CS203 Project- 2048 Game

Team Members-

Yash- 202051205

Nishant- 202051127

Ankit Kumar Mishra- 202051028

Nakul Yadav- 202051124

Nagpure Atharva Sunil- 202051123

Content-

Problem Statement: *Creating 2048 game*

Information About Game

1. Introduction

2048 is an exciting tile-shifting game, where we move tiles around to combine them, aiming for increasingly larger tile values.

2. How to Play 2048

A game of 2048 is played on a 4×4 board generally. But we can change the dimensions accordingly. Each position on the board may be empty or may contain a tile, and each tile will have a number on it. When we start, the board will have two tiles in random locations, each of which either has a “2” or a “4” on it – each has an independent 10% chance of being a “4”, or otherwise a is a “2”. Moves are performed by shifting all tiles towards one edge – up, down, left, or right. When doing this, any tiles with the same value that are adjacent to each other and are moving together will merge and end up with a new tile equal to the sum of the earlier two: After we’ve made a move, a new tile will be placed onto the board. This is placed in a random location, and will be a “2” or a “4” in the same way as the initial tiles – “2” 90% of the time and “4” 10% of the time. The game then continues until there are no more moves possible. In general, the goal of the game is to reach a single tile with a value of “2048”. However, the game doesn’t stop here, and we can continue

playing as far as it is possible to go, aiming for the largest possible tile. In theory, this is a tile with the value “131,072”.

3. ***Problem Explanation***

Solving this game is an interesting problem because it has a random component. It’s impossible to correctly predict not only where each new tile will be placed, but whether it will be a “2” or a “4”. As such, it is impossible to have an algorithm that will correctly solve the puzzle every time. The best that we can do is determine what is likely to be the best move at each stage and play the probability game. At any point, there are only four possible moves that we can make. Sometimes, some of these moves have no impact on the board and are thus not worth making. The challenge is then to determine which of these four moves is going to be the one that has the best long-term outcome.

GAME PLAN

We will create these classes to complete our objective of creating 2048 game.

Game/Start class:

- Start class main method
- Game class extends JPanel
- GameLoop
- KeyListener
- Update/Render

KeyBoard class:

- Static class
- Previous and current keystate Boolean[]
- Gets set in Game class

Tile class:

- On create, draw image and store
- Keep track of animations
- Position offsetted from GameBoard image.

GameBoard class:

- Tile[][]
- Move method for combining and setting animation on tile
- Key input here
- Created and used in Game class
- Board image

DrawUtils/Point class:

- Width/height of message
- Point in row/col

Algorithms

1. Firstly, when the board image is drawn, we have to randomly spawn any two tiles and fill them with 2 or 4 randomly to begin the game.

startingTiles is a global variable to define how many tiles you want at the starting of the game, generally we want 2 tiles in the starting so, startingTiles=2

Start ()

1. for i = 0 to startingTiles
2. call spawRandom().
3. Exit

This method is just starting the game calling the spawnRandom method two times to add tiles in the tile matrix at the beginning of the game.

Tile complexity is same as that of spawnRandom method as it does not depend on the size of the tile matrix i.e., on row and col variables.

board [] [] is a global variable which is a 2D matrix containing tiles,

spawnRandom()

1. notValid = true
2. while(notValid)
3. Select row & col by random generator both less than 4.
4. Tile current = board[row][col].
5. if (current = null)
6. Randomly select 2 or 4 and store it in value.

7. Create new Tile object with value and coordinates getTileX(col) and getTileY(row).
8. board[row][col] = new Tile.
9. notValid = false.
10. Exit.

getTileX(col)

1. return the distance of the tile from the left end of the tile matrix.

getTileY(row)

1. return the distance of the tile from the upper end of the tile matrix.

Now the spawnRandom method explanation:

What is this method actually doing? This method when called will select a tile in the tile matrix randomly which is empty i.e., having 0 as string in it and assign value 2 with 90% probability and value 4 with 10% probability to the tile. Coming to the time complexity of this method:

1. --------- 1
2. --------- t
3. --------- t
4. --------- t
5. --------- t
6. --------- t
7. --------- (1 + time of getTileX + time of getTileY) * t
8. --------- t

9. ---------- t

10. ---------- 1

Time of getTileX = O(1) as only 1 return statement is there

Similarly time of getTileY = O(1)

So total time is = 1+t+t+t+t+t+3t+t+t+1 = 10t+2

Now calculate t:

How many times the while loop will run it will depend on the notValid variable which is changing inside the if structure so in the worst case the random number generator generates every possible tile except the last one which is empty so it will run n*n times in the worst case which is row * col.

So, t = O(n^2)

And hence, The time complexity of the spawnRandom method is O(n^2).

2. *move method:*

This method is moving tiles in the direction entered by the user and combine the tiles which can combine according to the rule that in the direction of movement of the tiles the two similar tiles combine to form a single tile having the value twice as before.

Input: row, col, horizontaldirection, verticaldirection, Direction dir

Output: False if the tile at the given position can't move, True if it is moved to new position.

move (row, col, horizontaldirection, verticaldirection, Direction dir)

1. canMove = false.
2. Create tile class object 'current' and initialize to board[row][col].
3. if (current = NULL) then return false.
4. Boolean move = true.
5. int newRow = row, newCol = col.
6. while(move)
7. newCol += horizontaldirection, newRow += verticaldirection.
8. if (checkOutOfBounds) then break.
9. if (board[newRow][newCol] = NULL):
10. store the old tile value to the new tile.
11. store null value to the old tile.
12. set the position of the new tile.
13. canMove = true.
14. else if (value at new tile = value at old tile and board[newRow][newCol]. canCombine()) then:

```
15.             board[newRow][newCol].setCanCombine(false).
16.             store value at new tile to twice of it.
17.             canMove = true.
18.             store null to the old tile.
19.             set the position of the new tile.
20.             add the new tile value in the score.
21.    else move = false.
22. return canMove.
```

Time complexity:

1. ----------1
2. ----------1
3. ----------1
4. ----------1
5. ----------1
6. ----------t
7. ----------t
8. ----------t
9. ----------t
10. ----------t
11. ----------t
12. ----------t
13. ----------t

14. ---------t

15. ---------t

16. ---------t

17. ---------t

18. ---------t

19. ---------t

20. ---------t

21. ---------t

22. ---------1

So, total complexity of this method is = 6 + 16t

Now t is the number of times while loop runs so in the worst case we have to move the tile from one end of the board to other so in that case the loop will execute n times.

So, the total time complexity is 6+16n

i.e., O(n)

CheckOutOfBounds method:

Input: Direction dir, row, col

Output: false or true depending on the cases.

1. if (dir = Left) then return col<0.

