***Make a Square Operating System~2 Project.***

# ***Project Description:-***

## ***Introduction:***

The Square Puzzle Project aims to construct a 4x4 square using a set of puzzle pieces with different shapes and sizes. The goal is to design an algorithm that efficiently analyzes the puzzle pieces, determines valid combinations, and generates a suitable arrangement. The input includes information about the pieces, dimensions, and shapes. The program will output a 4-row by 4-column square grid with unique numeric identifiers for each piece using threads.

***Objectives/*** ***Goals:***

**Algorithm Development:** Design and develop an efficient algorithm or program to arrange a given set of 4 or 5 puzzle pieces into a 4x4 square.

**Piece Utilization:** Ensure that all provided pieces are utilized in forming the square, considering rotations and flips as valid transformations.

**Multiple Solutions Handling:** Account for scenarios with multiple valid solutions for a given set of puzzle pieces. Implement a mechanism using threads to identify and report these multiple solutions accurately.

**Output Formattin**g: Generate an output that presents the arrangement of pieces within a 4-row by 4-column grid. Utilize numeric identifiers to represent each piece within the square, replacing the solid portions of the puzzle piece representation.

# ***What We Have Done:-***

It was time-consuming to understand what the project should do and understand the functional requirements.

Our plan was designed to divide the project into multiple sections.

As of my last knowledge update in January 2022, there isn't a standardized set of Tetris shapes for a 4x4 grid that is widely recognized or used in the official Tetris guidelines. Tetris shapes typically involve arrangements of four squares in various configurations.

However, it's worth noting that Tetris is a highly adaptable and modifiable game, and different versions and variations exist. Game developers and enthusiasts often create their own versions of Tetris with unique rules, shapes, and grids.

If there have been changes or new developments in the world of Tetris since my last update, I recommend checking the official Tetris guidelines, websites, or forums dedicated to Tetris enthusiasts for the latest information on any modifications or innovations in Tetris gameplay, including 4x4 grid shapes.

# ***Code Documentation:-***

**Board:** The ConnectedZerosCounter Java class contains two methods: checkadj and dfs. The checkadj method counts the number of connected zero elements in a 4x4 matrix using depth-first search (DFS). It iterates through the matrix, invoking the dfs method for each unvisited zero elements to explore connected zero elements in four directions. The dfs method performs DFS traversal, marking visited elements and accumulating the count of connected zero elements. The count returned represents the total number of connected zero elements that do not form a group of four connected zeros. The class has limitations, such as assuming a fixed 4x4 matrix size and lacking input validation for matrix dimensions or handling matrices of different sizes. Modifications may be necessary based on specific requirements or use cases.

public class ConnectedZerosCounter {  
 public static int checkadj(int[][] matrix) {  
 int connectedZerosCount = 0;  
  
 boolean[][] visited = new boolean[4][4];  
  
 for (int i = 0; i < 4; i++) {  
 for (int j = 0; j < 4; j++) {  
 if (matrix[i][j] == 0 && !visited[i][j]) {  
 int count = *dfs*(matrix, visited, i, j, 0);  
 if (count % 4 != 0) {  
 connectedZerosCount += count;  
 }  
 }  
 }  
 }  
  
 return connectedZerosCount;  
 }  
  
 private static int dfs(int[][] matrix, boolean[][] visited, int row, int col, int countSoFar) {  
 if (row < 0 || row >= 4 || col < 0 || col >= 4 || visited[row][col] || matrix[row][col] != 0) {  
 return countSoFar;  
 }  
  
 visited[row][col] = true;  
  
 int count = countSoFar + 1;  
 count = *dfs*(matrix, visited, row - 1, col, count); // Up  
 count = *dfs*(matrix, visited, row + 1, col, count); // Down  
 count = *dfs*(matrix, visited, row, col - 1, count); // Left  
 count = *dfs*(matrix, visited, row, col + 1, count); // Right  
  
 return count;  
 }  
   
}

**HelloApplication:** The movepiece class adjusts the positioning and movement of puzzle pieces within a 4x4 matrix grid to solve a puzzle. It uses different methods and instances from various classes such as ConnectedZerosCounter, pieces, rotate, and solver to arrange pieces in various positions and rotations within the matrix. The moves method attempts to find a valid arrangement by trying various piece placements and rotations, checking for connected zero elements, and adjusting positions accordingly. If successful, it returns the completed matrix; otherwise, it indicates the inability to complete the matrix with the given pieces and algorithm.

