Programming Assignment: SneakyRooks

Computer Science at UCF

This was an end of the year assignment designed to reinforce the kinds of algorithmic and clever thinking that the professor had been building towards over the entirety of the Computer Science I course at the University of Central Florida. In particular, this served as an exercise in coming up with efficient solutions to problems, since the solution for the assignment had to have a worst-case runtime that does not exceed O(m + n) (linear runtime).

1. Problem Statement

I was given a list of coordinate strings for rooks on an arbitrarily large square chess board, and I needed to determine whether any of the rooks can attack one another in the given configuration.

For context, In the game of chess, rooks can move any number of spaces horizontally or vertically (up, down, left, or right). For example, the rook on the following board (denoted with a letter 'R') can move to any position marked with an asterisk ('*'), and no other positions:

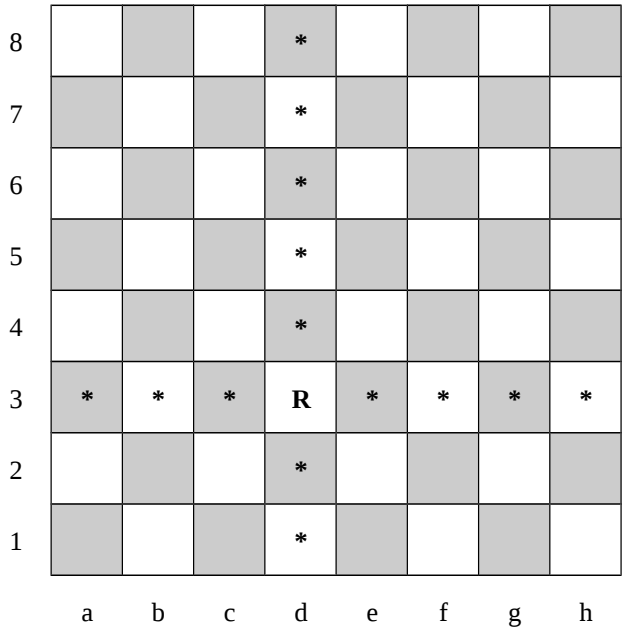

Figure 1: *The rook at position d3 can move to any square marked with an asterisk.*

Thus, on the following board, none of the rooks (denoted with the letter 'R') can attack one another:

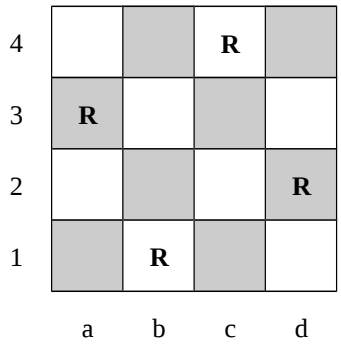

Figure 2: *A 4x4 board in which none of the rooks can attack one another.*

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

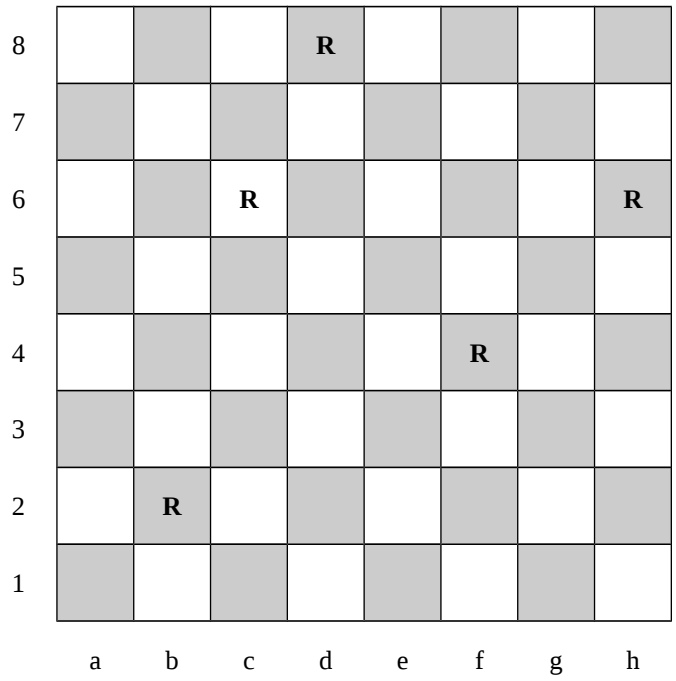

Figure 3: *An 8x8 board in which two of the rooks can attack one another.*

2. Chess Board Coordinates

2.1. 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 rooks at positions *a3*, *b1*, *c4*, and *d2*.

Because I was going to be dealing with much larger chess boards in this program, I needed some sort of notation that allowed me 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).

Converting these strings to their corresponding numeric coordinates was one of a few key algorithmic / mathemagical challenges I faced in this assignment.

2.2. Coordinate Struct (SneakyRooks.h)

To store rook coordinates, I had to use the struct definition that the professor had specified in `SneakyRooks.h` without any modifications.

```
#include "SneakyRooks.h"
```

The struct I will use to hold rook coordinates is defined in `SneakyRooks.h` as follows:

```
typedef struct Coordinate
{
   int col;  // The column where this rook is located (1 through board width).
   int row;  // The row where this rook is located (1 through board height).
} Coordinate;
```

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 each string is processed or examined only some small, constant number of times (e.g., once or twice).

Equivalently, the professor specified that I may conceive of all the string lengths as being less than or equal to k, in which case the worst-case runtime that my 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 the runtime can be linear with respect to m or n – whichever one happens to be the dominant term for any individual test case.

7. Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (`.c` files) at the command line:

```
gcc SneakyRooks.c testcase01.c
```

By default, this will produce an executable file called `a.out`, which you can run by typing:

```
./a.out
```

If you want to name the executable file something else, use:

```
gcc SneakyRooks.c testcase01.c -o SneakyRooks.exe
```

...and then run the program using:

```
./SneakyRooks.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./SneakyRooks.exe > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt sample_output/output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt sample_output/output01.txt
1c1
< fail whale :(
---
> Hooray!
seansz@eustis:~$ _
```

```
gcc -c SneakyRooks.c
gcc SneakyRooks.c testcase01.c
```