

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

1	Introduction	2
2	Importance of Data Structures and Algorithms (OOP)	2
3	Purpose of the Project	2
4	Application of DSA Principles in the Game	3
4.1	Data Structures	3
4.2	Algorithms	3
5	Properties of the 2048 Game	3
5.1	Goal of the Game	3
5.2	Rules of the Game	3
6	Methodology	4
6.1	Overview of Classes Used	4
6.2	Class Dependencies and Relationships	4
6.3	Main Classes and Functionality	4
6.3.1	Board Class	4
6.3.2	Tile Class	5
7	Algorithms	5
7.1	Move and Merge Algorithm	5
7.2	Random Tile Generation	7
7.3	Game Over Detection	8
7.4	Additional Behaviors and Image Handling	9
7.4.1	Centralized Image Loading	9
7.5	GamePanel Class	9
7.5.1	Directional Control and Image Updates	11
8	Results, Limitations, and Conclusion	11
8.1	Limitations	11
8.2	Conclusion	12
9	GitHub Repository	12

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.

2 Importance of Data Structures and Algorithms (OOP)

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.
- **Object-Oriented Programming (OOP)** facilitates modularity, encapsulation, and code reuse through class hierarchies.

The 2048 game benefits from OOP principles through:

- **Encapsulation** - hiding implementation details within classes
- **Inheritance** - extending functionality through class hierarchies
- **Polymorphism** - allowing for flexible behavior implementation
- **Abstraction** - simplifying complex systems into manageable components

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
- **ArrayLists** - Used for managing dynamic collections of game elements
- **Classes and Objects** - Used to encapsulate game components
- **Queue** - Used in the animation system for sequential processing

4.2 Algorithms

- **Merge Algorithm** - For combining tiles with the same value
- **Random Tile Generation** - For placing new tiles on the board
- **Win/Loss Detection** - For determining game state
- **Tile Movement Algorithms** - For handling directional moves

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. The player can continue beyond this goal to achieve higher-numbered tiles and a higher score.

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:

- **Game** - The main class that controls game flow
- **Board** - Represents the game board and its state
- **Tile** - Represents individual tiles with their values
- **GamePanel** - Handles the visual representation and user input
- **Direction** - Enum representing the four possible move directions
- **ScoreManager** - Tracks and updates the game score
- **Animation** - Handles smooth visual transitions

6.2 Class Dependencies and Relationships

- The **Game** class contains a **Board** object
- The **Board** class contains a 2D array of **Tile** objects
- The **GamePanel** class renders the **Board** and **Tile** objects
- The **ScoreManager** is updated by the **Game** class when tiles merge
- The **Animation** class works with the **GamePanel** to provide visual feedback

6.3 Main Classes and Functionality

6.3.1 Board Class

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

```
1 public class Board {
2     private Tile[][] grid;
3     private final int SIZE = 4;
4
5     public Board() {
6         grid = new Tile[SIZE][SIZE];
7         // Initialize empty board
8         for (int row = 0; row < SIZE; row++) {
9             for (int col = 0; col < SIZE; col++) {
10                 grid[row][col] = null;
11             }
12         }
13     }
14 }
```

```
13         addRandomTile(); // Add initial tiles
14         addRandomTile();
15     }
16
17     // More methods...
18 }
```

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)
- Checking for available moves: $O(n^2)$
- Board state checking: $O(n^2)$

6.3.2 Tile Class

Represents individual tiles on the board with values and positions:

```
1 public class Tile {
2     private int value;
3     private int row;
4     private int col;
5     private boolean merged;
6
7     public Tile(int value, int row, int col) {
8         this.value = value;
9         this.row = row;
10        this.col = col;
11        this.merged = false;
12    }
13
14    // Getters and setters
15 }
```

