project Scope

This project focuses on creating a fully-functional implementation of the 2048 game with the following features:

- Complete game mechanics matching the original game.
- Graphical user interface with smooth animations.
- Score tracking and high score persistence.
- Game state management (win/loss conditions).
- Performance optimization for smooth gameplay.

2 Importance of Data Structures and Algorithms

Data structures and algorithms form the foundation of efficient programming. In game development:

- **Data Structures** provide organized ways to store and access data, which is critical for representing the game board and state.
- **Algorithms** enable efficient manipulation of these data structures, allowing for game logic implementation.

2.1 Data Structure Selection Rationale

For the 2048 game implementation, several data structures were considered:

Table 1: Data Structure Comparison for Game Board Representation

Data Structure	Advantages	Disadvantages
2D Array	Direct index access (O(1)), Simple to implement, Memory efficient	Fixed size, No built-in methods
ArrayList of ArrayLists	Dynamic size, Built-in methods	Slightly less efficient for index access
HashMap with coordinate keys	Flexible for sparse boards, Good for larger boards	Overhead for simple 4x4 board, More complex implementation

After analysis, a 2D array was selected for the game board representation due to its efficiency and simplicity for the fixed-size 4x4 grid used in 2048.

3 Purpose of the Project

The main objectives of this project include:

- Implementing a fully functional 2048 game with a graphical user interface.
- Demonstrating practical application of data structures and algorithms.
- Applying object-oriented design principles to create maintainable code.
- Analyzing algorithm efficiency and performance in a real-world application.
- Developing problem-solving skills through game logic implementation.

4 Application of DSA Principles in the Game

The 2048 game implementation incorporates several key DSA principles:

4.1 Data Structures

- **2D Arrays** - Used to represent the game board grid, providing O(1) access time to any cell.
- **ArrayLists** - Used for managing dynamic collections such as empty cells for random tile placement.
- **Classes and Objects** - Used to encapsulate game components (Tile, Board, Game).

- **Queue** - Used in the animation system for sequential processing of visual effects.
- **Stack** - Used for implementing undo functionality by storing previous game states - allows players to revert moves.
- **HashMap** - Used for caching resources like images and colors for efficient retrieval.

4.2 Algorithms

- **Merge Algorithm** - For combining tiles with the same value.
- **Random Tile Generation** - For placing new tiles on the board using probability distribution.
- **Win/Loss Detection** - For determining game state by analyzing available moves.
- **Tile Movement Algorithms** - For handling directional moves (up, down, left, right).
- **Backtracking Algorithm** - For implementing an auto-solver feature (in developed).

5 Properties of the 2048 Game

5.1 Goal of the Game

The primary objective of the 2048 game is to slide numbered tiles on a 4×4 grid to combine them and create a tile with the number 2048. If the player successfully creates this tile, they win the game. The game also tracks the player's score based on the values of merged tiles.

5.2 Rules of the Game

- The game starts with two randomly placed tiles (either 2 or 4) on a 4×4 grid.
- The player can slide tiles in four directions: up, down, left, and right.
- When two tiles with the same number touch during a move, they merge into one tile with the sum of their values.
- After each move, a new tile (either 2 or 4) appears at a random empty position.
- The game ends when no valid moves are possible (board is full with no possible merges).
- The player wins when a tile with the value 2048 appears on the board.

6 Methodology

6.1 Overview of Classes Used

The game implementation follows an object-oriented approach with several key classes:

- **Main** - The entry point of the game application.
- **GameLogic** - The main class that controls game flow.
- **Board** - Represents the game board and its state.
- **GamePanel** - Handles the visual representation and user input.
- **ScoreManager** - Tracks and updates the game score.
- **TileAnimation** - Handles smooth visual transitions.
- **PrevState** - Represents a previous game state for undo functionality.
- **Constants** - Contains constant values used throughout the game.
- **WelcomePanel** - Displays the welcome screen and instructions.
- **GameBoard** - Represents the game board with controls for tile movement.
- **Game2048** - Holds the buttons and manages the game state.

