Programming Assignment #1: SneakyQueens

COP 3503, Fall 2021

Due: Sunday, September 5, *before* 11:59 PM

Abstract

This is a warm-up assignment to get you thinking mathematically, algorithmically, and *cleverly*. It will encourage you to think in terms of implementing efficient solutions to problems, since your program needs to have a worst-case runtime that does not exceed O($m + n$) (linear runtime). It's also a fairly gentle return to Java for those who haven't used it in a while, and involves a direct application of the base conversion knowledge you gained in Computer Science 1 (albeit with a minor twist).

You might find this problem very tricky at first. It's important to struggle with it. Don't be discouraged if you don't solve it right away. Maybe walk away, take a break, and come back to it later (perhaps even the following day). You might be amazed by what your brain can do if you let it work on a problem in the background and/or if you come back to a problem well-rested, with a fresh perspective.

Please feel free to seek out help in office hours if you're lost, and remember that it's okay to have conceptual discussions with other students about this problem, as long as you're not sharing code (or pseudocode, which is practically the same thing). Just keep in mind that you'll benefit more from this problem if you struggle with it a bit before discussing it with anyone else.

Deliverables

SneakyQueens.java

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Problem Statement

You will be given a list of coordinate strings for queens on an arbitrarily large square chess board, and you need to determine whether any of the queens can attack one another in the given configuration.

In the game of chess, queens can move any number of spaces in any of eight directions: up, down, left, right, or any of four possible diagonal directions (up-left, up-right, down-left, or down-right). For example, the queen on the following board (denoted with a letter ‘Q’) can move to any position marked with an asterisk (‘*’), and no other positions:

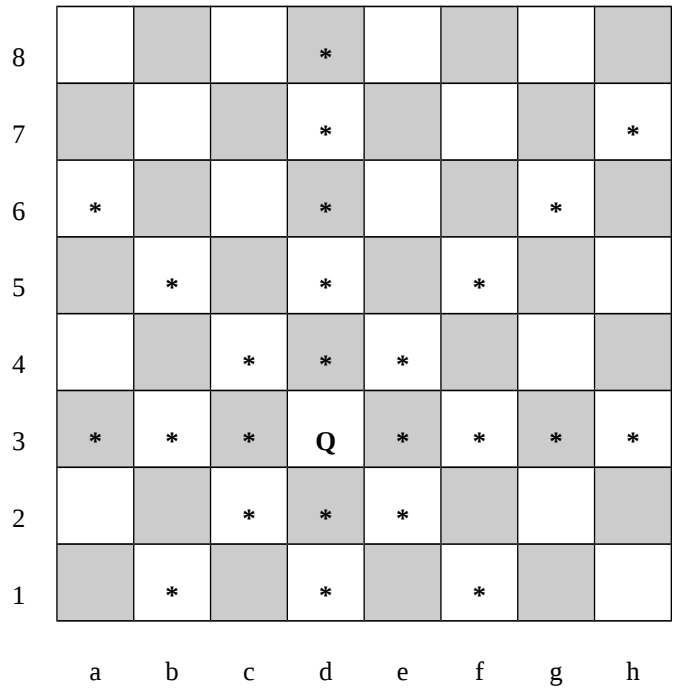

Figure 1: *The queen at position d3 can move to any square marked with an asterisk.*

Thus, on the following board, none of the queens (denoted with the letter ‘Q’) can attack one another:

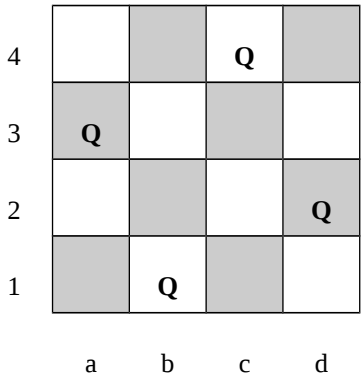

Figure 2: *A 4x4 board in which none of the queens can attack one another.*

In contrast, on the following board, the queens at *c6* and *h6* can attack one another, as can the queens at *f4* and *h6*:

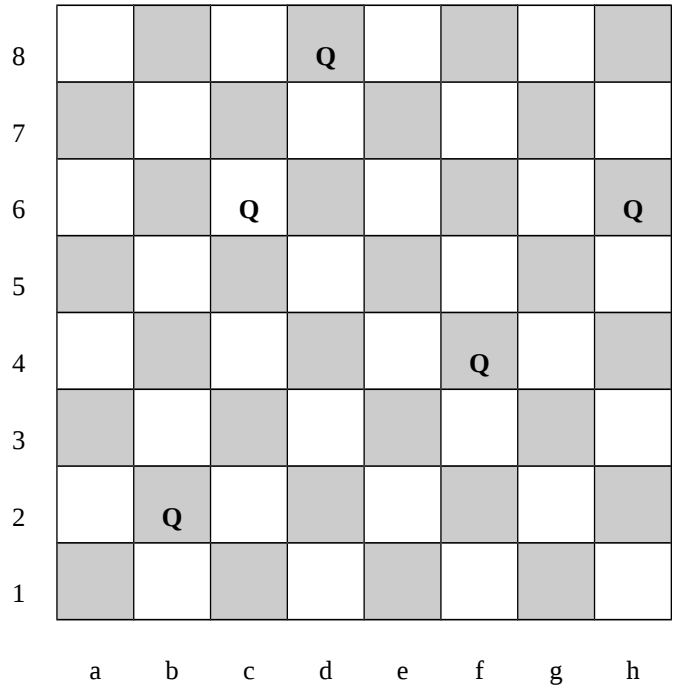

Figure 3: *An 8x8 board in which some of the queens can attack one another.*

2. Coordinate System

One standard notation for the location of a chess piece on an 8x8 board is to give its column, followed by its row, as a single string with no spaces. In this coordinate system, columns are labeled *a* through *h* (from left to right), and rows to be numbered 1 through 8 (from bottom to top).

So, for example, the board in Figure 2 (above, on pg. 2) has queens at positions *a3*, *b1*, *c4*, and *d2*.

Because you're going to be dealing with much larger chess boards in this program, you'll need some sort of notation that allows you to deal with boards that have more than the 26 columns we can denote with the letters *a* through *z*. Here's how that will work:

Columns will be labeled *a* through *z* (from left to right). After column *z*, the next 26 columns will be labeled *aa* through *az*. After column *az*, the next 26 columns will be labeled *ba* through *bz*, and so on. After column *zz*, the next 26 columns will be labeled *aaa* through *aaz*.

Essentially, the columns are given in a base 26 numbering scheme, where digits 1 through 26 are represented using *a* through *z*. However, this counting system is a bit jacked up since there's no character to represent the value zero. (That's part of the fun.)

All the letters in these strings will be lowercase, and all the strings are guaranteed to be valid representations of board positions. They will not contain spaces or any other unexpected characters.

For example:

1. In the coordinate string *a1*, the *a* tells us the piece is in the first column (from the left), and the 1 tells us the piece is in the first row (from the bottom).
2. Similarly, the string *z32* denotes a piece in the 26th column (from the left) and 32nd row (from the bottom).
3. The string *aa19* represents a piece in the 27th column (from the left) and 19th row (from the bottom).
4. The string *fancy58339* would represent a piece in the 2,768,999th column (from the left) and the 58,339th row (from the bottom). (However, as you'll see below, 2,768,999 exceeds the maximum width of the chess boards you'll be required to handle.)

Converting these strings to their corresponding numeric coordinates is one of a few key algorithmic / mathemagical challenges you face in this assignment. You might want to write a separate helper method that does that for you.

3. Runtime Requirements

In order to pass all test cases, the worst-case runtime of your solution cannot exceed $O(m + n)$, where m is both the length and width of the square chess board, and n is the number of coordinate strings to be processed. This figure assumes that the length of each coordinate string is bounded by some constant, which means you needn't account for that length in your runtime analysis, provided that the entire length of each string is processed or examined only some small, constant number of times (e.g., once or twice).

Equivalently, you may conceive of all the string lengths as being less than or equal to k, in which case the worst-case runtime that your solution cannot exceed would be expressed as $O(m + nk)$.

Note! $O(m + n)$ is just another way of writing $O(\text{MAX}\{m, n\})$, meaning that your runtime can be linear with respect to m or n – whichever one happens to be the dominant term for any individual test case.

