OS-2 Project Documentation

Project 5:

MakeASquare

1)Project Description:

- The "Puzzle Solver - Make a Square" project is designed to create a square of a specified size using a set of puzzle pieces. The goal is to arrange the pieces in such a way that they form a complete square, with each piece occupying its designated location in the solution. The program takes as input the number of pieces, along with the specifications for each piece, and outputs all possible solutions or reports that no solution is possible.

- The "Puzzle Solver - Make a Square" project is an engaging computational challenge aimed at arranging puzzle pieces to construct a square of user-specified dimensions. The primary objective is to explore various combinations of rotations and placements to create a complete square using a given set of pieces. The program offers flexibility in handling different puzzle shapes, allowing for an interactive and dynamic puzzle-solving experience.

- Input:
  The first line contains number of pieces. Each piece is then specified by listing a single line with two
  integers, the number of rows and columns in the piece, followed by one or more lines which specify the

shape of the piece. The shape specification consists of 0 or 1 characters, with the 1 character indicating
the solid shape of the puzzle (the 0 characters are merely placeholders). For example, piece A above
would be specified as follows:
2 3
111
101

- Output:
  Your program should report all solution, in the format shown by the examples below. A 4-row by 4-column
  square should be created, with each piece occupying its location in the solution. The solid portions of
  piece #1 should be replaced with `1' characters, of piece #2 with ‘2' characters.
  Sample output that represents the figure above could be:
  1112
  1412
  3422
  3442
  For cases which have no possible solution simply report "No solution possible".

2) What we have actually did:

Our team embarked on the "Puzzle Solver - Make a Square" project with a focus on collaboration and innovation. Here's a breakdown of our contributions and achievements:

1. **Project Planning:**

- We initiated the project with a comprehensive planning phase, outlining the objectives, scope, and target functionalities of the puzzle solver.

2. **Algorithm Development:**
    - The algorithmic core of the project, represented by the solve_puzzle function, was developed collaboratively. We leveraged a recursive approach to explore different piece placements and orientations.
3. **Functional Components:**
    - Key functional components, including place_piece, try_place, rotate_piece, and flip_piece functions, were meticulously developed to ensure the algorithm's flexibility and effectiveness.
4. **Input Handling:**
    - We implemented robust input handling mechanisms to accommodate user specifications for the number of pieces and the shape of each puzzle piece.
5. **Output Formatting:**
    - The output, showcasing visual representations of feasible solutions, was carefully formatted to present a 4x4 grid with each puzzle piece identified by a unique numeric label.
6. **User Interface:**
    - Depending on project requirements, an optional user interface component was designed and integrated. This component enhances user interaction and provides a more intuitive experience.

7. **Testing Procedures:**
    - Rigorous testing scenarios were devised to assess the program's performance under various conditions. These included different numbers of pieces, irregular puzzle shapes, and edge cases. We also ensured graceful handling of invalid inputs.

3) Team members role:

Toaa Assem: Algorithm Specialist

Hassan Abdelhamed: Documentation

Omar Sayed: Threads

Mohammed Nasser: Tester

Ranim Aboelabbas: Tester

Youssef Ahmed: Video & GUI

4) Code documentation:

(1) mThreading Class:

Attributes:

- **boardYdim, boardXdim: Dimensions of the puzzle board.**
- **pieceYdim, pieceXdim: Dimensions of puzzle pieces.**
- **piecesThread: HashMap storing puzzle pieces for the thread.**
- **usablePieces: Array containing usable puzzle pieces.**
- **solutionsThread: ArrayList to store solutions found by the thread.**
- **board: 2D array representing the puzzle board.**
- **depth: Current depth in the recursive solving process.**

```
package makeasquare;

import java.util.*;

public class mThreading implements Runnable {
    //dimensions of board

    static int boardYdim = 4;
    static int boardXdim = 4;
    //dimensions of pieces
    static int pieceYdim = 4;
    static int pieceXdim = 4;
    public HashMap<Integer, ArrayList<int[][]>> piecesThread = new HashMap<Integer, ArrayList<int[][]>>();
    public int[] usablePieces;
    public ArrayList<int[][]> solutionsThread = new ArrayList<int[][]>();
    public int[][] board;
    public int depth = 0;
```

Constructor:

Initializes the mThreading class with puzzle pieces and usable pieces for the thread.

```
public mThreading(HashMap<Integer, ArrayList<int[][]>> pieces, int[] usablePieces) {
    this.piecesThread = pieces;
    this.usablePieces = usablePieces;
}
```

solve Method:

The **solve** method orchestrates the exploration of potential solutions for the puzzle. It systematically considers available puzzle pieces, their permutations, and attempts various placements on the board. Through recursion, it navigates the puzzle landscape, seeking valid configurations. When a valid solution is found, it is added to the list of solutions, preventing duplicate entries.

```
public static void solve
(int[][] board, int[] usablePieces, int depth,ArrayList<int[][]> solutions ,HashMap<Integer,ArrayList<int[][]>> pieces) {
        //for every piece in pieces
        for(int index = 0; index < usablePieces.length; index++) {
                ArrayList<int[][]> permutations = pieces.get(usablePieces[index]);

                //for each permutation for the given piece
                for(int perm = 0; perm < permutations.size(); perm++) {

                        //for each point on the board
                        for(int boardY = 0; boardY < boardYdim; boardY++) {
                                for(int boardX = 0; boardX < boardXdim; boardX++) {

                                        //create a copy of the board -- to edit
                                        int[][] newBoard = new int[board.length][board[0].length];
                                        for(int y = 0; y < board.length; y++) {
                                                for(int x = 0; x < board[0].length; x++) {
                                                        newBoard[y][x] = board[y][x];
                                                }
                                        }
```

```
//try to place piece
boolean returnValue = placePiece(boardY, boardX, usablePieces[index],
                                                    currentPerm: permutations.get( index: perm)

//if piece has been placed
if(returnValue) {
        int[] newPieces = new int[usablePieces.length-1];
        int indexCounter = 0;

        //create a new int[] containing all pieces other than the one that was just placed
        for(int i = 0; i < usablePieces.length; i++) {
                if(usablePieces[i] != usablePieces[index]) {
                        newPieces[indexCounter] = usablePieces[i];
                        indexCounter++;
                }
        }
 //if newPieces is empty -- no pieces remain
 if(newPieces.length == 0) {

          //always add the first solution
          if(solutions.isEmpty()) {
                  solutions.add( e: newBoard);
          }

          //check if solution already exists
          else if(doesSolutionExist( sol: newBoard,solutions) == false) {
                  solutions.add( e: newBoard);
     }
 }
 //recursion
 else {
          solve( board: newBoard,  usablePieces: newPieces, depth+1, solutions,pieces);
 }
```

placePiece Method:

The **placePiece** method endeavors to position a puzzle piece on the board at specified coordinates. It evaluates the boundaries, ensuring the piece fits within them. Additionally, it checks for overlapping and occupied spaces on the board. If the piece can be successfully placed, the method updates the board accordingly and returns **true**; otherwise, it returns **false**..

```
public static boolean placePiece
(int boardY, int boardX, int currentPiece,int[][] currentPerm, int[][] currentBoard)
{

        //for each point in the piece
        for(int pieceY = 0; pieceY < pieceYdim; pieceY++) {
                for(int pieceX = 0; pieceX < pieceXdim; pieceX++) {

                        //if the piece has a filled square
                        if(currentPerm[pieceY][pieceX] != 0) {

                                int y = boardY+pieceY;

                                //check y boundary
                                if(y >= boardYdim) {
                                        return false;
                                }
                                int x = boardX+pieceX;

                                        //check x boundary
                                        if(x >= boardXdim) {
                                                return false;
                                        }

                                        //check if board has empty spot
                                        if(currentBoard[y][x] != 0) {
                                                return false;
                                        }

                                        currentBoard[y][x] = currentPiece;
                                }
                        }
                }
        }
        return true;
}
```

doesSolutionExist Method:

The **doesSolutionExist** method assesses whether a given solution already exists in the list of discovered solutions. It iterates through the list, comparing each solution's configuration with the provided one. If a match is found, indicating a duplicate solution, the method returns **true**; otherwise, it returns **false**. This

prevents redundant entries in the list of solutions.

```
public static boolean doesSolutionExist(int[][] sol,ArrayList<int[][]> solutions) {
        boolean match;

        for(int index = 0; index < solutions.size(); index++) {
                match = true;
                for(int y = 0; y < 2; y++) {
                        for(int x = 0; x < 2; x++) {
                                if((solutions.get(index))[y][x] != sol[y][x]) {
                                        match = false;
                                }
                        }
                }
                if(match == true) {
                        return true;
                }
        }
        return false;
}
```

(2) Master Class:

```
package makeasquare;

import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Master implements Runnable {

    static int[] inputPieces;
    static int keyThreadLJT = 0;
    static int keyThreadSZI = 0;
    public Thread t1, t2, t3, t4;
    public static int[] keyOfSolve;
```

Constructor:

Constructor for the **Master** class. Initializes the **inputPieces** array.

```java
public Master(int[] inputPieces) {
    this.inputPieces = inputPieces;
}
```

---

run Method (implements **Runnable**):

Entry point for the thread. It orchestrates the multi-threaded execution of the puzzle-solving process.

