Programming Assignment #5: SneakyRooks

COP 3502, Fall 2020

Due: Thursday, December 3, *before* 11:59 PM

Abstract

This is a wrap-up assignment designed to reinforce the kinds of algorithmic and clever thinking we've been building up together over the course of the semester. In particular, this will serve as an additional exercise in coming up with efficient solutions to problems, since your solution for this assignment needs to have a worst-case runtime that does not exceed O($m + n$) (linear runtime).

The assignment also involves a direct application of the base conversion material we covered recently (albeit with a minor twist).

You might find it a bit tricky to solve this problem within the runtime restriction 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

SneakyRooks.c

Note! The capitalization and spelling of your filename matter!

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

1. Overview

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

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:

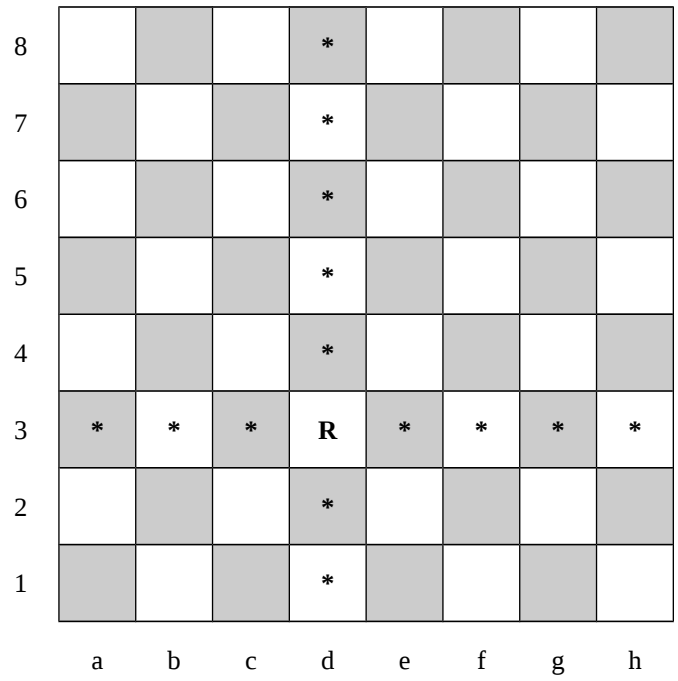

Figure 1: *The rook at position d3 can move to any square marked with an asterisk.*

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

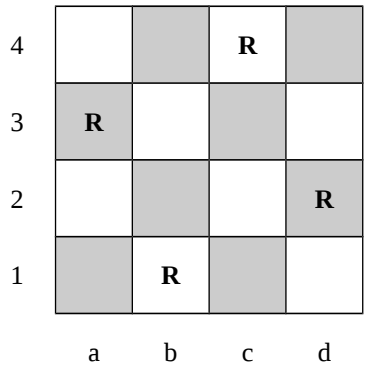

Figure 2: *A 4x4 board in which none of the rooks can attack one another.*

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

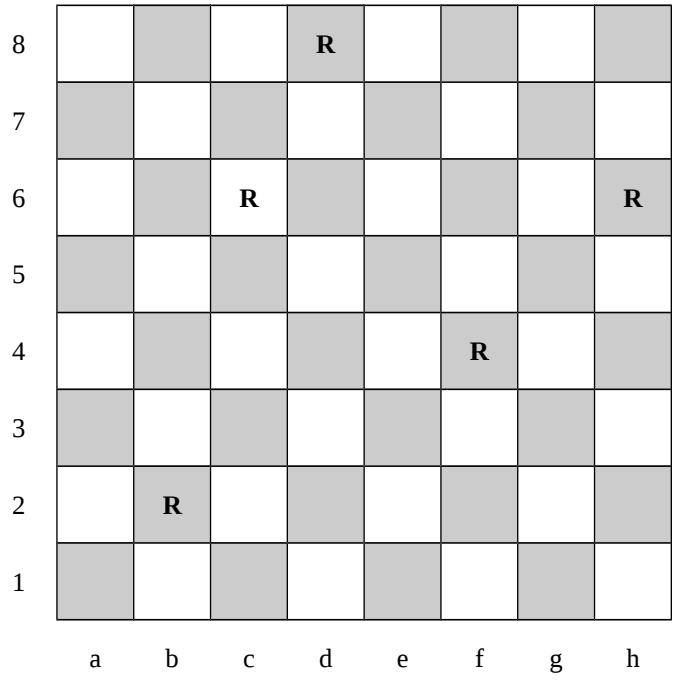

Figure 3: *An 8x8 board in which two of the rooks can attack one another.*

2. Chess Board Coordinates

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 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).

Converting these strings to their corresponding numeric coordinates is one of a few key algorithmic / mathemagical challenges you face in this assignment. You will have to write a function that does that for you.

3. Coordinate Struct (SneakyRooks.h)

To store rook coordinates, you must use the struct definition we have specified in *SneakyRooks.h* without any modifications. You **must** *#include* the header file in your *SneakyRooks.c* source file like so:

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

Note that the capitalization of *SneakyRooks.h* matters! Filenames are case sensitive in Linux, and that is of course the operating system we'll be using to test your code.