2. else if (dir = Right) then return col>COLS-1.

3. else if (dir = Up) then return row<0.

4. else if (dir = Down) then row>ROWS-1.

5. return false.

6. Exit.

This method is just checking whether our control is going out of the game board bound or not. It returns true if it goes out of game board bound and false when it is in the game board bound.

Time Complexity:

This method has only if structure and every statement is executing only once during the method execution.

So, the total time complexity of this method is O (1).

3. update method:

1. call checkKeys() method.
2. if (score >= highscore) then highscore = score.
3. for i=0 to ROWS.
4. for j=0 to COLS.
5. Create tile class object ‘current’ and store board[i][j] in it.
6. if (current = NULL) then continue.
7. Call resetPosition() method.
8. Exit.

This method is updating the highscore and updating the position of the tile after every input direction entered from the keyboard.

Time complexity:

1. ----------time taken by checkKeys() method lets say Tck.
2. ----------1
3. ----------n+1
4. ----------n*(n+1)
5. ----------n*n
6. ----------n*n
7. ----------n*n *time taken by resetPosition() method let say Trp.
8. ----------1

So total time is = Tck+1+n+1+n*(n+1) + 2n*n + Trp*n*n

= Tck + Trp* n^2 (all the lower order terms can be neglected)

= O(n^3) + O (1) * n^2 = O(n^3).

So, the total complexity of this method is O(n^3).

resetPosition method:

input = Tile current, row, col.

Output = void.

Tile class variable SLIDE_SPEED

1. if (current = NULL) then return.
2. x = getTileX (col).
3. y = getTileY (row).
4. distX = current.getX() – x.
5. distY = current.getY() – y.
6. if (absolute value of distX < SLIDE_SPEED) then
7. current.setX (current.getX() – distX).
8. if (absolute value of distY < SLIDE_SPEED) then
9. current.setY (current.getY() – distY).
10. if (distX < 0) then
11. set x of current to initial plus SLIDE_SPEED.
12. if (distY < 0) then
13. set y of current to initial plus SLIDE_SPEED.
14. if (distX > 0) then
15. set x of current to initial minus SLIDE_SPEED.
16. if (distY > 0) then

17. set y of current to initial minus SLIDE_SPEED.
18. Exit.

This method is sliding the tile by changing its coordinates continuously using the reference of SLIDE_SPEED which depicts the speed or by which factor the tile is to move and distX and distY helps to change the coordinates whether to increase or decrease the x or y coordinates. Now the time complexity of this method is:

As you can clearly see there is no looping involved in this method no recursive call and the method calls inside this resetPosition method are only getter and setters whose time complexity is of O (1). Since every statement will be executed only once when this method is called.

So, the overall time complexity of this method is O (1).

4. checkDead method:

1. for i=0 to ROWS
2. for j=0 to COLS
3. if (board[i][j] = NULL) then return.
4. if (checkSurroundingTiles(i, j, board[i][j]) then return
5. Call sethighscore() method.
6. Exit.

This method checks whether there are any other moves left in the game or not. Whether the game is over or there is still any possible direction move left in which the game can proceed.

Time complexity:

1. ----------n+1
2. ----------n*(n+1)
3. ---------- n*n
4. ---------- n*n * time taken by checkSurroundingTiles method
5. ----------time taken by sethighscore () method
6. ----------1

So, the total time is = n+1+n*(n+1) +n*n +n*n *Tcst + Tshs + 1

= n*n + n*n *Tcst = n*n + n*n *1 = O(n^2)

So, the overall time complexity of this method is O(n^2).

checkSurroundingTiles method:

input: row, col, Tile current.

Output: true or false.

1. if (row > 0) then
2. select check tile immediate left to the current tile.
3. if (check = NULL) then return true.
4. if (value at check = value at current) then return true.
5. if (row < ROWS-1) then
6. select check tile immediate right to the current tile.
7. if (check = NULL) then return true.
8. if (value at check = value at current) then return true.
9. if (col > 0) then
10. select check tile immediate upper to the current tile.
11. if (check = NULL) then return true.
12. if (value at check = value at current) then return true.
13. if (col < COLS-1) then
14. select check tile immediate down to the current tile.
15. if (check = NULL) then return true.
16. if (value at check = value at current) then return true.
17. return false.
18. Exit.

This method checks all the four-surrounding tile with respect to current tile which must not be empty or have value equal to the current tile value otherwise the game is not over yet. There are other possible moves left in it to continue the game.

Time complexity of this method is O (1) as it is not dependent on ROWS or COLS and doesn't contain any loop also so every statement will run exactly once during the execution of this method.

So, the overall time complexity of this method is O (1).

5. *moveTiles method:*

Input: Direction dir.

Output: void

1. canMove = false.
2. horizontaldirection = 0, verticaldirection = 0.
3. if (dir = left) then
4. horizontaldirection = -1.
5. for i=0 to ROWS
6. for j=0 to COLS
7. if (! canMove) then
8. canMove = call move method with arguments i, j, horizontaldirection, verticaldirection, Direction dir.
9. else
10. call move method with arguments i, j, horizontaldirection, verticaldirection, Direction dir.
11. if (dir = Right) then
12. horizontaldirection = +1.
13. for i=0 to ROWS
14. for j=COLS-1 to 0
15. if (! canMove) then
16. canMove = call move method with arguments i, j, horizontaldirection, verticaldirection, Direction dir.
17. else

```
18.                 call move method with arguments i, j,
horizontaldirection, verticaldirection, Direction dir.
19. if (dir = Up) then
20.     verticaldirection = -1.
21.     for i=0 to ROWS
22.         for j=0 to COLS
23.             if (! canMove) then
24.                 canMove = call move method with arguments i, j,
horizontaldirection, verticaldirection, Direction dir.
25.             else
26.                 call move method with arguments i, j,
horizontaldirection, verticaldirection, Direction dir.
27. if (dir = Down) then
28.     verticaldirection = +1.
29.     for i=ROWS-1 to 0
30.         for j=COLS-1 to 0
31.             if (! canMove) then
32.                 canMove = call move method with arguments i, j,
horizontaldirection, verticaldirection, Direction dir.
33.             else
34.                 call move method with arguments i, j,
horizontaldirection, verticaldirection, Direction dir.
35. else print dir is not valid.
36. for i=0 to ROWS
```

37. for j=0 to COLS
38. Create tile class object current and initialize it to board[i][j].
39. if (current = NULL) then continue.
40. set setCanCombine value of current to true.
41. if (canMove) then
42. Call spawnRandom method.
43. Call checkdead method.
44. Exit.

