Original Goal

“For my project, I want to create a program that will solve various pentomino puzzles by using recursive backtracking to fill a given board… It will first thread start by putting one pentomino in a certain position and anther thread which will create a “new” board. Then it will attempt to place a different pentomino on the “new” board, and so on recursively. The program will attempt to find all (or none, if none exist) solutions for the given board.”

Problem Definition

Make a square with size 4X4 by using 4 or 5 pieces. The pieces can be rotated or flipped, and all pieces should be used to form a square. Example sets of pieces.

There may be more than one possible solution for a set of pieces, and not every arrangement will work even with a set for which a solution can be found. Examples using the above set of pieces... Rotate piece D 90 degree then flip horizontal {R 90 + F H}

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".

You must provide many sample inputs (many text files) during the discussion time, to test your project on many samples. Each text file represents one problem to be solved.

Algorithm:

Developed Algorithm.

The algorithm, done by the recursive function **solve()**, consists of 6 nested for loops.

The first for loop iterates through all pieces. The second for loop iterates through all permutations for the given piece.

The 3rd and 4th for loops iterate through the board's y and x coordinates, respectively.

Next, a secondary function **placePiece()** is called and passed the current board y and x coordinates, and the current piece, permutation, and board. **placePiece()** uses two for loops that iterate through the current piece's y and x coordinates and overlay the board 2D array with the piece's 2D array. If the overlay is successful, the piece is placed, and if not, the function returns.

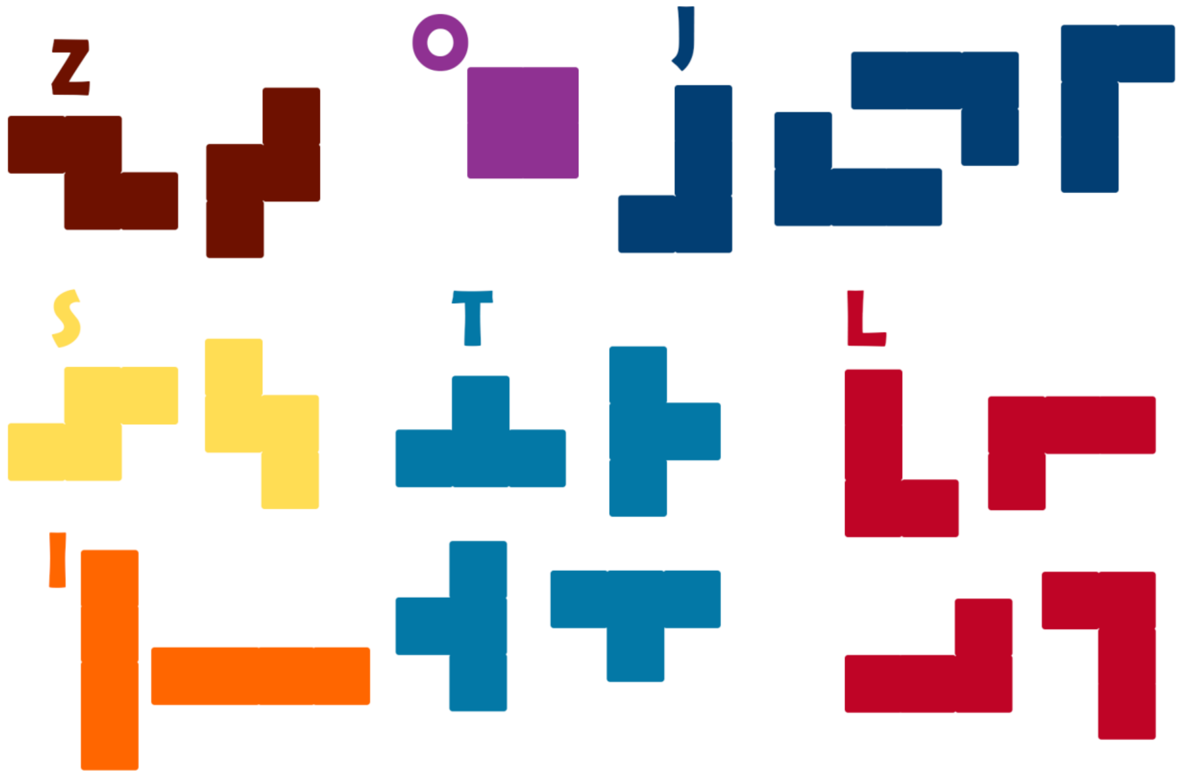
With these iterations, all possibilities are tried, and when all 12 pentominoes can be placed on the board successfully, a solution is recorded.

Pseudocode.

```
solve(board, pieces, solutions):
    for each piece in pieces:
        for each permutation for the piece:
            for each y coordinate of the board:
                for each x coordinate of the board:
                    placepiece(boardy, boardx, piecenum, permutation, board)
                    if piece has been placed:
                        remove placed piece from pieces
                    if pieces is empty:
                        record solution
                    else solve(updated board, updated pieces)


placepiece(boardy, boardx, piecenum, permutation, board):
    for each y coordinate of the piece:
        for each x coordinate of the piece:
            if the current square of the piece is filled:
            y = board y + piece y
            x = board x + piece x
```

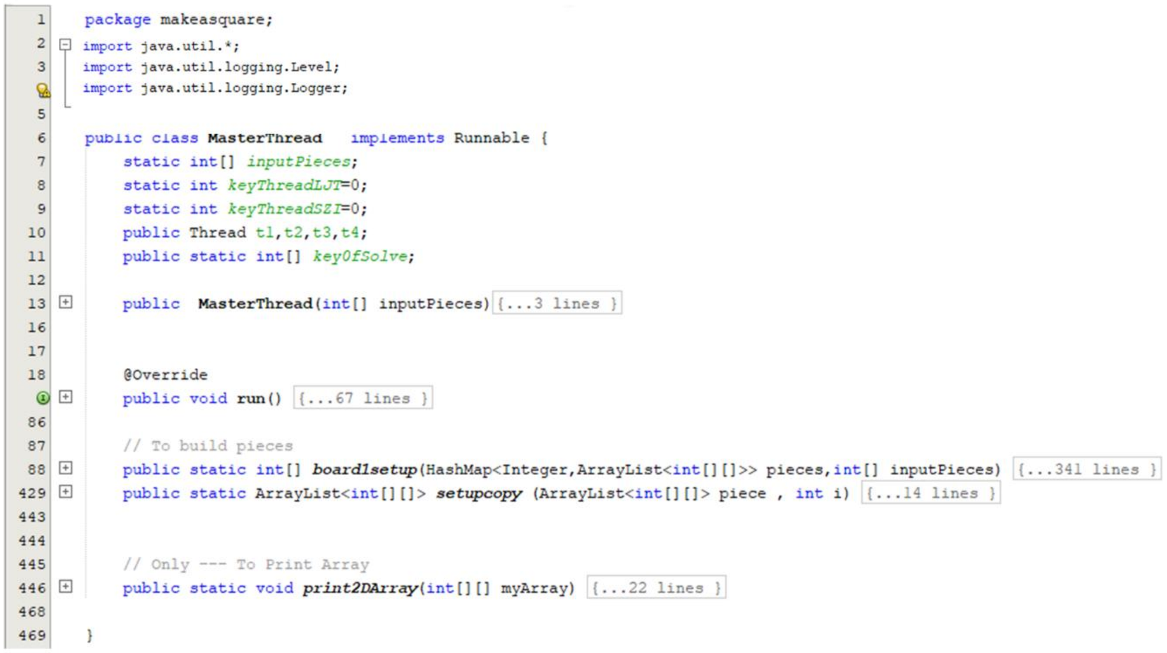
```
                if the piece goes out of bounds:
                        return false
                if board[x][y] is not empty:
                        return false
                board[x][y] = piecenum
        return true
```

Set Pieces.



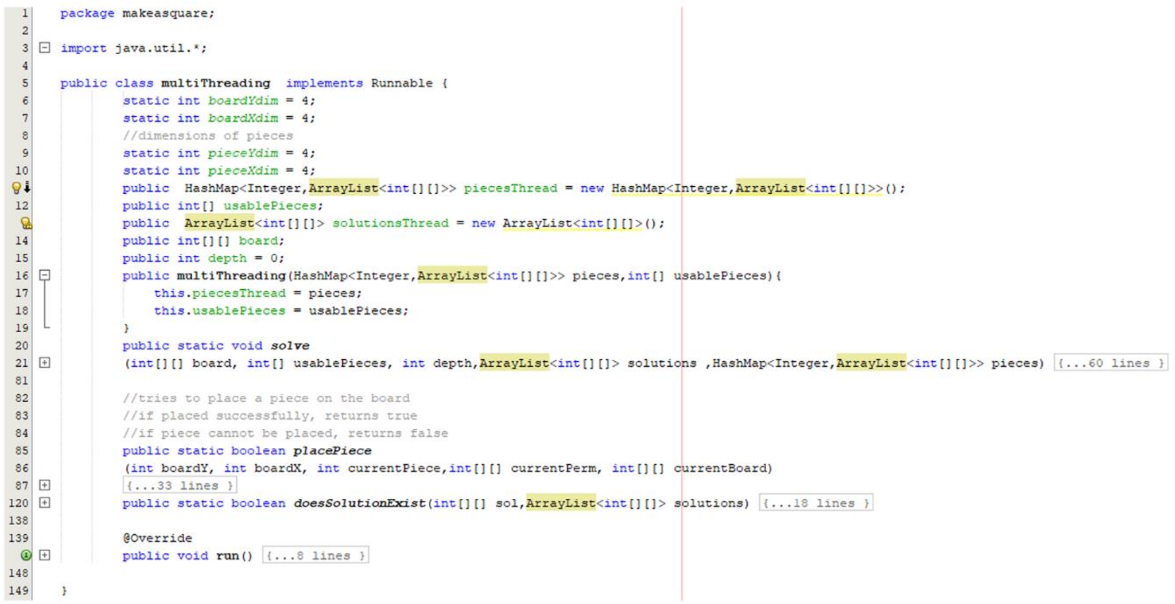
Description of Theads in project:

MasterThread (Input Pieces);



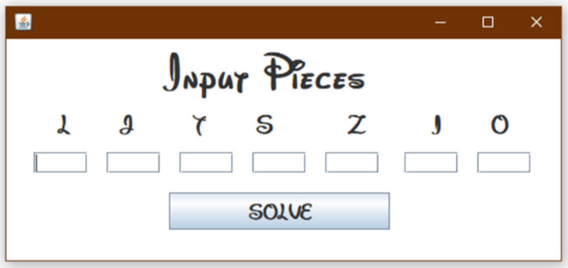
```java
package makeasquare;
import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class MasterThread   implements Runnable {
    static int[] inputPieces;
    static int keyThreadLJT=0;
    static int keyThreadSZI=0;
    public Thread t1,t2,t3,t4;
    public static int[] keyOfSolve;

    public  MasterThread(int[] inputPieces){...3 lines }


    @Override
    public void run() {...67 lines }

    // To build pieces
    public static int[] board1setup(HashMap<Integer,ArrayList<int[][]>> pieces,int[] inputPieces) {...341 lines }
    public static ArrayList<int[][]> setupcopy (ArrayList<int[][]> piece , int i) {...14 lines }


    // Only --- To Print Array
    public static void print2DArray(int[][] myArray) {...22 lines }

}
```

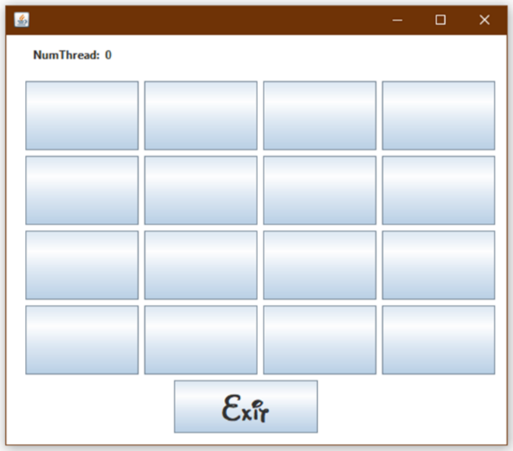
MultiThreading (Pieces , Usable Pieces);



```java
package makeasquare;

import java.util.*;

public class multiThreading  implements Runnable {
        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;
        public multiThreading(HashMap<Integer,ArrayList<int[][]>> pieces,int[] usablePieces){
            this.piecesThread = pieces;
            this.usablePieces = usablePieces;
        }
        public static void solve
        (int[][] board, int[] usablePieces, int depth,ArrayList<int[][]> solutions ,HashMap<Integer,ArrayList<int[][]>> pieces) {...60 lines }

        //tries to place a piece on the board
        //if placed successfully, returns true
        //if piece cannot be placed, returns false
        public static boolean placePiece
        (int boardY, int boardX, int currentPiece,int[][] currentPerm, int[][] currentBoard)
        {...33 lines }
        public static boolean doesSolutionExist(int[][] sol,ArrayList<int[][]> solutions) {...18 lines }

        @Override
        public void run() {...8 lines }

}
```

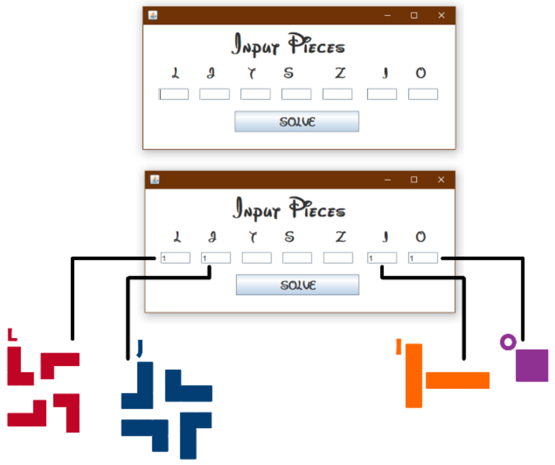
GUI to I/O Pieces:

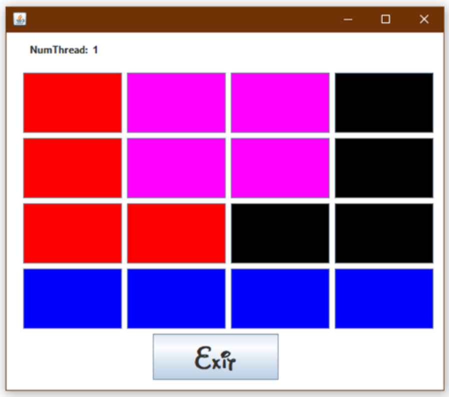




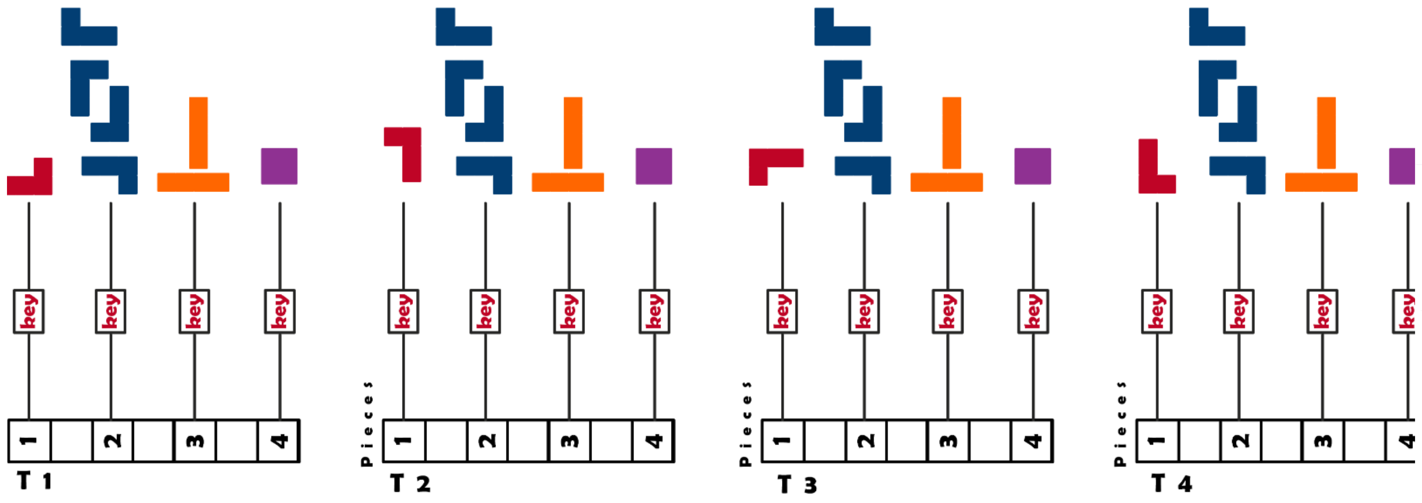
Example:

Input & Output pieces.





MasterThread.



key
1
2
3
4
T 1
Pieces
T 2
T 3
T 4