public class movepiece {  
 ConnectedZerosCounter checker = new ConnectedZerosCounter();  
 int n = 0;  
 int r = 0;  
 pieces mymat = new pieces();  
 rotate rot = new rotate();  
 //int l[][] = mymat.l;  
 //int matrix[][] = mymat.mat;  
 int m [][] = new int[4][4];  
 solver solves = new solver();  
 public int[][] moves(int [][]l, int matrix[][], int ro){  
 m = solves.solve(l, matrix, n, r);  
 while (m == matrix)  
 {   
 if (r + l[0].length <= 3)  
 {  
 r += 1;  
 m = solves.solve(l, matrix, n, r);  
 if (checker.*checkadj*(m) != 0)  
 m = matrix;   
 }  
 else if (n + l.length <= 3)  
 {   
 r = 0;  
 n += 1;  
 m = solves.solve(l, matrix, n, r);  
 if (checker.*checkadj*(m) != 0)  
 m = matrix;   
 }  
 else  
 {  
 if (ro <= 3)  
 {  
 l = rot.rotateArrayClockwise(l);  
 r = 0;  
 n = 0;  
 ro++;  
 }  
 else  
 {   
 System.*out*.println("the matrix couldn't be complete with this pieces in this algorithm");  
 return (matrix);   
 }  
 }  
 }  
 return (m);  
 }  
 }

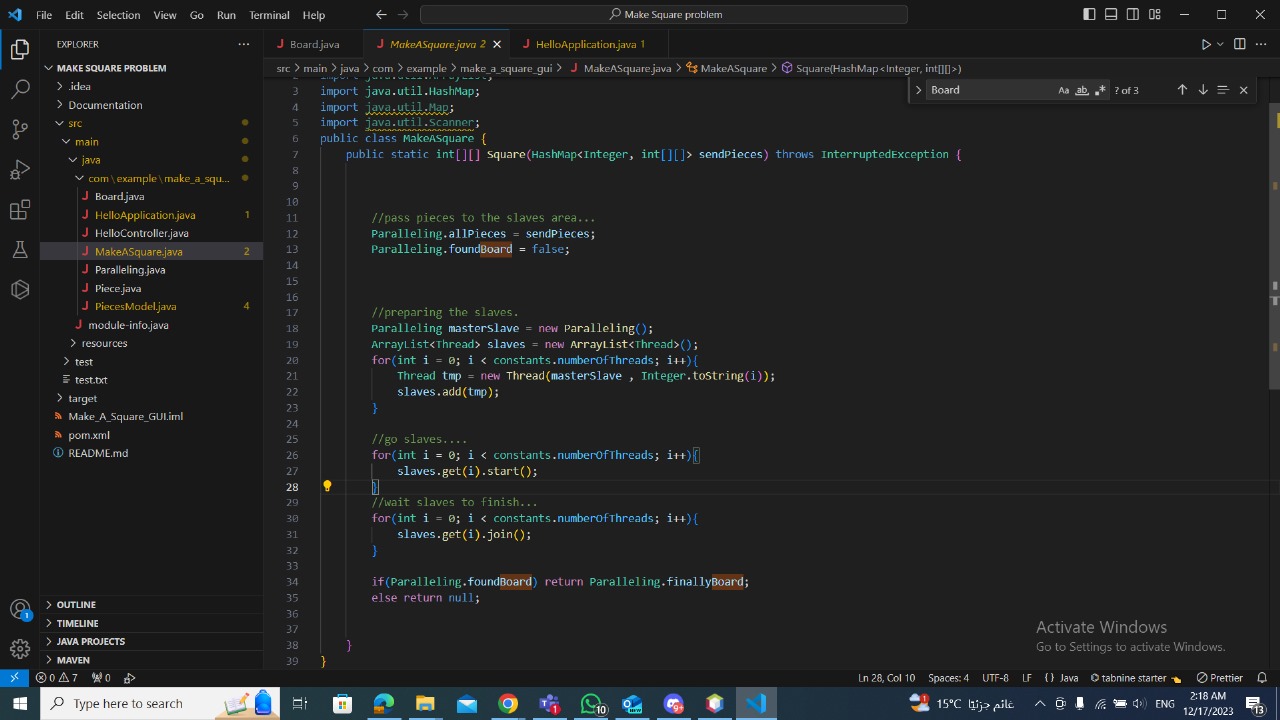
**Hello Counter:** The rotate class provides a method, rotateArrayClockwise, which performs a 90-degree clockwise rotation on a given 2D integer array (representing a matrix). It creates a new array to store the rotated matrix, swapping the rows and columns accordingly. The method iterates through the original matrix, transposing elements to achieve the clockwise rotation, and returns the rotated matrix as output.