This method takes the direction in which we have to move all the tiles as entered by the user. Then it traverses every tile and call the move method which returns true if the tile has been moved and false if it didn't and store that value in canMove variable. Now this variable helps us to depict that whether the board changed or not after the keyboard input. Because if the board has changed then we have to add a new tile into it and also check whether there is any other possible move is left or not which is done in the last if statement of the method. In this method lines 3 to 10, 11 to 18, 19 to 26 and 27 to 34 are exactly similar the only change in them is they are running for different directions. Logic and algorithm are exactly same for these lines of group. After that we also have to set the setCanCombine value of every tile containing any number to true and no change on the rest.

Now, the time complexity:

1. ----------1
2. ----------1
3. ----------1
4. ----------1

5. ----------n+1

6. ----------n*(n+1)

7. ---------- n*n

8. ---------- n*n * time taken by move method call Tm

9. ---------- n*n

10. ---------- n*n * time taken by move method call Tm

for 11 to 18, 19 to 26 and 27 to 34 is exactly same as lines 3 to 10.

35. ----------1

36. ----------n+1

37. ----------n*(n+1)

38. ---------- n*n

39. ---------- n*n

40. ---------- n*n *time taken by setter setCanCombine

41. ----------1

42. ----------time taken by spawnRandom method call Tsr

43. ----------time taken by checkdead method call Tcd

44. ----------1

So, the total time complexity is

= 1+1+1+1+ 4*(n+1+n*(n+1) + 2n*n+2n*(n+1) *Tm) +1+n+1+n*(n+1) +2n*n +n*n *1 + 1 + Tsr + Tcd +1

Now, we have to find Tm, Tsr and Tcd which are already calculated above i.e., Tm = O (n) , Tsr = O (n^2) and Tcd = O (n^2)

So, time complexity of this method will be

= O (n^3) i.e., the highest degree in the polynomial function.

So, the time complexity of this method is O (n^3).

checkKeys method:

1. if (keyboard input is left) then call moveTiles (Direction Left).
2. if (keyboard input is right) then call moveTiles (Direction Right).
3. if (keyboard input is up) then call moveTiles (Direction Up).
4. if (keyboard input is down) then call moveTiles (Direction Down).
5. Exit.

This method is just calling the moveTiles method according to the input entered from the keyboard.

Time complexity:

1. ---------- time taken by moveTiles method call Tmt.
2. ---------- time taken by moveTiles method call Tmt.
3. ---------- time taken by moveTiles method call Tmt.
4. ---------- time taken by moveTiles method call Tmt.
5. ----------1

So, the time complexity will be 4*Tmt

And, Tmt = O(n^3)

So, the overall time complexity of this method is of O(n^3).

After analysing all the algorithms we have the time complexity of this game project is O (n^3).

Source code:

- ## Start class:

```
import javax.swing.JFrame;

public class start {
    public static void main(String[]args) {
        Game game = new Game();

        JFrame window = new JFrame("2048 game");
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setResizable(false);
        window.add(game);
        window.pack();
        window.setLocationRelativeTo(null); //set location of the screen to center
        window.setVisible(true);

        game.start();
    }
}
```

This class is the starting point of the game which creates the Game class object, creating the frame for the game and calling the start method of the game class to start the execution of the actual 2048 game.