Executes the puzzle-solving process, involving multi-threading, puzzle setup, and solution printing.

```java
@Override
public void run() {
    try {
        HashMap<Integer,ArrayList<int[][]>> pieces1 = new HashMap<Integer,ArrayList<int[][]>>();
        HashMap<Integer,ArrayList<int[][]>> pieces2 = new HashMap<Integer,ArrayList<int[][]>>();
        HashMap<Integer,ArrayList<int[][]>> pieces3 = new HashMap<Integer,ArrayList<int[][]>>();
        HashMap<Integer,ArrayList<int[][]>> pieces4 = new HashMap<Integer,ArrayList<int[][]>>();

        int[] usablePieces1 = null;
        usablePieces1 = board1setup( pieces:pieces1,inputPieces);
        int[] usablePieces2 = null;
        usablePieces2 = board1setup( pieces:pieces2,inputPieces);
        int[] usablePieces3 = null;
        usablePieces3 = board1setup( pieces:pieces3,inputPieces);
        int[] usablePieces4 = null;
        usablePieces4 = board1setup( pieces:pieces4,inputPieces);

        keyOfSolve = usablePieces1;
```

```java
        multiThreading m1 = new multiThreading( pieces:pieces1 , usablePieces:usablePieces1);
        multiThreading m2 = new multiThreading( pieces:pieces2 , usablePieces:usablePieces2);
        multiThreading m3 = new multiThreading( pieces:pieces3 , usablePieces:usablePieces3);
        multiThreading m4 = new multiThreading( pieces:pieces4 , usablePieces:usablePieces4);


        t1 = new Thread( target:m1);
        t2 = new Thread( target:m2);
        t3 = new Thread( target:m3);
        t4 = new Thread( target:m4);

        System.out.println( x:"Start");
        t1.start();
        t2.start();
        t3.start();
        t4.start();

        t1.join();
        t2.join();
        t3.join();
        t4.join();
        System.out.println( x:"finish");
```

```
67                  if (!m1.solutionsThread.isEmpty()) {
68                      PrintSolutionsThreads pr1 = new PrintSolutionsThreads( sol:m1.solutionsThread.get( index:0), num0fthread:1);
69                      Thread p1 = new Thread( target:pr1);
70                      p1.start();
71                  }
72                  if (!m2.solutionsThread.isEmpty()) {
73                      PrintSolutionsThreads pr1 = new PrintSolutionsThreads( sol:m2.solutionsThread.get( index:0), num0fthread:2);
74                      Thread p1 = new Thread( target:pr1);
75                      p1.start();
76                  }
77                  if (!m3.solutionsThread.isEmpty()) {
78                      PrintSolutionsThreads pr1 = new PrintSolutionsThreads( sol:m3.solutionsThread.get( index:0), num0fthread:3);
79                      Thread p1 = new Thread( target:pr1);
80                      p1.start();
81                  }
82                  if (!m4.solutionsThread.isEmpty()) {
83                      PrintSolutionsThreads pr1 = new PrintSolutionsThreads( sol:m4.solutionsThread.get( index:0), num0fthread:4);
84                      Thread p1 = new Thread( target:pr1);
85                      p1.start();
86                  }else {
87                      PrintSolutionsThreads pr1 = new PrintSolutionsThreads();
88                      Thread p1 = new Thread( target:pr1);
89                      p1.start();}
```

```
            } catch (InterruptedException ex) {
                Logger.getLogger( name:MasterThread.class.getName()).log( level:Level.SEVERE,  msg:null,  thrown:ex);
            }
        }
```

board1setup Method:

Prepares puzzle pieces for the first board, considering the number of occurrences specified in **inputPieces**. It organizes the pieces into a hashmap, each identified by a unique integer key. The resulting hashmap represents the usable pieces for solving the puzzle.

```
96      public static int[] board1setup(HashMap<Integer,ArrayList<int[][]>> pieces,int[] inputPieces) {
97
98              System.out.println( x: "wowo");
99              //--------------------PIECE Z--------------------//
100
101             int[][] pieceZa = {{1,1,0,0,0},
102                                {0,1,1,0,0},
103                                {0,0,0,0,0},
104                                {0,0,0,0,0}};
105
106             int[][] pieceZb = {{0,1,0,0,0},
107                                {1,1,0,0,0},
108                                {1,0,0,0,0},
109                                {0,0,0,0,0}};
110
111
                ArrayList<int[][]> pieceZ = new ArrayList<int[][]>();
113
114             pieceZ.add( e:pieceZa);
115             pieceZ.add( e:pieceZb);
119
120                 //----------------------PIECE I----------------------//
121
122                 int[][] pieceIa = {{1,0,0,0,0},
123                                    {1,0,0,0,0},
124                                    {1,0,0,0,0},
125                                    {1,0,0,0,0}};
126                 int[][] pieceIb = {{1,1,1,1,0},
127                                    {0,0,0,0,0},
128                                    {0,0,0,0,0},
129                                    {0,0,0,0,0}};
130
                    ArrayList<int[][]> pieceI = new ArrayList<int[][]>();
132
133                 pieceI.add( e:pieceIa);
134                 pieceI.add( e:pieceIb);
135
```

```
136
137                 //----------------------PIECE J----------------------//
138
139                 int[][] pieceJa = {{1,0,0,0,0},
140                                    {1,1,1,0,0},
141                                    {0,0,0,0,0},
142                                    {0,0,0,0,0}};
143
144                 int[][] pieceJb = {{0,1,0,0,0},
145                                    {0,1,0,0,0},
146                                    {1,1,0,0,0},
147                                    {0,0,0,0,0}};
148
149                 int[][] pieceJc = {{1,1,0,0,0},
150                                    {1,0,0,0,0},
151                                    {1,0,0,0,0},
152                                    {0,0,0,0,0}};
153
154                 int[][] pieceJd = {{1,1,1,0,0},
155                                    {0,0,1,0,0},
156                                    {0,0,0,0,0},
157                                    {0,0,0,0,0}};
                    ArrayList<int[][]> pieceJ = new ArrayList<int[][]>();
161
162                 pieceJ.add( e:pieceJa);
163                 pieceJ.add( e:pieceJb);
164                 pieceJ.add( e:pieceJc);
165                 pieceJ.add( e:pieceJd);
166
```

```
168    //                  ----------------------PIECE L----------------------//
169
170                    int[][] pieceLa = {{1,0,0,0,0},
171                                       {1,0,0,0,0},
172                                       {1,1,0,0,0},
173                                       {0,0,0,0,0},
174                                       {0,0,0,0,0}};
175
176                    int[][] pieceLb = {{1,1,1,0,0},
177                                       {1,0,0,0,0},
178                                       {0,0,0,0,0},
179                                       {0,0,0,0,0},
180                                       {0,0,0,0,0}};
181
182                    int[][] pieceLc = {{0,0,1,0,0},
183                                       {1,1,1,0,0},
184                                       {0,0,0,0,0},
185                                       {0,0,0,0,0},
186                                       {0,0,0,0,0}};
187
188                    int[][] pieceLd = {{1,1,0,0,0},
189                                       {0,1,0,0,0},
190                                       {0,1,0,0,0},
191                                       {0,0,0,0,0},
192                                       {0,0,0,0,0}};
193
194
195
                       ArrayList<int[][]> pieceL = new ArrayList<int[][]>();
197
198                    pieceL.add( e:pieceLa);
199                    pieceL.add( e:pieceLb);
200                    pieceL.add( e:pieceLc);
201                    pieceL.add( e:pieceLd);
202
```

```
//                //----------------------PIECE O----------------------//

                  int[][] pieceOa = {{1,1,0,0,0},
                                     {1,1,0,0,0},
                                     {0,0,0,0,0},
                                     {0,0,0,0,0},
                                     {0,0,0,0,0}};


                  ArrayList<int[][]> pieceO = new ArrayList<int[][]>();

                  pieceO.add( e:pieceOa);
//                //----------------------PIECE S----------------------//

                  int[][] pieceSa = {{0,1,1,0,0},
                                     {1,1,0,0,0},
                                     {0,0,0,0,0},
                                     {0,0,0,0,0}};

                  int[][] pieceSb = {{1,0,0,0,0},
                                     {1,1,0,0,0},
                                     {0,1,0,0,0},
                                     {0,0,0,0,0}};

                  ArrayList<int[][]> pieceS = new ArrayList<int[][]>();

                  pieceS.add( e:pieceSa);
                  pieceS.add( e:pieceSb);
```

```
//              //--------------------PIECE T--------------------//

                int[][] pieceTa = {{1,1,1,0,0},
                                   {0,1,0,0,0},
                                   {0,0,0,0,0},
                                   {0,0,0,0,0},
                                   {0,0,0,0,0}};

                int[][] pieceTb = {{0,1,0,0,0},
                                   {1,1,1,0,0},
                                   {0,0,0,0,0},
                                   {0,0,0,0,0},
                                   {0,0,0,0,0}};

                int[][] pieceTc = {{0,1,0,0,0},
                                   {1,1,0,0,0},
                                   {0,1,0,0,0},
                                   {0,0,0,0,0},
                                   {0,0,0,0,0}};

                int[][] pieceTd = {{1,0,0,0,0},
                                   {1,1,0,0,0},
