CS 445: Data Structures
Summer 2017

Assignment 3

Assigned: Tuesday, June 20 **Due:** Saturday, July 01 11:59 PM

---

1 Motivation

In this assignment, you will use backtracking to find *normal semi-magic squares*.

A *magic square* is a $n \times n$ square grid of positive integers with special properties. In particular, each cell must contain a unique integer (no duplicates), and the sum of the integers in any row, column, or diagonal is equal to a particular value (called the *magic sum*). For instance, consider the following 3×3 magic square, with magic sum of 72:

23	28	21
22	24	26
27	20	25

A *semi-magic square* is similar to a magic square, except only the rows and columns are considered (the sum of the diagonals is not constrained). All magic squares are also semi-magic squares, but not all semi-magic squares are magic squares.

Finally, a *normal* magic square contains each of the first n^2 positive integers. A normal magic square (or semi-magic square) always has a magic sum of $n(n^2 + 1)/2$. For instance, consider the following 3×3 normal magic square, with magic sum of $3(3^2 + 1)/2 = 15$:

8	1	6
3	5	7
4	9	2

Note that, as a normal semi-magic square, this square contains each value from 1 to $3^2 = 9$.

In this assignment, we will focus on $n \times n$ **normal semi-magic squares**, containing values 1 to n^2, where each row and column must add up to $n(n^2 + 1)/2$.

2 Provided files

First, carefully read the provided files. You can find these files on Pitt Box in a folder named `cs445-a3-abc123`, where `abc123` is your Pitt username.

2.1 Provided code

The `Queens` class includes a backtracking solution to the 8 Queens problem. If you run this class with the `-t` command line argument (called a *flag*), it will run tests of its `reject`, `isFullSolution`, `extend`, `next` methods.

The `MagicSquare` class includes the basic skeleton of a backtracking solution. This class also includes the method `readSquare`, which reads a partially-completed square from a file and returns it as a two-dimensional `int` array, with `0` designating an empty cell (yet to be filled). Finally, it includes a method `printSquare`, which prints out a square to the terminal.

It is recommended that you use this provided code as a starting point. However, you may choose not to use this provided code, as long as your program reads the same input format and uses the required backtracking techniques.

2.2 Example Squares

Several partially-completed squares are available for you to test your code in the `squares/` directory. Below is a summary of these files.

Filename	**Size**	**Solution?**
`two.sq`	2×2	No
`three1.sq`	3×3	Yes
`three2.sq`	3×3	No
`five1.sq`	5×5	Yes
`five2.sq`	5×5	No

Each of these files is plaintext (i.e., can be opened with a text editor such as Notepad or TextEdit). Columns are separated by tabs, and rows are separated by newline. You can also use this format to create partially-completed squares of your own for testing.

3 Tasks

You must write a backtracking program to find values for all of the cells in the square that are not filled. Your solution must satisfy the rules of normal semi-magic squares: each number between 1 and n^2 must appear exactly once, and each row and column must sum up to $n(n^2+1)/2$. You *must* accomplish this using the backtracking techniques discussed and demonstrated in lecture and in `Queens.java`. That is, you need to build up a solution recursively, one cell at a time, until you determine that the current square is impossible to complete (in which case you will backtrack and try another assignment), or that the current square is complete and valid. Alternately, if you conclude that there is no valid solution for the problem instance given, you should print "No solution".

Your class must be named `MagicSquare`, and therefore must be in a file with the name `MagicSquare.java`. It must also be in the package `cs445.a3`.

3.1 Modes

Your program must support three modes. In **blank** mode, you will start from a blank square, and construct a normal semi-magic square of the requested size. In **fill** mode, you will start from a partially-completed square read from a file. Finally, in **test** mode, you will print the results of several tests of the backtracking mechanics.

Blank mode To execute in blank mode, simply give the square size as a command-line argument. For instance, the following command should find a 3×3 normal semi-magic square, starting from a blank square.

```
java cs445.a3.MagicSquare 3
```

Fill mode To execute in fill mode, give the square size **and** square filename as command-line arguments. The following command will find a 4×4 normal semi-magic square, starting from the provided partially-completed square described in `square.sq` (in the format described in section 2.2).

```
java cs445.a3.MagicSquare 4 square.sq
```

Your program will read in the partially-filled square from the file. This square will have several cells filled, and other cells (value of 0) will be considered empty. Your goal is to find values for the remaining cells **without changing any of the cells provided** (i.e., you can change only those with an initial value of 0).