- **Gameboard class:**

```
package Game;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics2D;
import java.awt.event.KeyEvent;
import java.awt.image.BufferedImage;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.util.Random;

public class GameBoard {

   public static final int ROWS = 4, COLS = 4;

   private final int startingTiles = 2;     // the number of tiles at the starting
   private Tile[][] board;                  // 2-d array for storing tiles at each location of the matrix
   private BufferedImage gameBoard,finalBoard; //
   private int x,y;                         // dimension of the game board
   private int score = 0, highscore = 0;    // To calculate score and high score
   private Font scoreFont;                  // to select font of the score

   private static int SPACING = 10;         // to have space between two tiles so that tiles don't overlap
   public static int BOARD_WIDTH = (COLS+1)*SPACING + COLS*Tile.WIDTH;     // Total width of the board
   public static int BOARD_HEIGHT = (ROWS+1)*SPACING + ROWS*Tile.HEIGHT;   // Total height of the board

   //Saving
   private String saveDataPath;
   private String fileName = "SaveData";

   public GameBoard(int x, int y) {
      try {
         saveDataPath =
GameBoard.class.getProtectionDomain().getCodeSource().getLocation().toURI().getPath();
      }
      catch(Exception e) {
         e.printStackTrace();
      }
      scoreFont = Game.main.deriveFont(32f);  // Size of the font of the score and high score
      this.x = x;
      this.y = y;
      board = new Tile[ROWS][COLS];          // creating the 2 d array
```

```
    gameBoard = new BufferedImage(BOARD_WIDTH,BOARD_HEIGHT,BufferedImage.TYPE_INT_RGB);
    finalBoard = new BufferedImage(BOARD_WIDTH,BOARD_HEIGHT,BufferedImage.TYPE_INT_ARGB);

    loadHighScore();
    createBoardImage();
    start();
}

private void createSaveData() {
    try {
        File file = new File(saveDataPath,fileName);

        FileWriter output = new FileWriter(file);
        BufferedWriter writer = new BufferedWriter(output);
        writer.write(""+0);
        writer.close();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

private void loadHighScore() {
    try {
        File f = new File(saveDataPath,fileName);
        if(!f.isFile()) {
            createSaveData();
        }
        BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(f)));
        highscore = Integer.parseInt(reader.readLine());
        reader.close();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}

private void setHighScore() {
    FileWriter output = null;
    try {
        File f = new File(saveDataPath,fileName);
        output = new FileWriter(f);
        BufferedWriter writer = new BufferedWriter(output);
            writer.write("" + highscore);
        writer.close();
    }
    catch(Exception e) {
        e.printStackTrace();
```

```
    }
}

private void createBoardImage() {
    Graphics2D g = (Graphics2D) gameBoard.getGraphics();
    g.setColor(Color.BLACK);
    g.fillRect(0, 0, BOARD_WIDTH, BOARD_HEIGHT);

    g.setColor(Color.LIGHT_GRAY);
    for(int row = 0;row<ROWS;row++) {
        for(int col = 0;col<COLS;col++) {
            int x = SPACING + SPACING*col + Tile.WIDTH*col;
            int y = SPACING + SPACING*row + Tile.HEIGHT*row;
            g.fillRoundRect(x, y, Tile.WIDTH, Tile.HEIGHT, Tile.ARC_WIDTH, Tile.ARC_HEIGHT);
        }
    }
}

private void start() {
    for(int i=0;i<startingTiles;i++) {
        spawnRandom();
    }
}

private void spawnRandom() {
    Random random = new Random();
    boolean notValid = true;

    while(notValid) {
        int row = random.nextInt(ROWS);
        int col = random.nextInt(COLS);
        Tile current = board[row][col];
        if(current == null) {
            int value = random.nextInt(10)<9?2:4;
            Tile tile = new Tile(value,getTileX(col),getTileY(row));
            board[row][col] = tile;
            notValid = false;
        }
    }
}

public int getTileX(int col) {
    return SPACING + col*Tile.WIDTH + col*SPACING;
}

public int getTileY(int row) {
    return SPACING + row*Tile.HEIGHT + row*SPACING;
}
```

```
    public void render(Graphics2D g) {
        Graphics2D g2d = (Graphics2D)finalBoard.getGraphics();
        g2d.drawImage(gameBoard, 0, 0, null);

        for(int row = 0;row<ROWS;row++) {
            for(int col = 0;col<COLS;col++) {
                Tile current = board[row][col];
                if(current == null) continue;
                current.render(g2d);
            }
        }

        g.drawImage(finalBoard, x, y, null);
        g2d.dispose();

        g.setColor(Color.GREEN);
        g.setFont(scoreFont);
        g.drawString("Score: " + score, Game.WIDTH/2 - DrawUtils.getMessageWidth("Score: "+score, scoreFont,
g)/2
                                , Game.HEIGHT - GameBoard.BOARD_HEIGHT-DrawUtils.getMessageHeight("Score:
"+score, scoreFont, g)*2-20);
        g.setColor(Color.RED);
        g.drawString("Best:" + highscore, Game.WIDTH-DrawUtils.getMessageWidth("Best: "+highscore,
scoreFont, g)-20, 40);
    }

    public void update() {
        checkKeys();

        if(score >= highscore) {
            highscore = score;
        }
        for(int row = 0;row<ROWS;row++) {
            for(int col=0;col<COLS;col++) {
                Tile current = board[row][col];
                if(current == null) continue;
                resetPosition(current ,row, col);
                if(current.getValue() == 2048) {
                }
            }
        }
    }

    private void resetPosition(Tile current,int row, int col) {
        if(current == null) return;

        int x = getTileX(col);
```

```
        int y = getTileY(row);

        int distX = current.getX() - x;
        int distY = current.getY() - y;

        if(Math.abs(distX) < Tile.SLIDE_SPEED) {
            current.setX(current.getX() - distX);
        }
        if(Math.abs(distY) < Tile.SLIDE_SPEED) {
            current.setY(current.getY() - distY);
        }

        if(distX < 0) {
            current.setX(current.getX() + Tile.SLIDE_SPEED);
        }
        if(distY < 0) {
            current.setY(current.getY() + Tile.SLIDE_SPEED);
        }
        if(distX > 0) {
            current.setX(current.getX() - Tile.SLIDE_SPEED);
        }
        if(distY > 0) {
            current.setY(current.getY() - Tile.SLIDE_SPEED);
        }
    }

    private boolean move(int row,int col,int horizontalDirection, int verticalDirection,Direction dir) {
        boolean canMove = false;
        Tile current = board[row][col];
        if(current == null) return false;
        boolean move = true;
        int newCol = col;
        int newRow = row;
        while(move) {
            newCol += horizontalDirection;
            newRow += verticalDirection;
            if(checkOutOfBounds(dir,newRow,newCol)) break;
            if(board[newRow][newCol] == null) {
                board[newRow][newCol] = current;
                board[newRow - verticalDirection][newCol - horizontalDirection] = null;
                board[newRow][newCol].setSlideTo(new Point(newRow,newCol));
                canMove = true;
            }
            else if(board[newRow][newCol].getValue() == current.getValue() &&
board[newRow][newCol].CanCombine()) {
                board[newRow][newCol].setCanCombine(false);
                board[newRow][newCol].setValue(board[newRow][newCol].getValue()*2);
                canMove = true;
```

```
            board[newRow -verticalDirection][newCol - horizontalDirection] = null;
            board[newRow][newCol].setSlideTo(new Point(newRow,newCol));
            score += board[newRow][newCol].getValue();
        }
        else {
            move = false;
        }
    }
    return canMove;
}

private boolean checkOutOfBounds(Direction dir, int row, int col) {
    if(dir == Direction.LEFT) {
        return col <0;
    }
    else if(dir == Direction.RIGHT) {
        return col>COLS-1;
    }
    else if(dir == Direction.UP) {
        return row < 0;
    }
    else if(dir == Direction.DOWN) {
        return row > ROWS-1;
    }
    return false;
}

private void moveTiles(Direction dir) {
    boolean canMove = false;
    int horizontalDirection = 0,verticalDirection = 0;

    if(dir == Direction.LEFT) {
        horizontalDirection = -1;
        for(int row = 0;row<ROWS;row++) {
            for(int col = 0;col<COLS;col++) {
                if(!canMove) {
                    canMove = move(row,col,horizontalDirection,verticalDirection,dir);
                }
                else move(row,col,horizontalDirection,verticalDirection,dir);
            }
        }
    }

    else if(dir == Direction.RIGHT) {
        horizontalDirection = 1;
        for(int row = 0;row<ROWS;row++) {
            for(int col = COLS-1;col>=0;col--) {
                if(!canMove) {
```

```
                canMove = move(row,col,horizontalDirection,verticalDirection,dir);
            }
            else move(row,col,horizontalDirection,verticalDirection,dir);
        }
    }
}

else if(dir == Direction.UP) {
    verticalDirection = -1;
    for(int row = 0;row<ROWS;row++) {
        for(int col = 0;col<COLS;col++) {
            if(!canMove) {
                canMove = move(row,col,horizontalDirection,verticalDirection,dir);
            }
            else move(row,col,horizontalDirection,verticalDirection,dir);
        }
    }
}

else if(dir == Direction.DOWN) {
    verticalDirection = 1;
    for(int row = ROWS-1;row>=0;row--) {
        for(int col = 0;col<COLS;col++) {
            if(!canMove) {
                canMove = move(row,col,horizontalDirection,verticalDirection,dir);
            }
            else move(row,col,horizontalDirection,verticalDirection,dir);
        }
    }
}

else {
    System.out.println(dir + " is not a valid direction");
}

for(int row = 0;row<ROWS;row++) {
    for(int col = 0;col<COLS;col++) {
        Tile current = board[row][col];
        if(current == null) continue;
        current.setCanCombine(true);
    }
}

if(canMove) {
    spawnRandom();
    checkDead();
}
}
```

```
private void checkDead() {
    for(int row = 0;row<ROWS;row++) {
        for(int col = 0;col<COLS;col++) {
            if(board[row][col]==null) return;
            if(checksurroundingTiles(row,col,board[row][col])) {
                return;
            }
        }
    }
    setHighScore();
}

private boolean checksurroundingTiles(int row, int col, Tile current) {
    if(row >0) {
        Tile check = board[row-1][col];
        if(check == null) return true;
        if(current.getValue() == check.getValue()) return true;
    }
    if(row<ROWS-1) {
        Tile check = board[row+1][col];
        if(check == null) return true;
        if(current.getValue() == check.getValue()) return true;
    }
    if(col >0) {
        Tile check = board[row][col-1];
        if(check == null) return true;
        if(current.getValue() == check.getValue()) return true;
    }
    if(col<COLS-1) {
        Tile check = board[row][col+1];
        if(check == null) return true;
        if(current.getValue() == check.getValue()) return true;
    }
    return false;
}

private void checkKeys() {
    if(Keyboard.typed(KeyEvent.VK_LEFT)) {
        moveTiles(Direction.LEFT);
    }
    if(Keyboard.typed(KeyEvent.VK_RIGHT)) {
        moveTiles(Direction.RIGHT);
    }
    if(Keyboard.typed(KeyEvent.VK_UP)) {
        moveTiles(Direction.UP);
    }
    if(Keyboard.typed(KeyEvent.VK_DOWN)) {
```

```
            moveTiles(Direction.DOWN);
        }
    }
}
```