                                   {1,0,0,0,0},
                                   {0,0,0,0,0},
                                   {0,0,0,0,0}};


                ArrayList<int[][]> pieceT = new ArrayList<int[][]>();

                pieceT.add( e:pieceTa);
                pieceT.add( e:pieceTb);
                pieceT.add( e:pieceTc);
                pieceT.add( e:pieceTd);


            // copy array inputPieces //
int[] copy_inputPieces = new int[inputPieces.length];
for(int i =0 ; i < inputPieces.length ;i++){
    copy_inputPieces[i] = inputPieces[i];
}
            //  ------------------ //
```

```
  int findnumThread=0;

for (int i = 0 , key  ; i < copy_inputPieces.length ;i++){
    if (copy_inputPieces[i] != 0 &&findnumThread==0){
     findnumThread++;
     ArrayList<int[][]> piecethread = new ArrayList<int[][]>();
        System.out.println( x: keyThreadSZI);
        System.out.println( x: keyThreadLJT);
     if(copy_inputPieces[i] == 1){
            key=i+1;
            switch(key) {
                case 5:
                    piecethread.add( e:pieceZ.get( index: keyThreadSZI));
                    pieces.put(key, value:piecethread);
                    break;
                case 6:
                    piecethread.add( e:pieceI.get( index: keyThreadSZI));
                    pieces.put(key, value:piecethread);
                    break;
            case 2:
                piecethread.add( e:pieceJ.get( index: keyThreadLJT));
                pieces.put(key, value:piecethread);
                break;
            case 1:
                piecethread.add( e:pieceL.get( index: keyThreadLJT));
                pieces.put(key, value:piecethread);
                break;
            case 7:
                pieces.put(key,  value:pieceO);
                break;
            case 4:
                piecethread.add( e:pieceS.get( index: keyThreadSZI));
                pieces.put(key, value:piecethread);
                break;
            case 3:
                piecethread.add( e:pieceT.get( index: keyThreadLJT));
                pieces.put(key, value:piecethread);
                break;
        }
}
```

```
else if (copy_inputPieces[i] != 0){
    key=i+1;
    switch(key) {
        case 5:
            piecethread.add( e:pieceZ.get( index:keyThreadSZI));
            pieces.put(5*(copy_inputPieces[i]+6), value:setupcopy( piece:piecethread,(copy_inputPieces[i]+6)));
            break;
        case 6:
             piecethread.add( e:pieceI.get( index:keyThreadSZI));
            pieces.put(6*(copy_inputPieces[i]+6), value:setupcopy( piece:piecethread,(copy_inputPieces[i]+6)));
            break;
        case 2:
            piecethread.add( e:pieceJ.get( index:keyThreadLJT));
            pieces.put(2*(copy_inputPieces[i]+6), value:setupcopy( piece:piecethread,(copy_inputPieces[i]+6)));
            break;
        case 1:
            piecethread.add( e:pieceL.get( index:keyThreadLJT));
            pieces.put(1*(copy_inputPieces[i]+6), value:setupcopy( piece:piecethread,(copy_inputPieces[i]+6)));
            break;
        case 7:
            pieces.put(7*(copy_inputPieces[i]+6), value:setupcopy( piece:pieceO,(copy_inputPieces[i]+6)));
            break;
            case 4:
                 piecethread.add( e:pieceS.get( index:keyThreadSZI));
                pieces.put(4*(copy_inputPieces[i]+6), value:setupcopy( piece:piecethread,(copy_inputPieces[i]+6)));
                break;
            case 3:
                piecethread.add( e:pieceT.get( index:keyThreadLJT));
                pieces.put(3*(copy_inputPieces[i]+6), value:setupcopy( piece:piecethread,(copy_inputPieces[i]+6)));
                break;
        }
    copy_inputPieces[i]--;
    i--;
    }
} else{
    if(copy_inputPieces[i] == 1){
        key=i+1;
        switch(key) {
            case 5:
                pieces.put(key, value:pieceZ);
                break;
            case 6:
                pieces.put(key,  value:pieceI);
                break;
```

```
                case 2:
                    pieces.put(key,  value:pieceJ);
                    break;
                case 1:
                    pieces.put(key,  value:pieceL);
                    break;
                case 7:
                    pieces.put(key,  value:pieceO);
                    break;
                case 4:
                    pieces.put(key,  value:pieceS);
                    break;
                case 3:
                    pieces.put(key,  value:pieceT);
                    break;
            }
    }
else if (copy_inputPieces[i] != 0){
    key=i+1;
    switch(key) {
        case 5:
            pieces.put(5*(copy_inputPieces[i]+6), value:setupcopy( piece:pieceZ,(copy_inputPieces[i]+6)));
            break;
        case 6:
            pieces.put(6*(copy_inputPieces[i]+6), value:setupcopy( piece:pieceI,(copy_inputPieces[i]+6)));
            break;
        case 2:
            pieces.put(2*(copy_inputPieces[i]+6), value:setupcopy( piece:pieceJ,(copy_inputPieces[i]+6)));
            break;
        case 1:
            pieces.put(1*(copy_inputPieces[i]+6), value:setupcopy( piece:pieceL,(copy_inputPieces[i]+6)));
            break;
        case 7:
            pieces.put(7*(copy_inputPieces[i]+6), value:setupcopy( piece:pieceO,(copy_inputPieces[i]+6)));
            break;
        case 4:
            pieces.put(4*(copy_inputPieces[i]+6), value:setupcopy( piece:pieceS,(copy_inputPieces[i]+6)));
            break;
```

```
                case 3:
                    pieces.put(3*(copy_inputPieces[i]+6), value: setupcopy( piece: pieceT,(copy_inputPieces[i]+6)));
                    break;
            }
        copy_inputPieces[i]--;
        i--;
        }
    }
}
        // copy array inputPieces //
        int[] copy2_inputPieces = new int[inputPieces.length];
        for(int i =0 ; i < inputPieces.length ;i++){
            copy2_inputPieces[i] = inputPieces[i];
        }
                //  ----------------- //
        Set<Integer> set0fKey = pieces.keySet();
        Object [] array0fKey  = set0fKey.toArray();

        int[] pieceKeyList = new int[pieces.size()];
        for (int k =0 ; k < array0fKey.length ;k++){
            pieceKeyList[k] = (int) array0fKey[k];
        }

        if(keyThreadLJT<3){
            keyThreadLJT++;
        }
        if(keyThreadSZI<1){
            keyThreadSZI++;
        }
        return pieceKeyList;
    }
```