6.2 Class Dependencies and Relationships

- The **GameLogic** class contains a 2D integer array grid and uses **ScoreManager** for score operations.
- The **GameLogic** class maintains a **Stack<PrevState>** for undo functionality.
- The **GameBoard** class (UI) contains and manages a **GameLogic** instance.
- The **GameBoard** class also contains a separate **Board** object and **AI** instance.
- The **GameBoard** class uses **ScoreManager** for score tracking and display.
- The **Game2048** class (main game window) contains a **GameBoard** instance.
- The **WelcomePanel2048** class receives a **ScoreManager** instance and creates **Game2048**.
- The **Main** class creates and initializes **ScoreManager** and **WelcomePanel2048**.
- The **TileAnimation** class is used by **GameBoard** for visual effects.
- The **AI** class implements **AIStrategy** interface and uses **ScoreManager**.
- The **ScoreManager** implements **ScoreService** interface for score persistence.

- The **PrevState** class is used by **GameLogic** to store previous game states.
- All classes reference **Constants** for configuration values like colors, sizes, and game parameters.

6.3 Main Classes and Functionality

6.3.1 Board Class

Code Example 1: Board Class Implementation with Grid Management

This class represents the game board as a 2D array of tiles:

```
public class Board {
    private int[][] grid;
    private static final int SIZE = Constants.SIZE;
    public GameLogic gameLogic;

    public Board() {
        grid = new int[SIZE][SIZE];
    }

    public int[][] getGrid() {
        int[][] copy = new int[SIZE][SIZE];
        for (int i = 0; i < SIZE; i++) {
            System.arraycopy(grid[i], 0, copy[i], 0, SIZE);
        }
        return copy;
    }
}
```

Listing 1: Board Class Implementation

The **Board** class has the following time complexities:

- Initialization: $O(n^2)$ where n is the board size (4×4)
- getGrid() method: $O(n^2)$ for creating a deep copy of the grid

Note: The Board class is a simple data container. Complex operations like checking for available moves and board state validation are handled by the **GameLogic** class.