This is the main class in this project as all the game logic is in this class i.e., adding new tile to random position, adding value to new tiles, move tiles according to the direction entered, merge tiles with same numbers, move tiles to new position, manage the board, calculate the score and take care of all the rest game rules. This is the biggest class including all the algorithms used in this project. And this class provides the running time of this project.

- **Game class:**

```
package Game;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics2D;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.image.BufferedImage;

import javax.swing.JPanel;

public class Game extends JPanel implements KeyListener, Runnable{

   private static final long serialVersionUID = 1L;
   public static final int WIDTH = 600; //width of the screen
   public static final int HEIGHT = 700; // height of the screen
   public static final Font main = new Font("Bebas Neue Regular ",Font.PLAIN,30);
   private Thread game;
   private boolean running;
   private BufferedImage Image = new BufferedImage(WIDTH,HEIGHT,BufferedImage.TYPE_INT_RGB);
   private GameBoard board;

   public Game() {
      setFocusable(true);
      setPreferredSize(new Dimension(WIDTH,HEIGHT));
      addKeyListener(this);
      board = new GameBoard(WIDTH/2 - GameBoard.BOARD_WIDTH/2,HEIGHT - GameBoard.BOARD_HEIGHT -
20);
   }

   private void update() {
      board.update();
      Keyboard.update();
   }

   private void render() {
      Graphics2D g = (Graphics2D)Image.getGraphics();
      g.setColor(Color.WHITE);
      g.fillRect(0, 0, WIDTH, HEIGHT);
      board.render(g);
      g.dispose();

      Graphics2D g2d = (Graphics2D)getGraphics();
      g2d.drawImage(Image,0,0,null);
      g2d.dispose();
```

```
}

@Override
public void run() {
    long fpsTimer = System.currentTimeMillis();
    double nsPerUpdate = 1000000000.0/60;

    //last update time in nanoseconds
    double then = System.nanoTime();
    double unprocessed = 0;

    while(running) {
        boolean shouldrender = false;
        double now = System.nanoTime();
        unprocessed += (now - then)/nsPerUpdate;
        then = now;

        //update queue
        while(unprocessed >= 1) {
            update();
            unprocessed--;
            shouldrender = true;
        }

        //render
        if(shouldrender) {
            render();
            shouldrender = false;
        }
        else {
            try {
                Thread.sleep(1);
            }catch(Exception e) {
                e.printStackTrace();
            }
        }
    }

    //fps Timer
    if(System.currentTimeMillis() - fpsTimer > 1000) {
        fpsTimer += 1000;
    }
}

public synchronized void start() {
    if(running) return;
    running = true;
    game = new Thread(this,"game");
```

```
        game.start();
    }

    @Override
    public void keyTyped(KeyEvent e) { }

    @Override
    public void keyPressed(KeyEvent e) {
        Keyboard.keyPressed(e);
    }

    @Override
    public void keyReleased(KeyEvent e) {
        Keyboard.keyReleased(e);

    }
}
```

This is a class which takes care of all the running times involved in the game. It forms the background image of the game on the frame formed by the start class. And this is the driver class which calls the gameboard class object and its methods to execute them on the scheduled time.

- **Tile class:**

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;

public class Tile {

    public static final int WIDTH = 100;
    public static final int HEIGHT = 100;
    public static final int SLIDE_SPEED = 50;  // the speed at which the tiles slide
    public static final int ARC_WIDTH = 50;  // the length of the arc from width side
    public static final int ARC_HEIGHT = 50;  // the length of the arc from height side

    private int value;                // value at any tile
    private BufferedImage tileImage;    // object which holds the image
    private Color background;          // background color of the tile
    private Color text;                // color of the text on the tile
    private Font font;                 // font of the text on the tile
    private Point slideTo;
    private int x,y;                  // position of the tile

    private boolean canCombine = true;

    public Tile(int value,int x,int y) {
        this.value = value;
        this.x = x;
        this.y = y;
        slideTo = new Point(x,y);
        tileImage = new BufferedImage(WIDTH,HEIGHT,BufferedImage.TYPE_INT_RGB);
        drawImage();
    }

    private void drawImage() {

        Graphics2D g = (Graphics2D)tileImage.getGraphics();
        if(value == 2){
            background = new Color(0xe9e9e9);
            text = new Color(0x000000);
        }
        else if(value == 4) {
            background = new Color(0x004563);
            text = new Color(0x000736);
        }
        else if(value == 8) {
            background = new Color(0xffb266);
            text = new Color(0x000000);
```

```
    }
    else if(value == 16) {
        background = new Color(0xff8000);
        text = new Color(0x000000);
    }
    else if(value == 32) {
        background = new Color(0xf77842);
        text = new Color(0x000000);
    }
    else if(value == 64) {
        background = new Color(0xff0000);
        text = new Color(0x000000);
    }
    else if(value == 128) {
        background = new Color(0xff00ff);
        text = new Color(0x000000);
    }
    else if(value == 256) {
        background = new Color(0xff00ff);
        text = new Color(0x000000);
    }
    else if(value == 512) {
        background = new Color(0xff00ff);
        text = new Color(0x000000);
    }
    else if(value == 1024) {
        background = new Color(0xff00ff);
        text = new Color(0x000000);
    }
    else {
        background = Color.black;
        text = Color.white;
    }

    g.setColor(new Color(0,0,0,0));
    g.fillRect(0, 0, WIDTH, HEIGHT);

    g.setColor(background);
    g.fillRoundRect(0, 0, WIDTH, HEIGHT, ARC_WIDTH, ARC_HEIGHT);

    g.setColor(text);

    if(value <= 32) {
        font = Game.main.deriveFont(50f);
    }
    else {
        font = Game.main;
    }
```