Listing 2: Tile Class Implementation

7 Algorithms

7.1 Move and Merge Algorithm

The most complex algorithm in the game is the tile movement and merging logic:

```
1 public boolean move(Direction dir) {
2     boolean moved = false;
3     resetMergeStatus();
4
5     switch (dir) {
```

```

6         case UP:
7             for (int col = 0; col < SIZE; col++) {
8                 for (int row = 1; row < SIZE; row++) {
9                     if (grid[row][col] != null) {
10                         moved |= moveUp(row, col);
11                     }
12                 }
13             }
14             break;
15             // Cases for DOWN, LEFT, RIGHT similarly implemented
16         }
17
18         if (moved) {
19             addRandomTile();
20             checkGameOver();
21         }
22
23         return moved;
24     }
25
26     private boolean moveUp(int row, int col) {
27         boolean moved = false;
28         Tile tile = grid[row][col];
29         int newRow = row;
30
31         // Move up as far as possible
32         while (newRow > 0 && grid[newRow-1][col] == null) {
33             newRow--;
34         }
35
36         // Check for merge
37         if (newRow > 0 && grid[newRow-1][col] != null &&
38             grid[newRow-1][col].getValue() == tile.getValue() &&
39             !grid[newRow-1][col].isMerged()) {
40
41             // Merge with the tile above
42             int mergedValue = tile.getValue() * 2;
43             grid[newRow-1][col].setValue(mergedValue);
44             grid[newRow-1][col].setMerged(true);
45             grid[row][col] = null;
46             score += mergedValue;
47             moved = true;
48         }
49         // Just move without merging
50         else if (newRow != row) {
51             grid[newRow][col] = tile;
52             grid[row][col] = null;
53             moved = true;
54         }
55
56         return moved;

```

57 }

Listing 3: Move and Merge Algorithm

Time Complexity Analysis:

- Move operation: $O(n^2)$ where n is the board size
- For each direction, we need to check each cell and potentially move it
- The while loop for moving tiles runs at most $O(n)$ times
- Overall complexity is $O(n^3)$ in the worst case

7.2 Random Tile Generation

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

```

1 private void addRandomTile() {
2     List<int[]> emptyCells = new ArrayList<>();
3
4     // Find all empty cells
5     for (int row = 0; row < SIZE; row++) {
6         for (int col = 0; col < SIZE; col++) {
7             if (grid[row][col] == null) {
8                 emptyCells.add(new int[]{row, col});
9             }
10        }
11    }
12
13    if (!emptyCells.isEmpty()) {
14        int index = (int)(Math.random() * emptyCells.size());
15        int[] position = emptyCells.get(index);
16        int value = Math.random() < 0.9 ? 2 : 4; // 90% chance
17            // for a 2, 10% for a 4
18
19        grid[position[0]][position[1]] = new Tile(value, position
20            [0], position[1]);
21    }
22 }
```

Listing 4: Random Tile Generation

Time Complexity Analysis:

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

7.3 Game Over Detection

The game ends when there are no valid moves left:

```

1 private boolean isGameOver() {
2     // Check for empty cells
3     for (int row = 0; row < SIZE; row++) {
4         for (int col = 0; col < SIZE; col++) {
5             if (grid[row][col] == null) {
6                 return false; // Game continues if there's an
                             // empty cell
7             }
8         }
9     }
10
11    // Check for possible merges horizontally
12    for (int row = 0; row < SIZE; row++) {
13        for (int col = 0; col < SIZE - 1; col++) {
14            if (grid[row][col].getValue() == grid[row][col+1].
15                getValue()) {
16                return false; // Game continues if a horizontal
                             // merge is possible
17            }
18        }
19    }
20
21    // Check for possible merges vertically    // Check for
    // possible merges vertically
22    for (int col = 0; col < SIZE; col++) {{
23        for (int row = 0; row < SIZE - 1; row++) {
24            if (grid[row][col].getValue() == grid[row+1][col].
25                getValue()) {
26                if (grid[row][col].
27                    getValue() == grid[row+1][col].getValue()) {
28                    return false; // Game continues if a vertical
                             // merge is possible
29                }
30            }
31        }
32    }}
33
34    return true; // No moves left, game overty cells and no
    // possible merges
35 }

```

Listing 5: Game Over Detection

Time Complexity Analysis:

- Checking for empty cells: $O(n^2)$
- Checking for possible merges: $O(n^2)$ horizontal merges: $O(n^2)$
- Overall complexity is $O(n^2)$ merges: $O(n^2)$