setupcopy Method:

Creates and returns a copy of a list of puzzle pieces (**piece**). Each element in the copied list is a puzzle piece where every value is multiplied by a specified factor **i**. This method is useful for generating variations of the

same puzzle piece for different threads or configurations.

```
public static ArrayList<int[][]> setupcopy (ArrayList<int[][]> piece , int i) {
        ArrayList<int[][]> piecescopy = new ArrayList<int[][]>();

        for(int[][] piececopy : piece) {
            int [][] newpiece = new int [piececopy.length][piececopy[0].length];
            for (int x = 0 ; x < piececopy.length ;x++){
                for (int y = 0 ; y < piececopy[0].length ;y++){
                    newpiece[x][y] = piececopy [x][y] * i;
                }
            }
            piecescopy.add( e:newpiece);
        }
        return piecescopy;
    }
```

print2DArray Method:

Prints a 2D array (**myArray**) in a structured format, displaying its contents row by row. It ensures proper alignment and separation between elements. This method is primarily designed for displaying puzzle pieces or solutions in a visually comprehensible manner.

```
public static void print2DArray(int[][] myArray) {
            int yLength = myArray.length;
            int xLength = myArray[0].length;
            for(int top = 0; top < xLength*3+2; top++) {
                    System.out.print( s:"-");
            }
            System.out.println();
            for(int row = 0; row < yLength; row++) {
                    System.out.print( s:"|");
                    for(int col = 0; col < xLength; col++) {
                            if(myArray[row][col] < 10) {
                                    System.out.print(" "+myArray[row][col]+" ");
                            }
                            else System.out.print(" "+myArray[row][col]);
                    }
                    System.out.println( x:"|");
            }
            for(int top = 0; top < xLength*3+2; top++) {
                    System.out.print( s:"-");
            }
            System.out.println();
```

(3) numOfPieces Class:

Constructors:

- **Description:** Initializes the form for inputting puzzle pieces.
- **Functionality:** Sets up the graphical user interface (GUI) components, including labels, text fields, and a button.

```
package makeasquare;

import javax.swing.JOptionPane;
import javax.swing.JTextField;

/**
 *
 * @author youssef Ahmed
 */
public class numOfPieces extends javax.swing.JFrame {

    /**
     * Creates new form InputPieces
     */
    public numOfPieces() {
        initComponents();
    }
```

jButton1ActionPerformed(ActionEvent evt):

- **Description:** Handles the action when the "SOLVE" button is clicked.
- **Functionality:** Gathers input values from text fields, validates the input, creates a **MasterThread** instance, and starts a new thread to solve the puzzle. Closes the input form after solving.

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

    int[] inputPieces = new int[7];
    JTextField [] jTextFieldsArray = {INput0, INput1, INput2, INput3, INput4, INput5, INput6};
    String numString;
    int numInt = 0;
    for(int i = 0 ; i< 7 ; i++ ){
        numString = jTextFieldsArray[i].getText();
        if(!numString.isEmpty()){
            inputPieces[i]=Integer.parseInt( s:numString);
            numInt += Integer.parseInt( s:numString);
        }
        else inputPieces[i] = 0;
    }
    System.out.println( x:numInt);
    if(numInt < 5){
        MasterThread s1 = new MasterThread(inputPieces);
        Thread t100 = new Thread( target:s1);
        t100.start();
```

```
        try {
            t100.join();

        } catch (InterruptedException ex) {
            JOptionPane.showMessageDialog( parentComponent: null,  message: "Error", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
        }
        this.dispose();
    }
    else JOptionPane.showMessageDialog( parentComponent: null,  message: "Only 4 or 5 picecs", title: "Error", messageType: JOptionPane.ERROR_MESSAGE);
}
```

4) Solutions Class:

```java
package makeasquare;

import java.awt.Color;
import javax.swing.JButton;
import javax.swing.JOptionPane;

public class Solutions extends javax.swing.JFrame implements Runnable {

    int[][] sol;
    static int[] num0fkey;
    public int no = 0;

    public Solutions(int[][] sol, int num0fthread) {
        initComponents();
        no = 1;
        this.sol = sol;
        num0fkey = Master.key0fSolve;
        jLabel2.setText("" + num0fthread);

    }
```

printSolution(int[][] grid, JButton[] JButtonsArray):

- **Description:** Prints the solution on the GUI by setting button colors based on the values in the solution grid.
- **Functionality:** Iterates through the grid, sets button colors based on key values, and hides buttons for empty cells.