```
        g.setFont(font);

        int drawX = WIDTH/2 - DrawUtils.getMessageWidth("" + value, font, g)/2;   // keeps the value in center
wrt height
        int drawY = HEIGHT/2 + DrawUtils.getMessageHeight("" + value, font, g)/2;  // keeps the value in center
wrt width
        g.drawString("" + value, drawX, drawY);        // draw the string into the tile
        g.dispose();
    }

    public void render(Graphics2D g) {
            g.drawImage(tileImage, x, y, null);        // draw the tile image
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
        drawImage();
    }

    public boolean CanCombine() {
        return canCombine;
    }

    public void setCanCombine(boolean canCombine) {
        this.canCombine = canCombine;
    }

    public Point getSlideTo() {
        return slideTo;
    }

    public void setSlideTo(Point slideTo) {
        this.slideTo = slideTo;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
```

```
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }
}
```

Tile class is only creating a block with passing value at the time of object declaration as the string in it and the coordinates as the position of it in the tile matrix. This class create the block, fill the colour of the tile, colour of the font, size of the font and draw image of the tile on the screen.

- Keyboard class:

```
import java.awt.event.KeyEvent;

public class Keyboard {

   public static boolean[] pressed = new boolean[256];
   public static boolean[] prev = new boolean[256];

   private Keyboard() {}

   public static void update() {
      for(int i=0;i<4;i++) {
         if(i==0) prev[KeyEvent.VK_LEFT] = pressed[KeyEvent.VK_LEFT];
         if(i==1) prev[KeyEvent.VK_RIGHT] = pressed[KeyEvent.VK_RIGHT];
         if(i==2) prev[KeyEvent.VK_UP] = pressed[KeyEvent.VK_UP];
         if(i==3) prev[KeyEvent.VK_DOWN] = pressed[KeyEvent.VK_DOWN];
      }
   }

   public static void keyPressed(KeyEvent e) {
      pressed[e.getKeyCode()] = true;
   }

   public static void keyReleased(KeyEvent e) {
      pressed[e.getKeyCode()] = false;
   }

   public static boolean typed(int keyEvent) {
      return !pressed[keyEvent] && prev[keyEvent];
   }
}
```

The Keyboard class helps to get input from the keyboard entered and it stores true or false according to which key from the keyboard is pressed in the Boolean array to keep track on which operation is to be performed.

- **Direction Enum:**

```
public enum Direction {
    LEFT,
    RIGHT,
    UP,
    DOWN
}
```

Direction enum is a special class that represents these four LEFT, RIGHT, UP and DOWN constants like final variables.

- **Point class:**

```
public class Point {

    public int row,col;

    public Point(int row,int col) {
        this.row = row;
        this.col = col;
    }
}
```

Point class gives us the coordinates at which a particular tile is located in the tile matrix to keep track on it.

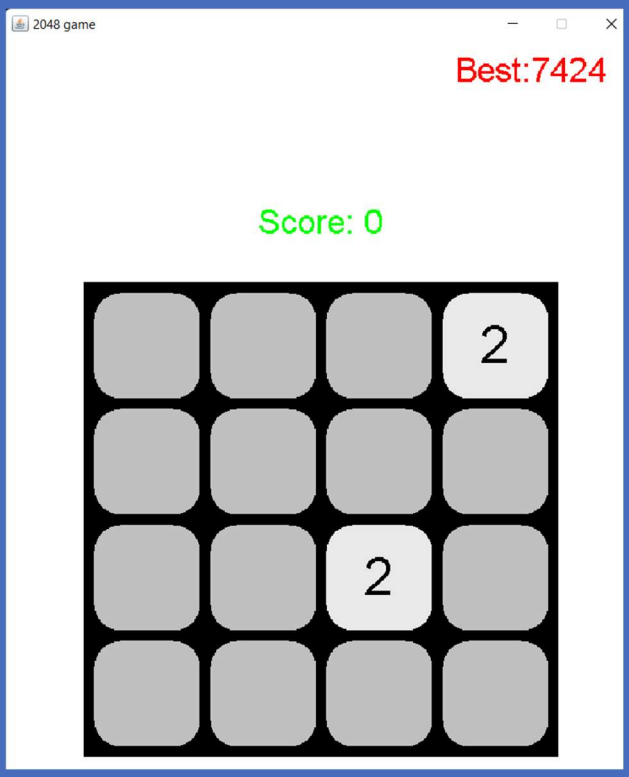
- **Drawutils class:**

```
import java.awt.Font;
import java.awt.Graphics2D;
import java.awt.font.TextLayout;
import java.awt.geom.Rectangle2D;

public class DrawUtils {

   private DrawUtils() { }
     aw
   public static int getMessageWidth(String message, Font font,Graphics2D g) {
      g.setFont(font);
      Rectangle2D bounds = g.getFontMetrics().getStringBounds(message, g);
      return (int)bounds.getWidth();
   }