6.3.2 GameLogic Class

Code Example 2: GameLogic Class Implementation with Game State Management

The GameLogic class manages the core game mechanics, board state, and move operations:

```java
public class GameLogic {
    private int[][] grid;
    private static final int SIZE = 4; // size of the grid
    private Stack<PrevState> undoStack; // stack for undo
       functionality
    private int score = 0;
    private ScoreManager scoreManager;

    public GameLogic() {
        grid = new int[SIZE][SIZE];
        undoStack = new Stack<>();
        score = 0;
        scoreManager = new ScoreManager();
        scoreManager.getBestScore();
    }

    public void initialize() {
        grid = new int[SIZE][SIZE];
        score = 0;
        if (scoreManager == null) {
            scoreManager = new ScoreManager();
        }
        scoreManager.getBestScore();
        addRandomTile();
        addRandomTile();
        undoStack.clear();
    }

    public int[][] getGrid() {
        int[][] copy = new int[SIZE][SIZE];
        for (int i = 0; i < SIZE; i++) {
            System.arraycopy(grid[i], 0, copy[i], 0, SIZE);
        }
        return copy;
    }

    public void move(String direction) {
        saveState(); // save current state before making a move
        int[][] before = new int[SIZE][SIZE];
        for (int i = 0; i < SIZE; i++) {
            System.arraycopy(grid[i], 0, before[i], 0, SIZE);
        }

        boolean canMove = false;
        switch (direction.toLowerCase()) {
            case "up":
                canMove = canMoveUp();
                if (canMove) moveUp();
                break;
            case "down":
```

```
50                  canMove = canMoveDown();
51                  if (canMove) moveDown();
52                  break;
53              case "left":
54                  canMove = canMoveLeft();
55                  if (canMove) moveLeft();
56                  break;
57              case "right":
58                  canMove = canMoveRight();
59                  if (canMove) moveRight();
60                  break;
61          }
62
63          if (canMove && !isSameGrid(before, grid)) {
64              addRandomTile();
65          }
66      }
67
68      public void saveState() {
69          if (canMoveUp() || canMoveDown() || canMoveLeft() ||
                canMoveRight()) {
70              int[][] gridCopy = new int[SIZE][SIZE];
71              for (int i = 0; i < SIZE; i++) {
72                  System.arraycopy(grid[i], 0, gridCopy[i], 0, SIZE
                        );
73              }
74              undoStack.push(new PrevState(gridCopy, score));
75          }
76      }
77
78      public void undo() {
79          if (!undoStack.isEmpty()) {
80              PrevState prevState = undoStack.pop();
81              grid = new int[SIZE][SIZE];
82              for (int i = 0; i < SIZE; i++) {
83                  System.arraycopy(prevState.grid[i], 0, grid[i],
                        0, SIZE);
84              }
85              score = prevState.score;
86          }
87      }
88
89      public void addRandomTile() {
90          Random random = new Random();
91          int emptyCount = 0;
92
93          // Count empty tiles
94          for (int i = 0; i < SIZE; i++) {
95              for (int j = 0; j < SIZE; j++) {
96                  if (grid[i][j] == 0) {
97                      emptyCount++;
```

```
98                  }
99              }
100         }
101
102         if (emptyCount == 0) return;
103
104         int pos = random.nextInt(emptyCount);
105         int k = 0;
106         for (int i = 0; i < SIZE; i++) {
107             for (int j = 0; j < SIZE; j++) {
108                 if (grid[i][j] == 0) {
109                     if (k == pos) {
110                         grid[i][j] = (random.nextInt(10) < 9) ? 2
                                : 4;
111                         return;
112                     }
113                     k++;
114                 }
115             }
116         }
117     }
118
119     public boolean isGameOver() {
120         for (int row = 0; row < SIZE; row++) {
121             for (int col = 0; col < SIZE; col++) {
122                 if (grid[row][col] == 0) {
123                     return false;
124                 }
125                 if (row < SIZE - 1 && grid[row][col] == grid[row
                    + 1][col]) {
126                     return false;
127                 }
128                 if (col < SIZE - 1 && grid[row][col] == grid[row
                    ][col + 1]) {
129                     return false;
130                 }
131             }
132         }
133         return true;
134     }
135
136     public boolean isWin() {
137         for (int row = 0; row < SIZE; row++) {
138             for (int col = 0; col < SIZE; col++) {
139                 if (grid[row][col] == 2048) {
140                     return true;
141                 }
142             }
143         }
144         return false;
145     }
```

```
146
147     // Additional methods for move validation and execution
148     private boolean canMoveUp() {
149         for (int col = 0; col < SIZE; col++) {
150             for (int row = 1; row < SIZE; row++) {
151                 if (grid[row][col] != 0 &&
152                     (grid[row - 1][col] == 0 || grid[row - 1][col
                        ] == grid[row][col])) {
153                     return true;
154                 }
155             }
156         }
157         return false;
158     }
159
160     private boolean canMoveDown() {
161         for (int col = 0; col < SIZE; col++) {
162             for (int row = SIZE - 2; row >= 0; row--) {
163                 if (grid[row][col] != 0 &&
164                     (grid[row + 1][col] == 0 || grid[row + 1][col
                        ] == grid[row][col])) {
165                     return true;
166                 }
167             }
168         }
169         return false;
170     }
171
172     private boolean canMoveLeft() {
173         for (int row = 0; row < SIZE; row++) {
174             for (int col = 1; col < SIZE; col++) {
175                 if (grid[row][col] != 0 &&
176                     (grid[row][col - 1] == 0 || grid[row][col -
                        1] == grid[row][col])) {
177                     return true;
178                 }
179             }
180         }
181         return false;
182     }
183
184     private boolean canMoveRight() {
185         for (int row = 0; row < SIZE; row++) {
186             for (int col = SIZE - 2; col >= 0; col--) {
187                 if (grid[row][col] != 0 &&
188                     (grid[row][col + 1] == 0 || grid[row][col +
                        1] == grid[row][col])) {
189                     return true;
190                 }
191             }
192         }
```

```
193             return false;
194         }
195     }
```

Listing 2: GameLogic Class Implementation

Time Complexity Analysis:

- **getGrid()**: $O(n^2)$ - Creates a deep copy of the 4×4 grid
- **move()**: $O(n^2)$ - Processes each cell for movement and merging operations
- **addRandomTile()**: $O(n^2)$ - Scans the grid twice: once to count empty cells, once to place the tile
- **isGameOver()**: $O(n^2)$ - Checks each cell and its adjacent cells for possible moves
- **isWin()**: $O(n^2)$ worst case - Scans all cells looking for 2048 tile
- **saveState()**: $O(n^2)$ - Creates a deep copy of the grid for undo functionality
- **undo()**: $O(n^2)$ - Restores the grid from the previous state
- **canMove methods**: $O(n^2)$ - Each scans the grid to check for valid moves

