Project 3: Sliding Block Puzzles

"A good puzzle, it's a fair thing. Nobody is lying. It's very clear and the problem depends just on you."
–Ernő Rubik, Inventor of Rubik's Cube

Background Information

For this project, you will be writing a program to solve sliding block puzzles. These puzzles consist of a number of rectangular blocks in a tray. The goal is to slide the pieces without lifting any out of the tray until a certain configuration is achieved. An example (from Winning Ways, E.R. Berlekamp et al., Academic Press, 1982) is shown below:

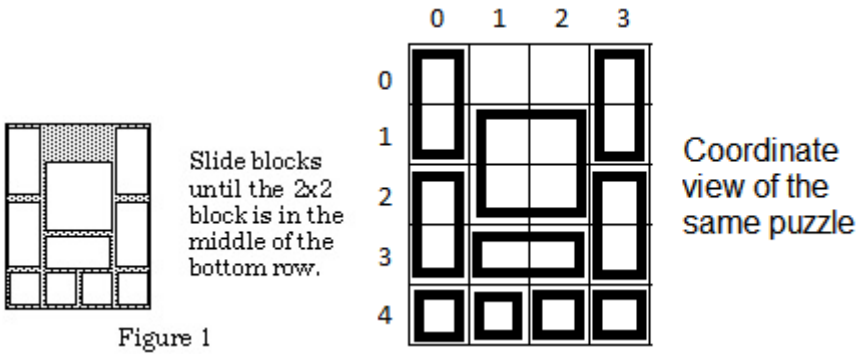


Figure 1

To see how these puzzles work, you can try out these online web apps: Pennant, Red Donkey.

Formatting

Blocks are represented by their top-left coordinate and their bottom-right coordinate. For example, the block at the top-left corner of the tray configuration above has its top-left coordinate at `0 0`, and has its bottom-right coordinate at `1 0`. Thus, that block is represented by the line:

```
0 0 1 0
```

The 2×2 block in the tray configuration above is represented by the line:

```
1 1 2 2
```

Tray configuration files represent trays and all of the blocks contained in a tray. The first line of every tray configuration file will have two positive integers: the height and the width of the tray (separated by a single space character). All subsequent lines of each tray configuration will have four space-separated non-negative integers that represent a block. Blocks can appear in the tray configuration file in any order. You may assume that the length and width of a tray are no larger than 256.

The tray configuration on the previous page can be represented by a file containing the following:

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

Goal configuration files are similar to tray configuration files, except they don't specify the dimensions of the tray. Instead, each line of a goal configuration file specifies the position of a block. In the example above, if our goal was to move the 2×2 block two spaces down in the tray configuration above (and we didn't care where any of the other blocks were), our goal configuration file would contain the single line:

```
3 1 4 2
```

If we also wanted to have other blocks at other positions, we would have more lines in our goal configuration file. Note that if there were multiple 2×2 blocks in the initial configuration, *any* of the 2×2 blocks can be at the specified position to satisfy the goal file example above.

Each **move** is represented by four non-negative space-separated integers. The first two integers make up the current top-left coordinate of the block we want to move. The last two integers make up the top-left coordinate of where we want the block to be after we execute the move. In the example above, if you wanted to move the 2×2 block up one space, you would output the move:

```
1 1 0 1
```

If you wanted to make more moves, your program would output more lines of moves.

1 `Checker`

Before you write your puzzle solver, you should write a `Checker.java` program that verifies whether an input list of moves applied to an initial tray configuration will result in some goal configuration. With past projects, your programs have accepted input via files and command line; for this project, your `Checker` program must accept input via `stdin`.

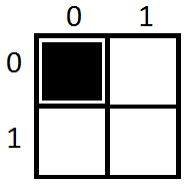
One way to do this is to use Java's `Scanner` class:

```
Scanner s = new Scanner(System.in);
```

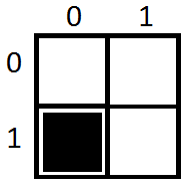
You may find the `Scanner` methods `nextInt()` and `nextLine()` helpful. For more information on the `Scanner` class, refer to the `Scanner` Java docs.

Now you should be able to enter in moves via command line as the program is running. For example, let's say `Checker` was initialized with the following tray configuration:

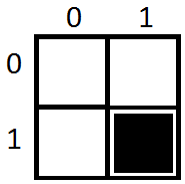
```
2 2
0 0 0 0
```



And the goal configuration `1 1 1 1`. You should be able to type the move `0 0 1 0` into the command line to move the block down:



Finally, the move `1 0 1 1` would move the block to the right, thus satisfying the goal configuration file:



If you would rather input a file of moves instead of entering in moves by hand, you can also pipe the contents of a file of moves to your program on a Unix terminal:

```
cat moves | java Checker init goal
```

where `moves` is a file of moves, `init` is an initial tray configuration file, and `goal` is a goal configuration file. This outputs the contents of the file `moves` to `stdout`, then directly feeds the output to the `Checker` program as `stdin`.

In the 2-move example above, the `moves` file would just contain:

```
0 0 1 0
1 0 1 1
```

At the end of execution, your program should print out an informative success / error message and call `System.exit(n)` where `n` is:

- 0 if `moves` is a valid list of moves that solve the input puzzle
- 1 if the input moves do not solve the puzzle-
- 2 `Checker` does not receive exactly 2 command line inputs
- 3 if any of the command line inputs to `Checker` is not a valid file
- 4 if the input from `stdin` is not properly formatted (i.e. does not contain exactly 4 integers per line).
- 5 if the contents of `init` or `goal` are not properly formatted
- 6 if the input from `stdin` contains an impossible move

If there are multiple errors present corresponding to multiple exit statuses, exit with the lowest of the statuses. To check the exit status of the most recent program you ran, enter the following into a Unix terminal:

```
echo $?
```

2 Solver

Write a `Solver.java` program that prints moves to `stdout` (e.g. `System.out.println`), one move per line. Your program should take in command line arguments as follows:

```
java Solver init goal
```

where `init` is an initial tray configuration file, and `goal` is a goal configuration file. If your `Checker` program works correctly, you can check if your `Solver` output is valid by running the following on a Unix system:

```
java Solver init goal | java Checker init goal
```

As with `Checker`, your `Solver` program should exit with a particular status code under different situations:

- 0 if there are no problems or errors
- 1 if your program finds that the puzzle has no solution
- 2 if `Solver` does not receive exactly 2 command line inputs
- 3 if any of the command line inputs to `Solver` is not a valid file
- 4 if the contents of `init` or `goal` are not properly formatted

Your `Solver` program will be graded on time efficiency (see the "Grading" section below). The amount of space your program needs, however, is not an important consideration, except that your program has to fit in the default allocation of memory provided on the instructional computers.

3 Readme

You must also include with your submission a `readme.pdf` (**must** be a PDF file, not a text file). This is worth a significant portion of your project grade. Your readme file should include the information listed below, answering all the questions in each category:

- **Division of Labor** (half a page): An explanation of how your team split the work for this assignment among your team members, and why you split it this way.
- **Design** (2-3 pages): A description of the overall organization of your submitted program (algorithms and data structures) that lists operations on blocks, trays, and the collection of trays seen earlier in the solution search. Diagrams will be useful here to show the correspondence between an abstract tray and your tray implementation. This description should contain enough detail for another CS 61BL student to understand clearly how the corresponding code would work.
- **Experimental Results** (1-2 pages per experiment): Three experiments comparing results of a design choice from the project. Each experiment should include the following sections and content, written in a way that a fellow CS 61BL student would understand:
  - Summary: description of the test and the results
  - Results: graphs and / or tables with the results of the test
  - Conclusions: explanation and interpretation of the results