```
public void printSolution(int [][] grid, JButton []JButtonsArray) {
  if (grid[0][0] != 0){
     for (int i = 0; i < 4; i++) {
         for (int j = 0; j < 4; j++) {
             if (grid[i][j]==0){
                JButtonsArray[(i*4)+j].setVisible( aFlag: false);
                 System.out.println( x: "NoSolution");
             }
             else if (grid[i][j]==num0fkey[0])
             {
                 JButtonsArray[(i*4)+j].setBackground( bg: Color.red);

             }
             else if (grid[i][j]==num0fkey[1])
             {
                 JButtonsArray[(i*4)+j].setBackground( bg: Color.yellow);
             }
             else if (grid[i][j]==num0fkey[2])
             {
                 JButtonsArray[(i*4)+j].setBackground( bg: Color.BLUE);
               else if (grid[i][j]==num0fkey[3])
               {
                   JButtonsArray[(i*4)+j].setBackground( bg: Color.MAGENTA);
               }
         }
       }
     }
}
```

5) MakeASquare Class:

main Method:

Entry point of the program. Calls the **openForm** method to display the input form for puzzle pieces.

```java
package makeasquare;

import java.awt.Color;
import javax.swing.JFrame;


public class MakeASquare {

    public static void main(String[] args)  {
        openForm(new InputPieces());

    }
```

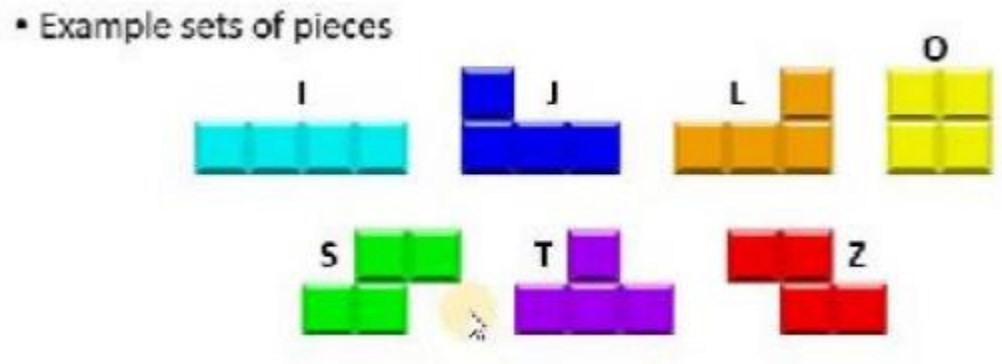
openForm Method:

Configures the provided JFrame for display, setting its properties such as location, default close operation, background color, and visibility. Essentially, it prepares and opens the specified form.

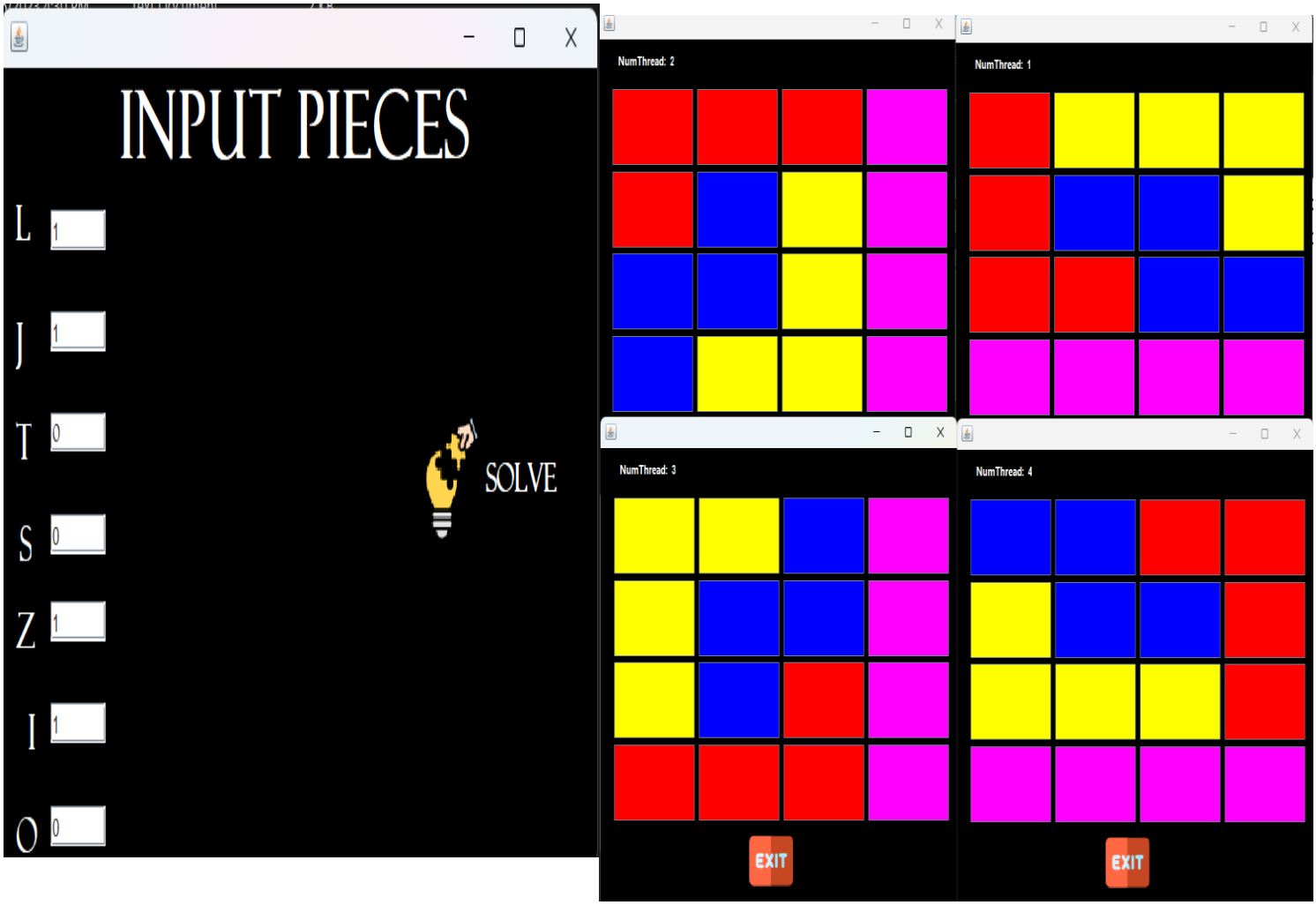
```java
    // open any form
     public static void openForm(JFrame form) {
         form.setLocationRelativeTo( c: null);
         form.setDefaultCloseOperation( operation: 2);
         form.getContentPane().setBackground( c: Color.black);
         form.pack();
         form.setVisible( b: true);
    }