Space Complexity Analysis:

- **Grid storage**: $O(n^2)$ - Main 4×4 integer array
- **Undo stack**: $O(k \times n^2)$ where k is the number of saved states
- **Temporary arrays**: $O(n^2)$ - For grid copying operations in move() and saveState()
- **Overall space complexity**: $O(k \times n^2)$ due to the undo stack storing multiple game states

6.3.3 PrevState Class

Code Example 3: State Management for Undo Functionality

The PrevState class stores previous game states for the undo feature:

```
public class PrevState {
    private static final int SIZE = Constants.SIZE;
    int[][] grid;
    int score;

    PrevState(int[][] grid, int score) {
        this.grid = new int[SIZE][SIZE];
        for (int i = 0; i < SIZE; i++) {
            System.arraycopy(grid[i], 0, this.grid[i], 0, SIZE);
        }
        this.score = score;
```

```
    }
}
```

Listing 3: PrevState Class Implementation

Design Notes:

- The GameLogic class uses a **Stack<PrevState>** for implementing undo functionality
- String-based direction handling provides flexibility for different input methods
- The class separates move validation (canMove methods) from move execution for better modularity
- Deep copying ensures game state integrity during undo operations
- The random tile generation uses a 90% probability for value 2 and 10% for value 4, matching the original game

6.3.4 ScoreManager Class

Code Example 4: Score Management and Persistence

The ScoreManager class handles score tracking and persistence:

```
public class ScoreManager implements ScoreService {
    private static final Logger LOGGER = Logger.getLogger(
       ScoreManager.class.getName());
    private static final String SCORE_FILE = "scores.txt";
    private static final String BEST_SCORE_KEY = "BestScore";

    private int bestScore;
    private final Map<String, Integer> scores;
    private final String filePath;

    public ScoreManager() {
        this(SCORE_FILE);
    }

    public ScoreManager(String filePath) {
        this.filePath = filePath;
        scores = new HashMap<>();
        loadScores();
    }

    public void loadScores() {
        File file = new File(filePath);

        if (!file.exists()) {
            LOGGER.info("Score file not found, initializing with
               default values");
```

```
25              saveScores(); // Create the file with default values
26              return;
27          }
28
29          try (BufferedReader reader = new BufferedReader(new
               FileReader(file))) {
30              String line;
31              while ((line = reader.readLine()) != null) {
32                  String[] parts = line.split(":");
33                  if (parts.length == 2) {
34                      try {
35                          int value = Integer.parseInt(parts[1].
                               trim());
36                          scores.put(parts[0].trim(), value);
37                      } catch (NumberFormatException e) {
38                          LOGGER.log(Level.WARNING, "Invalid␣score␣
                               value:␣" + parts[1], e);
39                      }
40                  }
41              }
42
43              // Update the instance variables
44              bestScore = scores.getOrDefault(BEST_SCORE_KEY, 0);
45
46              LOGGER.info("Loaded␣scores:␣Best=" + bestScore);
47
48          } catch (IOException e) {
49              LOGGER.log(Level.SEVERE, "Failed␣to␣load␣scores␣from␣
                   file", e);
50          }
51      }
52
53      private void saveScores() {
54          // Update the map with current values
55          scores.put(BEST_SCORE_KEY, bestScore);
56
57          try (BufferedWriter writer = new BufferedWriter(new
               FileWriter(filePath))) {
58              for (Map.Entry<String, Integer> entry : scores.
                   entrySet()) {
59                  writer.write(entry.getKey() + ":" + entry.
                       getValue());
60                  writer.newLine();
61              }
62              LOGGER.info("Scores␣saved␣successfully");
63          } catch (IOException e) {
64              LOGGER.log(Level.SEVERE, "Failed␣to␣save␣scores␣to␣
                   file", e);
65          }
66      }
67
```

```
68      public int getBestScore() {
69          // read best score from the file
70          bestScore = 0;
71          try {
72              File file = new File(filePath);
73              Scanner scanner = new Scanner(file);
74              while (scanner.hasNextLine()) {
75                  String line = scanner.nextLine();
76                  if (line.startsWith(BEST_SCORE_KEY + ":")) {
77                      String[] parts = line.split(":");
78                      if (parts.length == 2) {
79                          try {
80                              bestScore = Integer.parseInt(parts
                                    [1].trim());
81                              LOGGER.info("Best score loaded: " +
                                    bestScore);
82                          } catch (NumberFormatException e) {
83                              LOGGER.log(Level.WARNING, "Invalid
                                    best score value: " + parts[1], e)
                                    ;
84                          }
85                      }
86                  }
87              }
88          } catch (FileNotFoundException e) {
89              LOGGER.log(Level.SEVERE, "Score file not found", e);
90              e.printStackTrace();
91          }
92
93          return bestScore;
94      }
95
96      public boolean updateScore(int score) {
97          boolean isNewBest = false;
98
99          if (score > bestScore) {
100             bestScore = score;
101             isNewBest = true;
102             LOGGER.info("New best score: " + bestScore);
103         }
104
105         saveScores();
106         return isNewBest;
107     }
108 }
```