  Here are some questions to get you thinking about appropriate tests:
  - What data structure choices did you consider for the tray? What operations did you optimize? (Generation of possible moves? Comparison of the current configuration with the goal? Making a move?) How did these considerations conflict?
  - If you used a data structure that uses hashing, how did you choose a hash function for trays? How did your choice optimize the need for fast computations, minimal collisions, and economical use of memory?
  - How did you choose between moving blocks one square at a time and making longer block moves?
  - How did you choose between breadth-first and depth-first processing of the tree of move sequences? If you took a different approach, what was it and why did you take the approach?
- **Program Development** (1 page): An explanation of the process by which you constructed a working program:
  - What did you code and test first, and what did you postpone?
  - Why did you build the program in this sequence?
  - How did your team test your program? What test cases did you use for each of your classes, and how did you choose them?
- **Disclaimers** : In this section, describe parts of your solution that don't work, if any. You will lose fewer points for bugs in your project that are listed here.
- **Improvements** (half a page): If you were to make one more improvement to speed up your program, what would it be, and what is your evidence for expecting a significant speedup?

Students and software engineers are expected to write design documentation in industry and in courses such as CS162 and CS169, so you might as well get practice with them now.

Contest

Starting Thursday August 14 (after the final exam), an optional project 3 contest will be available. If you want to enter the contest, include a `contest.txt` file with your project submission. This file should have your team name as the first line of the file ($\leq$ 15 alphanumeric characters).

If you submit to the contest, we will time how long it takes for your program to solve a set of unreleased tray and goal files that the course staff has created. We will also check how fast your `Checker` program is. Extra credit points will be assigned as follows:

- Each member of the 1st place team will receive 3 course points
- Each member of the 2nd and 3rd place teams will receive 2 course points
- Each member of the 4th, 5th, and 6th place teams will receive 1 course point

More details regarding the contest will be released on a later date.

Submission Information

Submit project 3 on an instructional machine with the command `submit proj3` by Friday August 15, 10pm PDT. As with project 2, you must submit with a team of 3-4. Your submission must include the following files:

- `Checker.java`
- `Solver.java`
- `readme.pdf`

in addition to all other Java files you wrote for this project. Only one member should submit per team.

Within 24 hours of submitting your project, each team member must submit a group evaluation form. Failure to do so will result in a 1 point penalty. Project grades will be adjusted based on these responses.

Grading

This project is worth 30 course points. The grade breakdown is as follows:

- Code correctness and efficiency: 20
  - 12 points for correctly solving all of the easy puzzles in under 80 seconds each. Easy puzzles can be found at cs61bl/code/proj3/easy (see easy.csv for a complete listing of all easy puzzles). This is all or nothing.
  - 0.3 points per hard puzzle solved in under 80 seconds (capped at 8 points total). Hard puzzles can be found at cs61bl/code/proj3/hard (see hard.csv for a complete listing of all hard puzzles).
- `Readme.pdf`: 10

Checkpoint Checkoff

There will be a project checkpoint checkoff in lab on Monday August 11. To receive full points, your team must demonstrate to your TA that your `Checker` program is fully functional. Your team must also have a draft of your `readme.pdf` file, specifically the "Division of Labor" and the "Design" portions.

Helper Scripts

The staff has provided some scripts you can use to help test your program. The scripts (`run.easy`, `run.medium`, and `run.hard`) will run your `Solver` and `Checker` programs against all of the easy / medium / hard puzzles. They can be found at cs61bl/code/proj3/. To use the scripts, copy them to the same directory as your compiled `Checker.class` and `Solver.class` on an instructional machine. You may have to change the permissions of the script:

```
chmod +x run.easy
```

Run the scripts via command line. For example, if you want to run all of the easy puzzles, you would enter:

```
./run.easy
```

Each script will print out "Verified" for each puzzle your program successfully passes (either solves or correctly detects that there is no solution), and some sort of error message for every puzzle your program fails.

If you can't figure out how to get these to work, you are encouraged to use your favorite search engine to learn why. **You are not required to use these scripts.**

To run them on your own computer, copy all of the puzzle files to your computer and modify the hardcoded path variables in `run.easy`, `run.medium`, `run.hard`, and `runsuccess.sh` to point to where you have your puzzles.