$O(n^2)$

7.4 Additional Behaviors and Image Handling

ectionAdditional Behaviors and Image Handling

7.4.1 Centralized Image Loading

The game uses a resource manager to efficiently load and cache images:

```

1 public class ResourceManager
2     private static Map<String, BufferedImage> images = new
        HashMap<>(); {
3     HashMap<>();
4     public static BufferedImage loadImage(String path) {
5         if (images.containsKey(path)) {
6             return images.get(path);
7         }
8
9         try {
10             BufferedImage image = ImageIO.read(new File(path));
11             images.put(path, image);mage = ImageIO.read(new File(
                path));
12             return image;
13         } catch (IOException e) {
14             e.printStackTrace();
15             return null;          e.printStackTrace();
16         }
17         return null;
18     }

```

Listing 6: Image Resource Manager

This implementation uses a HashMap for $O(1)$ image lookup, preventing redundant loading of the same resources. HashMap for $O(1)$ image lookup, preventing redundant loading of the same resources.

7.5 GamePanel Class

The GamePanel class handles rendering and user interaction: Panel class handles rendering and user interaction:

```

1 public class GamePanel extends JPanel
2     private Board board;
3     private Map<Integer, Color> tileColors;
4     ap<Integer, Color> tileColors;
5     public GamePanel(Board board) {
6         this.board = board;
7         initializeColors();
8         setFocusable(true);lors();

```

```

9      addKeyListener(new GameKeyAdapter());    setFocusable(true
10    );
11  }      addKeyListener(new GameKeyAdapter());
12
13  @Override
14  protected void paintComponent(Graphics g)
15      super.paintComponent(g);    protected void paintComponent(
16      Graphics g) {
17      Graphics2D g2d = (Graphics2D) g;intComponent(g);
18      Graphics2D g2d = (Graphics2D) g;
19      // Draw background
20      g2d.setColor(new Color(187, 173, 160));
21      g2d.fillRect(0, 0, getWidth(), getHeight());etColor(new
22      Color(187, 173, 160));
23      etHeight();
24      // Draw tiles
25      for (int row = 0; row < 4; row++) {// Draw tiles
26          for (int col = 0; col < 4; col++) {
27              drawTile(g2d, board.getTile(row, col), row, col)
28              {
29              };
30          }
31
32          // Draw score and game over message if needed
33          drawUI(g2d);and game over message if needed
34      }
35
36  private void drawTile(Graphics2D g2d, Tile tile, int row, int
37      col) {
38      // Tile drawing implementation {
39      }
40  }

```

Listing 7: GamePanel Rendering

```

1  private class GameKeyAdapter extends KeyAdapter {
2      @Override
3      public void keyPressed(KeyEvent e) {Override
4          boolean moved = false;    public void keyPressed(KeyEvent
5          e) {
6          moved = false;
7          switch (e.getKeyCode()) {
8              case KeyEvent.VK_UP:
9                  moved = board.move(Direction.UP);
10                 break;
11                 moved = board.move(
12                 Direction.UP);
13             case KeyEvent.VK_DOWN:
14                 moved = board.move(Direction.DOWN);
15                 case KeyEvent.VK_DOWN:
16                 break;
17             case KeyEvent.VK_LEFT:
18                 break;
19                 moved = board.move(Direction.LEFT);e KeyEvent.

```

```

15         VK_LEFT:
16             break;
17         case KeyEvent.VK_RIGHT:
18             moved = board.move(Direction.RIGHT);
19             break; moved = board.move(Direction.RIGHT);
20             break;
21         if (moved) {
22             repaint();
23         }
24     }
25 }
26 }

```

Listing 8: Key Event Handling

7.5.1 Directional Control and Image Updates

Each key press triggers a move in the corresponding direction, which may update the game state and cause the UI to refresh if tiles were moved or merged.

8 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

8.1 Limitations

- The current implementation has $O(n^3)$ complexity for moves in the worst case
- No undo functionality or game state saving
- Limited animation capabilities

Future improvements could include optimizing the move algorithm for better performance, adding game state saving functionality, and implementing more sophisticated animations.

8.2 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.

9 GitHub Repository

The source code for this project is available at: <https://github.com/yourusername/2048-game>