Note that specifying a filename with all 0 values should behave identically to blank mode. Also note that filenames are relative to your terminal location, so square files should stay *outside* of the package directory (and if they are inside the `squares` directory, you should specify them as, e.g., `squares/two.sq`).

Test mode When run with the argument `-t`, your program should execute in test mode. In this mode, you should run each of the test methods described below in section 3.3. An example command is given below.

```
java cs445.a3.MagicSquare -t
```

3.2 Required Methods

As stated above, you **must** use the techniques we discussed in lecture for recursive backtracking. As such, you will then need to write the following methods to support your backtracking algorithm.

- `isFullSolution`, a method that accepts a partial solution and returns `true` if it is a complete, valid solution.
- `reject`, a method that accepts a partial solution and returns `true` if it should be rejected because it can **never** be extended into a complete solution.
- `extend`, a method that accepts a partial solution and returns another partial solution that includes one additional choice added on. This method will return `null` if no more choices can be added to the solution.

Note: Be sure that your `extend` method creates a **new** partial solution, rather than modifying its argument in-place (recall that the runtime stack can only contain references, not objects themselves!).

- `next`, a method that accepts a partial solution and returns another partial solution in which the *most recent* choice that was added has been changed to its next option. This method will return `null` if there are no more options for the *most recent* choice that was made (even if there are other options for other choices!).

Hints:

1. You should implement the above methods as efficiently as possible. 6 bonus points will be given for solutions efficient enough to solve a very hard puzzle within a fixed time limit. At a minimum, you should be able to solve the examples provided within 1 minute.
2. One key challenge in this assignment is, in fill mode, differentiating between cells that were filled by your program (and thus can be changed) and those that were specified in the provided file (and thus cannot be changed). If you were solving this puzzle by hand on paper, how would you differentiate between these two types of cells? I can think of two main approaches, each of which has a programming analogue.
3. As with the examples discussed in class, none of the 4 methods described above needs to be recursive itself. The recursion happens within `solve`, enabled by these helper methods.

3.3 Test Methods

When developing a complex program such as this one, it is important to test your progress as you go. For this reason, in addition to the backtracking-supporting methods above, you will be required to test your methods **as you develop them**. Re-read starting at Chapter 2.16 to review the concepts behind writing test methods. To test each of the backtracking methods, you need to write the following test methods. For each one, you should create a variety of partial solutions that cover as many corner cases as you can think of. Then, call the method you are testing on each partial solution, and ensure that the result is as expected. Include enough test cases that the correct output **convinces** you that your method works properly in all situations.

- `testIsFullSolution`, a method that generates partial solutions and ensures that the `isFullSolution` method correctly determines whether each is a complete solution.
- `testReject`, a method that generates partial solutions and ensures that the `reject` method correctly determines whether each should be rejected.
- `testExtend`, a method that generates partial solutions and ensures that the `extend` method correctly extends each with the correct next choice.
- `testNext`, a method that generates partial solutions and ensures that the, in each, `next` method correctly changes the most recent choice that was added to its next option.

You may either hardcode your test partial solutions, or submit `.sq` files containing the squares you want to test. In either case, when your program is run with the `-t` flag, your program must run the above four test methods and output the results.

4 Grading

Your grade for this assignment will be based on:

- Your program's success at finding magic squares in blank mode (20%) and fill mode (40%). This includes correctly stating that there is no solution, where applicable.
- The thoroughness of your test methods (40%).

5 Submission

Upload your java files in the provided Box directory as edits or additions to the provided code.

All programs will be tested on the command line, so if you use an IDE to develop your program, you must export the java files from the IDE and ensure that they compile and run on the command line. Do not submit the IDE's project files. Your TA should be able to download your `cs445-a3-abc123` directory from Box, and compile and run your code. Specifically, `javac cs445/a3/MagicSquare.java` must compile your program, when executed from the root of your `cs445-a3-abc123` directory. Similarly, the commands given in section 3.1 should run your program, when executed from the root of your directory.

In addition to your code, you may wish to include a `README.txt` file that describes features of your program that are not working as expected, to assist the TA in grading the portions that do work as expected.

Your project is due at 11:59 PM on Saturday, July 01. You should upload your progress frequently, even far in advance of this deadline: **No late submissions will be accepted.**