Listing 4: ScoreManager Class Implementation

This class implements the **ScoreService** interface for score management, providing methods to load, save, and update scores. It uses a text file for persistence, ensuring that scores are retained across game sessions.

6.3.5 UI and Animation System

The UI system consists of several key components:

- **GameBoard**: Handles rendering and user input
- **TileAnimation**: Manages smooth visual transitions
- **Game2048**: Main game window coordination

7 Algorithms

7.1 Move and Merge Algorithm

Code Example 5: Tile Movement and Merging Logic

The most complex algorithm in the game is the tile movement and merging logic, which uses a compress-merge-compress approach:

```
public void move(String direction) {
    saveState(); // save the current state before making a move
    int[][] before = new int[SIZE][SIZE];
    for (int i = 0; i < SIZE; i++) { // create a copy of the
       current grid
        System.arraycopy(grid[i], 0, before[i], 0, SIZE);
    }
    boolean canMove = false;
    switch (direction.toLowerCase()) {
        case "up":
            canMove = canMoveUp();
            if (canMove)
                moveUp();
            break;
        case "down":
            canMove = canMoveDown();
            if (canMove)
                moveDown();
            break;
        case "left":
            canMove = canMoveLeft();
            if (canMove)
                moveLeft();
            break;
        case "right":
            canMove = canMoveRight();
            if (canMove)
                moveRight();
            break;
        default:
            System.out.println("Invalid direction. Use up, down,
               left, or right.");
```

```
31      }
32      if (canMove && !isSameGrid(before, grid)) { // check if the
            grid has changed
33          addRandomTile(); // add a random tile if a move was made
34      }
35  }
36
37  // Example of moveLeft implementation
38  void moveLeft() {
39      for (int row = 0; row < SIZE; row++) {
40          int[] compressed = compressRowLeft(row);
41          mergeTiles(compressed);
42          setRowMoveLeft(row, compressed);
43      }
44  }
45
46  private int[] compressRowLeft(int row) {
47      int[] compressed = new int[SIZE];
48      int index = 0;
49      for (int col = 0; col < SIZE; col++) {
50          if (grid[row][col] != 0) {
51              compressed[index++] = grid[row][col];
52          }
53      }
54      return compressed;
55  }
56
57  private void mergeTiles(int[] line) {
58      // Step 1: Compress (slide non-zero tiles to the front)
59      int[] compressed = new int[SIZE];
60      int index = 0;
61      for (int value : line) {
62          if (value != 0) {
63              compressed[index++] = value;
64          }
65      }
66
67      // Step 2: Merge adjacent tiles with same value
68      for (int i = 0; i < SIZE - 1; i++) {
69          if (compressed[i] != 0 && compressed[i] == compressed[i +
                1]) {
70              compressed[i] *= 2;
71              score += compressed[i];
72              compressed[i + 1] = 0;
73              i++; // Skip next tile (already merged)
74          }
75      }
76
77      // Step 3: Compress again after merging
78      int[] merged = new int[SIZE];
79      index = 0;
```

```
80      for (int value : compressed) {
81          if (value != 0) {
82              merged[index++] = value;
83          }
84      }
85      System.arraycopy(merged, 0, line, 0, SIZE);
86  }
87
88  private void setRowMoveLeft(int row, int[] line) {
89      for (int col = 0; col < SIZE; col++) {
90          grid[row][col] = line[col];
91      }
92  }
```

Listing 5: Move and Merge Algorithm

Time Complexity Analysis:

