Assignment in Constructing Software Systems
CSS, Spring 2014

Sliding Blocks Puzzle[1]

A sliding blocks puzzle consists of a number of rectangular blocks that fit into a tray. The goal is to slide the blocks, without lifting any out of the tray, until achieving a certain configuration. An example is shown Figure 1. The minimum number of moves to solve the puzzle is 81.

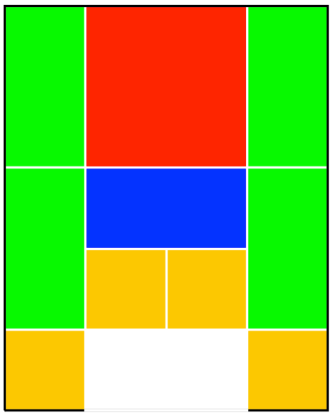

Figure 1. L'Ane Rouge (Red Donkey):
Slide blocks until the red 2x2 block is in the middle of the bottom row.

Examples of these puzzles may be found on the web, for example:

http://www.puzzleworld.org/SlidingBlockPuzzles/sqroot.htm

Your task is to write a program named `Solver.java` that generates a solution to such a puzzle (if a solution exists). Your program need not produce the solution with the fewest number of moves, but it must take as little execution time as possible.

[1] Adapted from Mike Clancy's Sliding Blocks assignment. http://nifty.stanford.edu/2007/clancy-slidingblocks/

Program input

Your program should take two command line arguments:

- The first argument is the name of a file that specifies the initial tray configuration. The first line of this file contains two positive integers, the height (number of rows) and width (number of columns) of the tray. Each subsequent line of this file contains four integers describing a block in the tray: the height and width of the block (both greater than 0), and the row and column of the upper left corner of the block. (The upper left corner of the tray is row 0, column 0.) Blocks are indistinguishable except for their size, and may appear in any order in this file. Thus, the block configuration in Figure 1 might be represented in the file as follows:

```
5 4
2 1 0 0
2 2 0 1
2 1 0 3
2 1 2 0
2 1 2 3
1 2 2 1
1 1 3 1
1 1 3 2
1 1 4 0
1 1 4 3
```

- The second argument is the name of a file that specifies the desired goal configuration. This file is similar in format to the initial configuration file. Each line of this file contains four integers: the length and width of the block (both greater than 0), and the desired position of the upper left corner of the block. This file will not necessarily contain entries for all blocks in the tray. Blocks may appear in any order in this file. The goal configuration for the puzzle in Figure 1 is represented by the single line

```
2 2 3 1
```

If there were more than one 2-by-2 block in the initial configuration, this one-line goal would specify the desired position (3, 1) of *any* of the 2-by-2 blocks. The goal configuration file can specify any number of blocks (up to the total number of blocks in the puzzle.) For example, if the goal was to move any two 1-by-1 blocks to the middle of the bottom row, then the goal configuration file might be:

```
1 1 4 1
1 1 4 2
```

Program output

A solution to the puzzle will represent a sequence of position changes of blocks that, when starting with the specified initial configuration, ends up with blocks in the positions specified in the goal. The only legal moves are those that slide a block horizontally or vertically - not both - into adjacent empty space. Blocks may only be moved an integer number of spaces, and either the row or the column will be the same in the start position as in the end position for each move. Blocks cannot be rotated.

Your program should produce a line of output to `System.out` for each block move that leads to a solution. Each such line contains four integers: the starting row and column of the upper left corner of the moving block, followed by the upper left corner's updated coordinates. Example output appears below; the indicated moves, applied to the starting configuration of Figure 1, could be the first five moves of a solution of the puzzle. (The annotations should not appear in the solution output.)

```
4 3 4 2          // move 1x1 block left
2 3 3 3          // move 2x1 block down
2 1 2 2          // move 1x2 block right
3 1 4 1          // move 1x1 block down
2 0 2 1          // move 2x1 block right
```

Not all problems have solutions. If the goal cannot be reached from the initial configuration, your program should output the string

```
No solution
```

and exit with the call `System.exit(1)`.

Deadlines and materials to be handed in

This assignment is divided into three parts: (1) input/output, (2) puzzle solving, and (3) test of solutions. For each part you should hand in your source code and a small report that presents your design, results, and conclusions. Include instructions on how to run your code. Hand in electronically to keld@ruc.dk. Details and deadlines are given below.

Part 1 (Deadline: **March 13** at 24^{00})

1.1 Write code that reads an initial tray configuration from file and creates data structures that represent the tray of blocks. Example puzzles may be downloaded here:

www.akira.ruc.dk/~keld/teaching/CSS_f14/Assignment/puzzles.zip

1.2 Include an "is OK" method to your class that represents a tray. The method should throw an `IllegalStateException` with an informative message if it finds a configuration problem, e.g. two blocks occupying the same space in the tray, or a block sitting outside the tray.

1.3 Add code that can draw a tray graphically, for example as shown in Figure 1. To simplify this task you can use class `StdDraw`, which is available here:

http://www.cs.princeton.edu/introcs/stdlib/StdDraw.java.html

You are of course also welcome to use Java's AWT and Swing framework.

1.4 (Optional for ECII) Write code for printing a tray textually, for example as shown below.

```
+---+---+---+---+
|   |       |   |
+   +       +   +
|   |       |   |
+---+---+---+---+
|   |       |   |
+   +---+---+   +
|   |   |   |   |
+---+---+---+---+
|   |xxx|xxx|   |
+---+---+---+---+
```

Part 2 (Deadline: **April 3** at 24^{00})

2.1 Write code for solving puzzles. Basically, your program will search the tree of possible move sequences to find a solution to a puzzle. The program must detect configurations that have previously been seen in order to avoid infinite cycling. Hashing is a good technique to apply here.

2.2 Include code that makes it possible for the user to see an animation of a solution.

2.3 (Optional for ECII) Include code that makes it possible for the user to see the solution as a sequence of tray printings (see question 1.4).

2.4 (Optional for ECII) Write code for producing solutions with the fewest possible moves.

Part 3 (Deadline: **May 8** at 24^{00})

3.1 Write a program named `Tester.java` that checks that a given list of moves is a solution of a specified puzzle. The program should have the same command line arguments as `Solver.java`. The list of moves should be read from standard input. Thus your two programs may be run with the UNIX (or DOS) command

```
java Solver init goal | java Tester init goal
```

where `init` and `goal` name the files containing the initial configuration and the goal configuration, respectively.

3.2 Run your program on the example puzzles provided on the web page of this course (see question 1.1). Report your results.

3.3 Hopefully, you are now wiser than at the beginning of the assignment. Write a final report that covers all three parts of the assignment.