   public static int getMessageHeight(String message, Font font,Graphics2D g){
      g.setFont(font);
      if(message.length() == 0) return 0;
      TextLayout t1 = new TextLayout(message,font,g.getFontRenderContext());
      return (int)t1.getBounds().getHeight();
   }
}
```

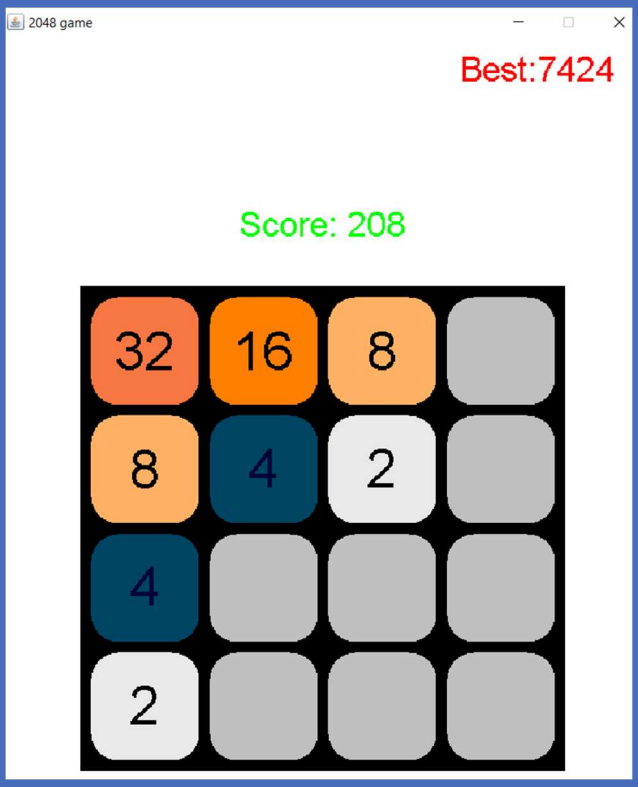
DrawUtils class helps us to determine the width and height of the message or the string inside the tile so as to fit it in the tile dimensions or to avoid overlapping in the tile matrix and string present in it.

Output:

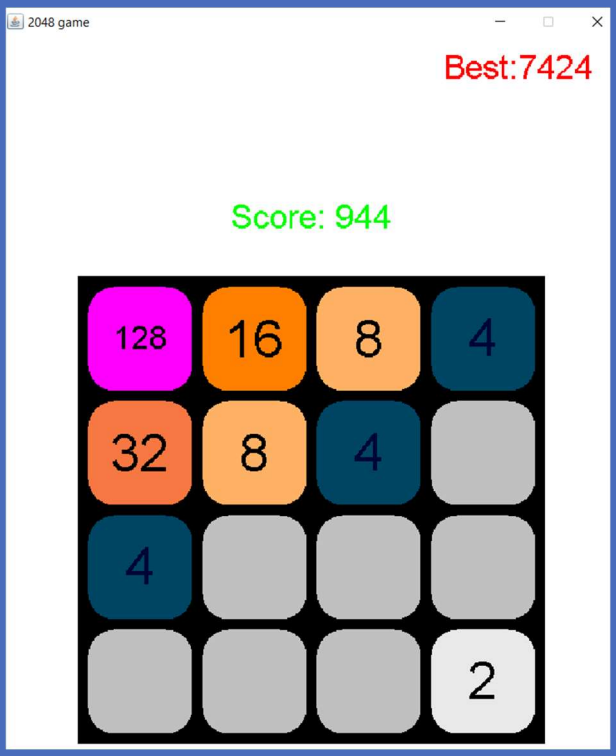
1.


2048 game
Best:7424
Score: 0
2
2

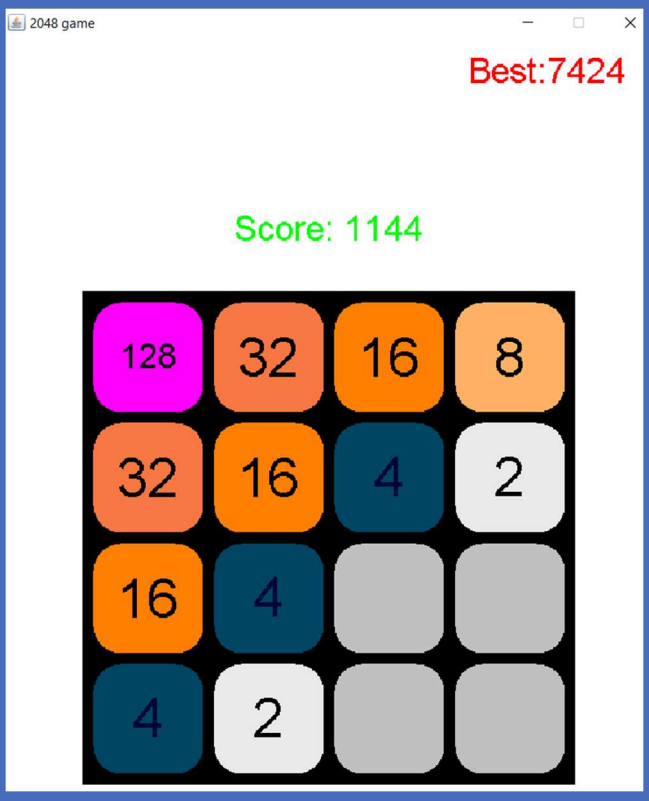
2.


2048 game
Best:7424
Score: 208
32
16
8
8
4
2
4
2

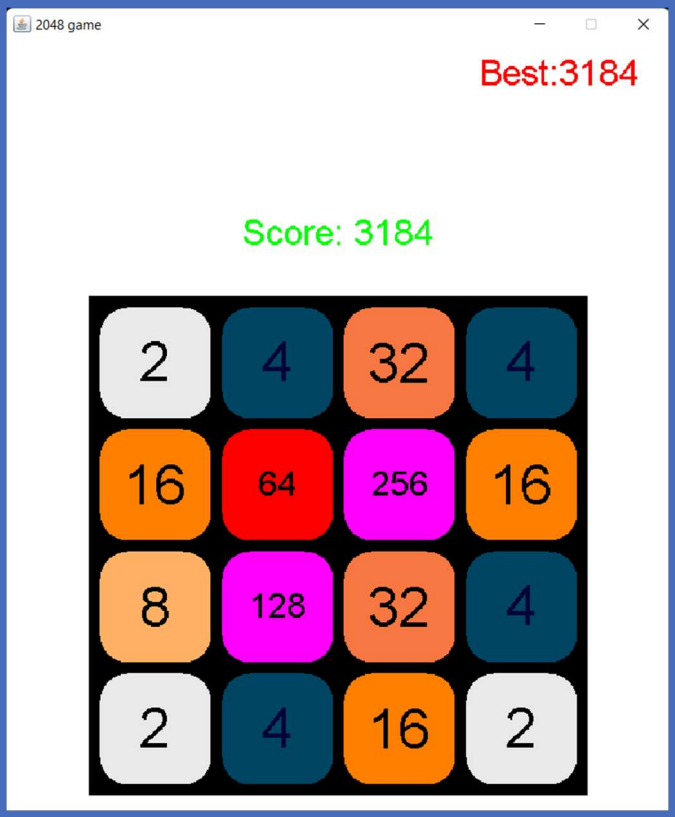
3.


2048 game
Best:7424
Score: 944
128
16
8
4
32
8
4
4
2

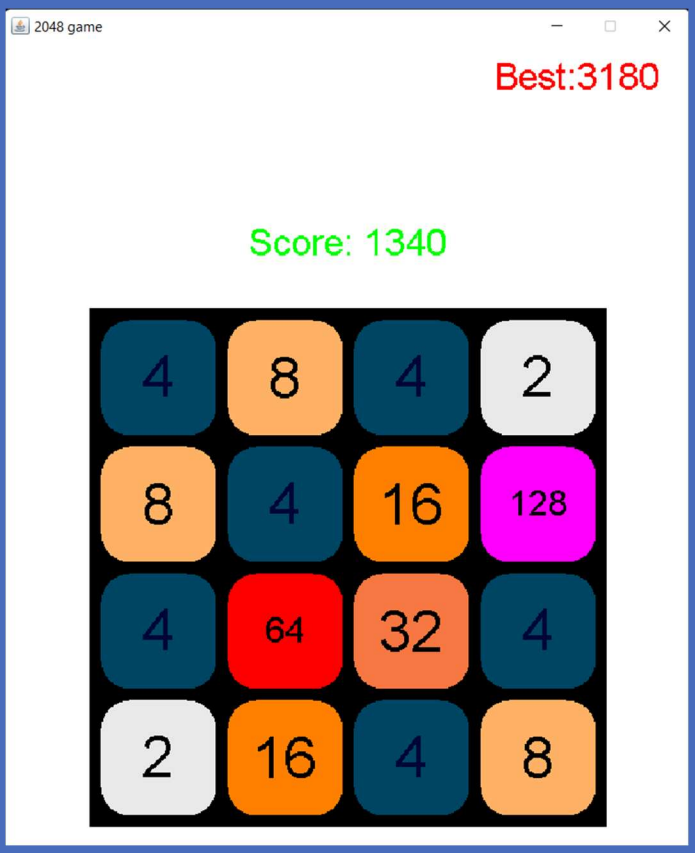
4.


2048 game
Best:7424
Score: 1144
128
32
16
8
32
16
4
2
16
4
4
2

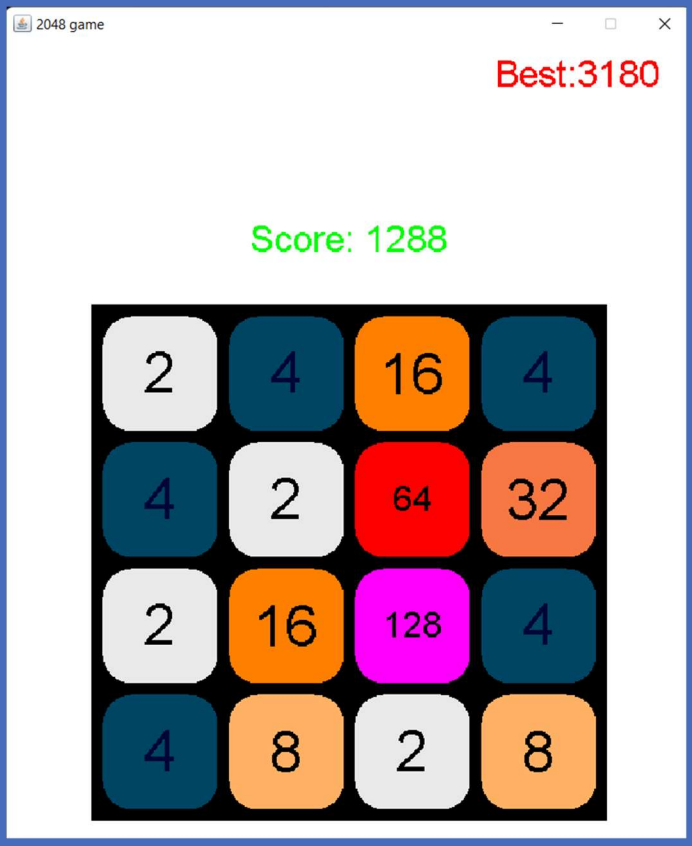
5.


2048 game
Best:3184
Score: 3184
2 4 32 4