4. Method and Class Requirements

Implement the following methods in a class named *SneakyQueens*. Please note that they are all ***public*** and ***static***. You may implement helper methods as you see fit.

```
public static boolean
allTheQueensAreSafe(ArrayList<String> coordinateStrings, int boardSize)
```

> **Description:** Given an ArrayList of coordinate strings representing the locations of the queens on a *boardSize* × *boardSize* chess board, return `true` if none of the queens can attack one another. Otherwise, return `false`.
>
> **Parameter Restrictions:** *boardSize* will be a positive integer, with *boardSize* ≤ 60,000, describing both the length and width of the square board. (So, if *boardSize* = 8, then we have an 8 × 8 board.)

coordinateStrings will be non-null, and any strings within that ArrayList will follow the format described above for valid coordinates on a *boardSize* × *boardSize* board.

Output: This method should **not** print anything to the screen. Printing stray characters to the screen (including newline characters) is a leading cause of test case failure.

```
public static double difficultyRating()
```

Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
public static double hoursSpent()
```

Return an estimate (greater than zero) of the number of hours you spent on this assignment.

5. Compiling and Testing SneakyQueens on Eustis (and the *test-all.sh* Script!)

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. To compile your program with one of my test cases:

   ```
   javac SneakyQueens.java TestCase01.java
   ```

2. To run this test case and redirect your output to a text file:

   ```
   java TestCase01 > myoutput01.txt
   ```

3. To compare your program's output against the sample output file I've provided for this test case:

   ```
   diff myoutput01.txt sample_output/TestCase01-output.txt
   ```

   If the contents of *myoutput01.txt* and *TestCase01-output.txt* are exactly the same, *diff* won't print anything to the screen. It will just look like this:

   ```
   seansz@eustis:~$ diff myoutput01.txt sample_output/TestCase01-output.txt
   seansz@eustis:~$ _
   ```

   Otherwise, if the files differ, *diff* will spit out some information about the lines that aren't the same.

4. I've also included a script, *test-all.sh*, that will compile and run all six test cases for you. You can run it on Eustis by placing it in a directory with *SneakyQueens.java* and all the test case files and typing:

   ```
   bash test-all.sh
   ```

   Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

6. Style Restrictions (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.
- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your import statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.
- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do *not* have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)
- ★ Do not use *var* to declare variables.

7. Grading Criteria and Miscellaneous Requirements

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've release with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

80%	Passes test cases (in linear runtime) with 100% correct output formatting. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.
20%	Adequate comments, whitespace, and style. To earn these points, you must adhere to the style restrictions set forth above. We will likely impose huge penalties for small deviations, because we really want you to develop good style habits in this class. Please include a header comment with your name and NID.

Your program must be submitted via Webcourses.

Please be sure to submit your *.java* file, not a *.class* file (and certainly not a *.doc* or *.pdf* file). Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Important! Programs that do not compile on Eustis will receive zero credit. When testing your code, you should ensure that you place *SneakyQueens.java* alone in a directory with the test case files (source files, sample output files, and the input text files associated with the test cases), and no other files. That will help ensure that your *SneakyQueens.java* is not relying on external support classes that you've written in separate *.java* files but won't be including with your program submission.

Important! You might want to remove *main()* and then double check that your program compiles without it before submitting. Including a *main()* method can cause compilation issues if it includes references to home-brewed classes that you are not submitting with the assignment. Please remove.

Important! Your program should not print anything to the screen. Extraneous output is disruptive to the TAs' grading process and will result in severe point deductions. Please do not print to the screen.

Important! No file I/O. In the required methods you write, please do not read or write to any files.

Important! Please do not create a java package. Articulating a *package* in your source code could prevent it from compiling with our test cases, resulting in severe point deductions.

Important! Name your source file, class(es), and method(s) correctly. Minor errors in spelling and/or capitalization could be hugely disruptive to the grading process and may result in severe point deductions. Similarly, failing to make any of the three required methods, or failing to make them *public* and *static*, may cause test case failure. Please double check your work!

Input specifications are a contract. We promise that we will work within the confines of the problem statement when creating the test cases that we'll use for grading. For example, the strings we pass to *allTheQueensAreSafe* are guaranteed to be properly formed coordinate strings. None of them will contain spaces, capital letters, punctuation, or other characters that would violate the coordinate notation system described in this writeup. Similarly, we will never pass a *boardSize* value less than 1 or greater than 60,000 to your *allTheQueensAreSafe* method.

However, please be aware that the test cases included with this assignment writeup are by no means comprehensive. Please be sure to create your own test cases and thoroughly test your code. Sharing test cases with other students is allowed, but you should challenge yourself to think of edge cases before reading other students' test cases.

Start early! Work hard! Ask questions! Good luck!