- **move()**: O(n^2) - Main method that coordinates the move operation
- **moveLeft()**: O(n^2) - Processes each row (n rows, each with n elements)
- **compressRowLeft()**: O(n) - Single pass through one row
- **mergeTiles()**: O(n) - Three sequential passes through the array
- **setRowMoveLeft()**: O(n) - Single pass to update the grid row
- **Overall complexity**: O(n^2) where n is the board size (4×4 = 16 operations)

Space Complexity Analysis:

- **Grid storage**: O(n^2) - Main 4×4 integer array
- **Temporary arrays**: O(n) - compressed[ ], merged[ ], and line[ ] arrays for processing
- **Before grid copy**: O(n^2) - Copy of original grid for comparison
- **Overall space complexity**: O(n^2) dominated by the grid storage and grid copy

Algorithm Steps:

1. **Compress**: Move all non-zero tiles to one side, eliminating gaps
2. **Merge**: Combine adjacent tiles with the same value
3. **Compress Again**: Remove gaps created by merging
4. **Update Grid**: Apply the processed line back to the grid

Visualization of Move and Merge Process:

Initial State

Step 1: Compress (Remove Gaps)

Step 2: Merge Adjacent Same Values

Final State (Already Compressed)

Figure 1: Visualization of moveLeft() algorithm showing the three-step process: compress, merge, and final compress. Example shows row [4,0,2,2] becoming [4,4,0,0] with score increase of 4 points.

7.2 Algorithm Efficiency Analysis

The current implementation is already quite efficient for the 4×4 grid size typical of 2048:

Key Efficiency Features:

- **Single-pass processing**: Each row/column is processed exactly once per move
- **In-place operations**: The compress-merge-compress pattern minimizes memory allocation
- **Early termination**: canMove methods prevent unnecessary processing when no moves are possible
- **Linear complexity per line**: Each row/column operation is O(n) where n=4

Performance Characteristics:

- **Move operation**: O(16) = O(1) for constant 4×4 grid
- **Memory usage**: O(16) = O(1) for temporary arrays
- **Cache efficiency**: Sequential access patterns optimize CPU cache usage
- **Predictable timing**: Consistent performance regardless of tile distribution

Algorithm Strengths:

1. **Simplicity**: Easy to understand and maintain
2. **Correctness**: Three-step process ensures proper tile behavior
3. **Consistency**: Same pattern used for all four directions
4. **Efficiency**: Optimal for the fixed 4×4 grid size

7.3 Random Tile Generation

Code Example 6: Weighted Random Tile Generation

After each move, a new tile needs to be added to a random empty cell:

```
public void addRandomTile() {
    Random random = new Random();
    int emptyCount = 0;
    // check how many empty tiles are there
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (grid[i][j] == 0) {
                emptyCount++;
            }
        }
    }
    // if there are no empty tiles, return
    if (emptyCount == 0) {
        return;
    }
    int pos = random.nextInt(emptyCount);
    int k = 0;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (grid[i][j] == 0) {
                if (k == pos) {
                    grid[i][j] = (random.nextInt(10) < 9) ? 2 :
                       4; // 90% for 2, 10% for 4
                    return;
                }
                k++;
            }
        }
    }
}
```

Listing 6: Random Tile Generation

Time Complexity Analysis:

- Finding empty cells: $O(n^2)$

- Selecting a random cell: O(1)
- Overall complexity is $O(n^2)$

Probability Analysis: The probability distribution for new tiles follows the original game:

- 90% chance of generating a '2' tile
- 10% chance of generating a '4' tile

This distribution creates a balanced difficulty level, with occasional higher-value tiles appearing to help progression.

7.4 Game Over and Win Detection

Code Example 7: Comprehensive Game Over Detection Logic

The game over detection algorithm determines when no more moves are possible. The game ends when the board is full AND there are no possible merges in any direction:

```
// check if the game is over
public boolean isGameOver() {
    for (int row = 0; row < SIZE; row++) {
        for (int col = 0; col < SIZE; col++) {
            // If any cell is empty, game can continue
            if (grid[row][col] == 0) {
                return false;
            }
            // Check if current cell can merge with cell below
            if (row < SIZE - 1 && grid[row][col] == grid[row +
               1][col]) {
                return false;
            }
            // Check if current cell can merge with cell to the
               right
            if (col < SIZE - 1 && grid[row][col] == grid[row][col
                + 1]) {
                return false;
            }
        }
    }
    return true; // No empty cells and no possible merges
}

// check win condition
public boolean isWin() {
    for (int row = 0; row < SIZE; row++) {
        for (int col = 0; col < SIZE; col++) {
            if (grid[row][col] == 2048) {
                return true; // Win condition met
```

```
28                 }
29             }
30         }
31         return false; // No win condition met
32 }
```

Listing 7: Game Over and Win Detection

Algorithm Logic:

1. **Empty Cell Check**: Scan entire grid for any empty cells (value = 0)
2. **Vertical Merge Check**: For each cell, check if it can merge with the cell below
3. **Horizontal Merge Check**: For each cell, check if it can merge with the cell to the right
4. **Early Termination**: Return false immediately when any valid move is found
5. **Complete Scan**: Only return true if no empty cells or possible merges exist

Time Complexity Analysis:

- **isGameOver()**: $O(n^2)$ - Single pass through all cells with constant-time operations
- **isWin()**: $O(n^2)$ worst case - Scans until 2048 tile found or entire grid checked
- **canMove()**: $O(n^2)$ - Calls individual direction check methods
- **Best case**: O(1) - Early termination when first empty cell or merge is found
- **Worst case**: $O(n^2)$ - Full grid scan when no moves available
- **Average case**: $O(n^2/2)$ - Statistically finds result midway through scan

Space Complexity Analysis:

- **Auxiliary space**: O(1) - Only uses loop variables
- **Input space**: $O(n^2)$ - References the existing grid
- **Overall space complexity**: O(1) - No additional data structures needed

Table 2: Time Complexity Analysis of Key Operations

Operation	Time Complexity	Notes
Board Initialization	$O(n^2)$	n is the board size (typically 4)
Tile Movement	$O(n^2)$	Processing each cell for movement and merging
Game State Check	$O(n^2)$	Checking for win/loss conditions
Random Tile Generation	$O(n^2)$	Finding empty cells dominates
Rendering	$O(n^2)$	Drawing all tiles on the board

Table 3: Space Complexity Analysis

Component	Space Complexity	Notes
Game Board	$O(n^2)$	Storage for the grid
Animation System	$O(k)$	k is the number of active animations
Undo History	$O(m \cdot n^2)$	m is the number of stored moves
Image Cache	$O(1)$	Constant for the fixed set of tile images
Total Space	$O(m \cdot n^2 + k)$	Dominated by undo history for many moves

8 Implementation Challenges and Solutions

8.1 Directional Movement Consistency

Challenge: Ensuring tiles move consistently in all four directions without duplicate code.

Solution: Implemented a unified movement algorithm using direction vectors, allowing the same code to handle all four directions with appropriate transformations.

8.2 Animation System

Challenge: Creating smooth animations without blocking the game thread.

Solution: Developed a queue-based animation system that processes animations frame-by-frame using a timer, allowing for non-blocking animations.

8.3 Merge Logic Edge Cases

Challenge: Handling complex merge scenarios where multiple tiles could potentially merge in sequence.

Solution: Implemented a merge flag to prevent tiles from merging more than once per move, ensuring game balance and consistent behavior.

9 Results, Limitations, and Conclusion

The implementation of the 2048 game demonstrates practical application of data structures and algorithms in a real-world scenario. The key results include:

- A functional game with intuitive controls and visual feedback
- Efficient algorithms for tile movement, merging, and game state management
- Object-oriented design providing modularity and code reuse

9.1 Limitations

- The current implementation has $O(n^3)$ complexity for moves in the worst case
- Limited animation capabilities

Future improvements could include optimizing the move algorithm for better performance, and implementing more sophisticated animations.

9.2 Future Work

Potential enhancements for future versions include:

- AI solver using minimax or expectimax algorithms
- Using Ai to automatically play the game
- Enhanced graphics and animations
- Mobile-friendly touch interface
- Online leaderboards
- Customizable board sizes
- Alternate game modes (time attack, challenge mode)

9.3 Conclusion

This project successfully demonstrates the application of data structures and algorithms concepts in game development. The implementation of the 2048 game shows how careful algorithm design and appropriate data structure selection contribute to creating an efficient and enjoyable game experience.

10 GitHub Repository

The source code for this project is available at: `https://github.com/KhanhTaiTran/DSA-Project.git`