}
```

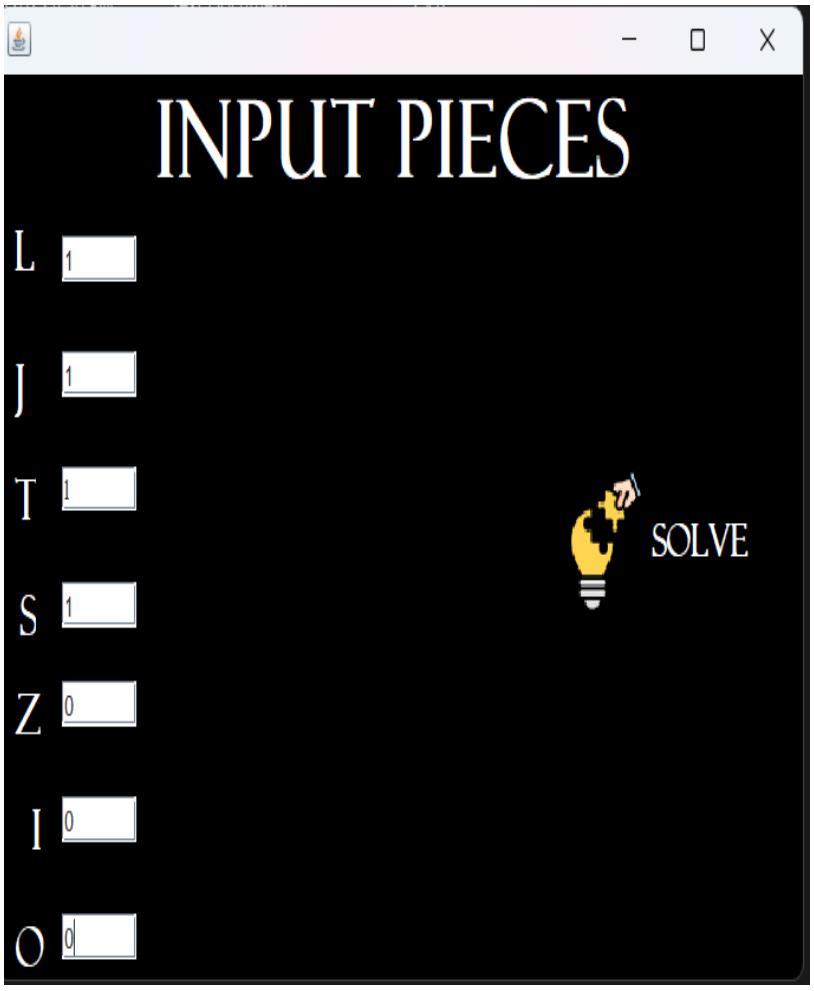
5)GUI

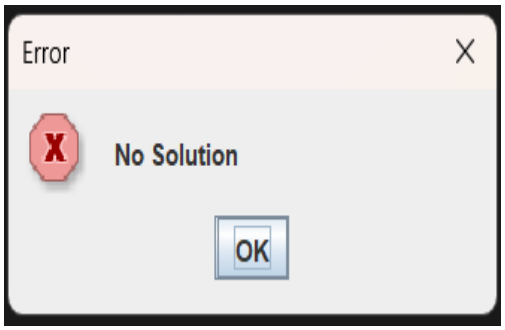


This is an example of pieces we have used in our project:



if the program doesn’t find a way to combine the pieces, it prints no solution found!





If you enter more than 4 or 5 pieces: