intro2cs/introcsp/introcst

Ex7: Recursions to the Rescue

Objectives: understand recursive code, write recursive code, practice backtracking.
Task: This exercise contains three tasks that you are required to accomplish:

Part A: Convert recursive code to non-recursive code.

The following **`Mystery`** class contains one public method: **`mysteryComputation.`**
As you can see, this method is implemented by calling a recursive method named **`mysteryRecursion.`** The names and parameters of both methods, as well as the lack of javadoc, do not follow the course readability guidelines, but for a good reason: as a first step, you have to figure out what this code does!

```
public class Mystery {
    public static int mysteryComputation(int n) {
        return mysteryRecursion(n, n-1);
    }

    private static int mysteryRecursion(int a, int b) {
        if(b <= 0) {
            return 0;
        } else {
            int c = mysteryRecursion(a, b - 1);
            if(a % b == 0) {
                c += b;
            }
            return c;
        }
    }
}
```

Your mission, should you choose to accept it... (Ok. You've got us. It's not really a choice. Neither was it in *Mission: Impossible*), is to implement, inside a new class named **`NonRecursiveMystery`**, the method
`public static int mysteryComputation(int n)`
that uses a non-recursive approach to carry out the same computation as **`Mystery.mysteryComputation`** (so it has to return the same result for every n). This method may not call any methods (be them other methods or itself). Do not forget to add thorough javadoc, concisely but precisely explaining what the method does and the meaning of its return value!

This page will self-destruct in 5 seconds. Good luck.

Part B: Get to the Zero

James Bond must pass the corridor leading to his enemy's lair undetected. To help him, Q provides him with a special gadget detecting the timing of when each part of the corridor is checked for intruders. The resulting map looks like this, where if a part of the corridor contains e.g. the number 4, then it is possible to move, undetected, from this part of the corridor only to a part exactly 4 steps away from it.

3	6	4	1	3	4	2	5	3	0

The mission starts with James Bond on the leftmost (first) part of the corridor, and his goal is to reach rightmost (last) part, occupied by a zero, denoting the villain. At each step, James Bond may move a distance indicated by the integer in the part of the corridor he is currently at. Bond may move either left or right along the corridor, but may not move past either end and may not change direction mid-way through a step. For example, the only legal first move is for him to run as fast as he can three squares to the right (to the corridor part marked with a 1), because there is no room to move three squares to the left.
For example, the above puzzle may be solved by making the following series of moves:

3	6	4	1	3	4	2	5	3	0
3	6	4	**1**	3	4	2	5	3	0
3	6	**4**	1	3	4	2	5	3	0
3	6	4	1	3	4	**2**	5	3	0
3	6	4	1	3	4	2	5	**3**	0
3	6	4	1	3	**4**	2	5	3	0
3	6	4	1	3	4	2	5	3	0

Even though the above puzzle is solvable — and indeed has more than one solution — some puzzles of this form may be unsolvable. Consider, for example, the following puzzle:

3	1	2	3	0

Here, one can bounce between the two 3's, but cannot reach any other square. (Don't worry, though. James Bond always will find a way to reach the villain, and will return.)

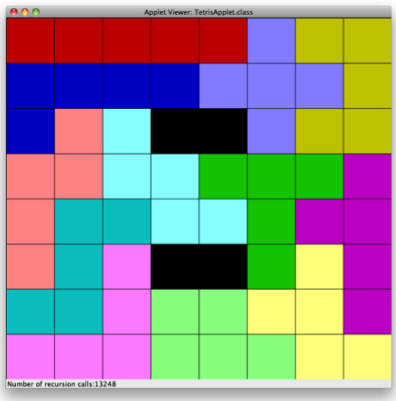
In a new class named **`GetToTheZero`**, write a method:

```
public static boolean isSolvable(int start, int[] board)
```

that takes a starting position of the Agent, and a puzzle board. The function should return true if it is possible to solve the puzzle from the configuration specified in the parameters, and should return false if it is impossible. (Well, for normal humans. Nothing is impossible for Bond.)

You may assume that the integers in the given array are positive, except for the last entry, the goal square, which is always zero. The values of the elements in the vector must be the same after calling your function as they are beforehand (which means that if you choose to change them during processing, you need to change them back!). Your implementation should be recursive, but efficient enough to be able to handle large puzzles.

Part C: Fill-the-Board Puzzle



A Puzzle piece is a shape made of five equal-sized grid squares, where each square shares an edge with one or more other squares. There are exactly twelve such possible shapes (all of which can be seen in the image on the right), not counting all the possible rotations and reflections of the basic shapes. Indiana Jones must place the twelve pieces on an 8-by-8 board, in which four of the squares have already been marked as occupied. Thus, the twelve pieces together with the four pre-occupied squares fill the entire board. Indi must do this fast, or he will be sealed inside a cave forever. In this part of the exercise, your task is to implement a solver that finds a legal arrangement of pieces for the *general version of the puzzle* , given a list of pieces and a board. In this generalized version, each piece can have a different number of squares, the number of pieces may be different, the board can have different height and/or width, and there may be any number of pre-filled squares on the board. Each piece on the list given to the solver may be used at most once in each solution of the puzzle.

The recommended approach for solving this problem is using recursion. The recursive method looks at a board that has already been partially filled with some pieces. The method considers each possible rotation and reflection (collectively called “conformations” from now on) of each remaining puzzle piece in turn. It tries to place that conformation on the board so that it fills the next vacant place (see the next paragraph). If the conformation fits, the method calls itself recursively to try to fill in the rest of the solution. If this attempt fails (or if the conformation does not fit), the method goes on to the next conformation (or to the next piece, if this was the last unchecked conformation of some piece). This approach, of slowly building a solution while eliminating illegal partial solutions (similar to what we used for the 8-queens puzzle), is called “backtracking”, and it is very useful for solving problems with no a priori intuition.

Here are a few more details regarding how we recommend “trying to place a conformation on the board so that it fills the next vacant place” (a phrase we used above):

1. Find the topmost board row that has a vacant place, and find the leftmost vacant place on that row - let’s call it the “next available place” from now on. You should try to legally place the conformation on the board so that one of the squares of this conformation fills this “next available place”.

2. It may seem, at first glance, that if this conformation is of a puzzle piece with k squares, then you should try to place it in k different positions on the board (each time trying to cover the "next available place" with a different square from this conformation). This is actually not the case. The only square, from those making up that conformation, that has a chance of succeeding, is the leftmost square in the top row of that conformation (so you should only try that one, since the cave is about to seal Indiana Jones inside any moment!). This is because of the way we have defined the "next available place" - make sure you understand why this works (if you understand why this works, then you have correctly understood this description).

Preparation: PuzzlePiece

As a first step, you will implement the class **PuzzlePiece**, which represents a puzzle piece, and can return all its possible conformations.

In order to avoid calculating the conformations of a piece over and over again (which takes time), you should calculate them only once in the constructor of **PuzzlePiece**, and store them as class fields. Creating the conformations can be done in the following way:

1. Let's call the conformation given to the constructor "the initial conformation". Starting with the initial conformation, keep rotating it by 90 degrees clockwise until the initial conformation is reached again - the conformations reached by these rotations are all the rotations of one side (we haven't handled flipping yet). There are at most 4 of these.
2. Now flip the initial conformation (vertically, so that the bottom row becomes the top row) and let's call the result "the flipped initial conformation". If the flipped initial conformation is the same as any of the conformations already reached in step 1., then ignore it and finish (the conformations reached in step 1. are all the conformations in this case). Otherwise, repeat the above process again: keep rotating the flipped initial conformation until it is reached again, and these rotations, together with the conformations reached in step 1., are all the conformations of this piece. Thus, there can be between 1 and 8 conformations in total.

Main Task: PuzzleSolver

Your next, main task, is to implement the **PuzzleSolver** class that solves the game given a board and a list of pieces, using recursive backtracking.

When writing your implementation, you will use the following two classes provided to you:

- **PuzzleData** - a container class representing the puzzle board and pieces. Each instance of this class holds two arrays:
    - **PuzzlePiece[] pieces** - an array that holds objects representing all the legal pieces.
    - **int[][] board** - a two-dimensional int array that represents the board. A cell with a value of '-1' represents a pre-filled ('black') square, a cell with a value of '0' represents a blank cell (to be filled), and a positive number k represents a cell occupied by the k-th puzzle piece from the pieces array (e.g. a value of 2 indicates that the cell is occupied by the second piece, i.e. by pieces[1]).

The **PuzzleSolver** class constructor is given a **PuzzleData** instance as a parameter.

- **PuzzleGUI** - a GUI for graphically displaying the progress of the solver. It is provided as a parameter to the **`solve`** method of **PuzzleSolver**. In your solver, as each puzzle piece is placed on the board (or removed from it), the corresponding cells on the GUI's graphics window should be painted accordingly by calls to **PuzzleGUI**'s methods:
  `public void startPiece(int color)` - before starting drawing the piece
  `public void colorSquare(int x, int y)` - one call for each square of the piece
  `public void endPiece()` - to end drawing the piece.
  The color is 0 for white, and an integer between 1 and the number of pieces.
  In order to avoid confusion, note that x is the horizontal coordinate (column number), y is the vertical coordinate (row number), and (0,0) is the top-left corner (exactly the same as with the GridWindow class used for ex5). At any time during the solution, the GUI should display in its status bar the number of recursive calls made by the solver so far (see the school solution). To do this your solver should call the **PuzzleGUI** method: `setStatusMessage(String message)` with the message "Number of recursion calls: X" (where X is the number of recursive calls made by the solver).

Running your Code

To run your solver, we provide you with the full source code for the **PuzzleDriver** class. This class contains a main method that constructs the following objects and runs your solver on them:

1. a **PuzzleData** object containing a n-by-m board, containing only 0 and -1 values, and the array of puzzle pieces that you should use to fill the board.
2. a matching **PuzzleGUI** object.

The driver may be run using: **`java PuzzleDriver`** <*board_num*> <*delay_in_milliseconds*>

- *board_num* indicates which of a set of pre-defined boards to use (each board has the pre-filled black cells in different locations).
  board_num can be 1-3 for four pre-filled cells, or another number for no black cells.
  In such case, change the height and width of the board so the pieces can fill the board (width*height % 5 = 0). You may also try different pre-filled squares and board sizes.
- d*elay_in_milliseconds* indicates the delay between consecutive GUI operations (coloring squares) on the board. Use 1 to do it fast, or a bigger number if you want to follow the moves more slowly.

You may of course make changes to this class for debugging purposes, but when you are finished coding, make sure that your code works flawlessly with our original **PuzzleDriver** implementation, as you will not submit any changes to this class.

School solution

An executable version of the school solution can be invoked from the lab computers by typing: **`~introcsp/bin/ex7/puzzleDriver`** <*board_num*> <*delay*> (as described earlier).

Testing

It is recommended to check your JAR file by copying it to an empty directory, and running the automatic testers by executing the command: **`~introcsp/bin/testers/ex7 ex7.jar`** and verifying that all the tests pass successfully.
You may also download the tests JAR from here (until that file is put in place, there is a partial version). Extract the contents of the JAR file using the shell command:
`jar xvf ex7testing.jar` and follow the instructions in the file called **`TESTING`**.

Further Guidelines

- Please remember to follow the all the course coding style guidelines.
- In particular, remember to use constants, where appropriate, and refrain from duplication of code.
- Document your code – add comments before all major code blocks.
- Write complete Javadoc comments whenever appropriate.
- Don't forget to write a short description of each class in your **`README`** file.

Submission

Submit a JAR file called **`ex7.jar`** containing only the following files:

1. **`NonRecursiveMystery.java`**
2. **`GetToTheZero.java`**
3. **`PuzzlePiece.java`**
4. **`PuzzleSolver.java`**
5. **`README`**

Use the "Upload File" link on the course home page, under the ex7 link.

Good luck!