public class r {  
   
 public int[][] rotateArrayClockwise(int[][] original) {  
 int rows = original.length;  
 int columns = original[0].length;  
  
 int[][] rotated = new int[columns][rows];  
  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < columns; j++) {  
 rotated[j][rows - 1 - i] = original[i][j];  
 }  
 }  
  
 return rotated;  
 }  
}

**Make a square:**   
The provided code appears to be part of a Java program related to solving a puzzle or problem involving creating a square. Let's break down the main components of the code:

1. **Class Structure:**
   * The code is part of a Java package named **com.example.make\_a\_square\_gui**.
   * It includes several import statements for necessary Java classes, such as **ArrayList**, **HashMap**, **Map**, and **Scanner**.
2. **Class MakeASquare:**
   * This class contains a method named **Square** that takes a **HashMap<Integer, int[][]>** as a parameter.
   * The method throws an **InterruptedException**.
   * Inside the method, there are several operations related to parallel computing.
3. **Parallel Computing:**
   * The code sets up parallel processing using multiple threads to solve a problem concurrently.
   * It uses a class named **Paralleling** to handle the parallel computation.
   * It initializes an instance of **Paralleling** as **masterSlave**.
   * It creates a number of threads (**numberOfThreads**) and stores them in an **ArrayList** named **slaves**.
   * Each thread is associated with an index, which is a string representation of the index.
   * The threads are started (**start()**) and then joined (**join()**) to wait for their completion.
4. **Communication Between Threads:**
   * The **Paralleling** class seems to have a static field or shared data structure named **allPieces**, which is a **HashMap<Integer, int[][]>**. This data is shared among the threads.
   * The **foundBoard** and **finallyBoard** variables in the **Paralleling** class are also used in the parallel computation.
5. **Return Statement:**
   * If a certain condition (**Paralleling.foundBoard**) is met, the method returns a 2D array (**Paralleling.finallyBoard**).
   * Otherwise, it returns **null**.

Without the implementation of the **Paralleling** class and additional context, it's challenging to provide a detailed understanding of what problem or puzzle this code is solving. The code seems to involve parallel processing to find a solution, possibly related to creating a square. If you have the implementation of the **Paralleling** class or more information about the problem being solved, I could provide a more detailed explanation.

****

|  |  |
| --- | --- |
| **Team members** | **Roles** |
| Mahmoud Ramzy | Backend |
| Mohamed Emad | Backend |
| Mahmoud Tamer | Backend |
| Abdelrahman Moussa | GUI, Documentation |
| Kamal Amjad | GUI, Testing |
| Ziad Abdelaziz | GUI |