16 64 256 16
8 128 32 4
2 4 16 2

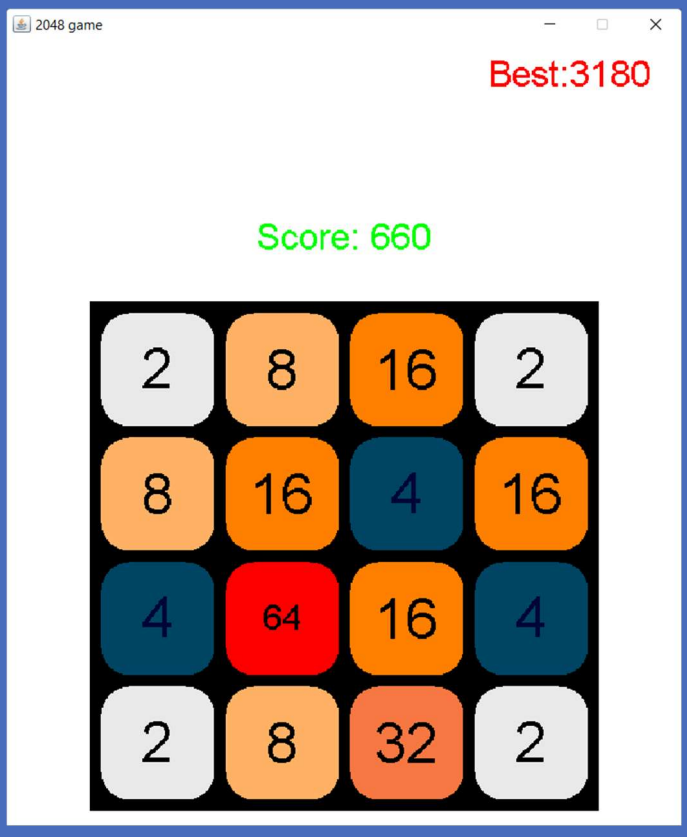
6.


2048 game
Best:3180
Score: 1340
4 8 4 2
8 4 16 128
4 64 32 4
2 16 4 8

7.


2048 game
Best:3180
Score: 1288
2 4 16 4
4 2 64 32
2 16 128 4
4 8 2 8

8.


2048 game
Best:3180
Score: 660
2 8 16 2
8 16 4 16
4 64 16 4
2 8 32 2

9.


2048 game
Best:3180
Score: 2628
2 4 16 4
16 256 8 2
64 8 64 8
4 2 32 2

10.


2048 game
Best:26524
Score: 26524
2048 512 256 4
32 64 16 32
8 16 8 4
4 8 4 2

Contributions:

(All methods mentioned below is present in Gameboard class)

1. Yash:
   Class- Gameboard
   Methods- start(), SpawnRandam()
   Extras- Contributed in making report

2. Nishant:
   Class- start
   Methods- move(), checkOutofBounds()

3. Ankit Kumar Mishra:
   Class- Game
   Methods- update(), resetPosition()
   Extras- Contributed in making report

4. Nakul Yadav:
   Class- Tile
   Methods- checkdead(), checkSurroundingTiles()

5. Nagpure Atharva Sunil
   Class- Keyboards, Direction, DrawUtils, Point
   Methods- moveTiles(), checkKeys()