The struct you 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;
```

4. 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, 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.

5. Function Requirements

In the source file you submit, *SneakyRooks.c*, you must implement the following functions. You may implement any auxiliary functions you need to make these work, as well. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly. Please do **not** include a *main()* function in your submission.

```
int allTheRooksAreSafe(char **rookStrings, int numRooks, int boardSize);
```

> **Description:** Given an array of strings, each of which represents the location of a rook on a square *boardSize* × *boardSize* chess board, return 1 if none of the rooks can attack one another. Otherwise, return 0. You must do this in O(*numRooks* + *boardSize*) time. Be sure to avoid memory leaks.
>
> **Parameter Restrictions:** *boardSize* will be a positive integer describing both the length and width of the square board. (So, if *boardSize* = 8, then we have an 8 × 8 board.) *rookStrings* will be a non-NULL (but possibly empty) array of strings. Any strings within that array will be unique (there will be no repeats), and all of those strings are guaranteed to follow the format described above for valid coordinates on a *boardSize* × *boardSize* board. *numRooks* will be a non-negative integer indicating the number of strings in the *rookStrings* array.
>
> **Output:** This function should not print anything to the screen.
>
> **Runtime Requirement:** This function's runtime must be no worse than O(*numRooks* + *boardSize*). For details, see Section 4, "Runtime Requirements" (above). Note that repeated calls to C's built-in *pow()* function (or a home-brewed *pow()* function) would be ill-advised, because that function will not have an O(1) runtime. Instead, use Horner's Rule when calculating multiple powers of the same base.
>
> **Returns:** 1 if all the rooks are safe, 0 otherwise.

```
void parseCoordinateString(char *rookString, Coordinate *rookCoordinate);
```

> **Description:** Parse through *rookString* to determine the numeric row and column where the given rook resides on the chess board, and populate *rookCoordinate* with that information. You may assume that *rookString* is non-NULL, and that it contains a valid coordinate string using the format described above in Section 2, "Chess Board Coordinates." You may assume that *rookCoordinate* is non-NULL and is a pointer to an existing *Coordinate* struct.
>
> **Returns:** Nothing. This is a *void* function.

```
double difficultyRating(void);
```

> **Returns:** A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

> **Returns:** A reasonable and realistic estimate (greater than zero) of the number of hours you spent on this assignment.

6. Running All Test Cases on Eustis (*test-all.sh*)

The test cases included with this assignment are designed to show you some ways in which we might test your code and to shed light on the expected functionality of your code. We've also included a script, *test-all.sh*, that will compile and run all test cases for you.

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.

To run *test-all.sh* on Eustis, first transfer it to Eustis in a folder with *SneakyRooks.c*, *SneakyRooks.h*, all the test case files, and the *sample_output* directory. Transferring all your files to Eustis with MobaXTerm is fairly straightforward, but if you want to transfer them from a Linux or Mac command line, here's how you do it:

1. At your command line on your own system, use *cd* to go to the folder that contains all your files for this project (*SneakyRooks.c*, *SneakyRooks.h*, *test-all.sh*, the test case files, and the *sample_output* folder).
2. From that directory, type the following command (replacing `YOUR_NID` with your actual NID) to transfer that whole folder to Eustis:

   ```
   scp -r $(pwd) YOUR_NID@eustis.eecs.ucf.edu:~
   ```

   Warning: Note that the `$(pwd)` in the command above refers to your current directory when you're at the command line in Linux or Mac OS. The command above transfers the *entire contents* of your current directory to Eustis. That will include all subdirectories, so for the love of all that is good, please don't run that command from your desktop folder if you have a ton of files on your desktop!

Once you have all your files on Eustis, you can run *test-all.sh* by connecting to Eustis and typing the following:

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

If you put those files in their own folder on Eustis, you will first have to *cd* into that directory. For example:

```
cd SneakyRooksProject
```

That command (*bash test-all.sh*) will also work on Linux systems and with the bash shell for Windows. It will not work at the Windows Command Prompt, and it might have limited functionality in Mac OS.

Warning: When working at the command line, any spaces in file names or directory names either need to be escaped in the commands you type (`cd project\ 5`), or the entire name needs to be wrapped in double quotes.

7. Running the Provided Test Cases Individually

If the *test-all.sh* script is telling you that some of your test cases are failing, you'll want to compile and run those test cases individually to inspect their output. Here's how to do that:

1. Place all the test case files released with this assignment in one folder, along with your *SneakyRooks.c* file.

2. At the command line, *cd* to the directory with all your files for this assignment, and compile your source file with one of our test cases (such as *testcase01.c*) like so:

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

3. To run your program and redirect the output to *output.txt*, execute the following command:

```
./a.out > output.txt
```

4. Use *diff* to compare your output to the expected (correct) output for the program:

```
diff output.txt sample_output/testcase01-output.txt
```

   If the contents of *output.txt* and *testcase01-output.txt* are exactly the same, *diff* won't have any output:

```
seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
seansz@eustis:~$ _
```

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

```
seansz@eustis:~$ diff output.txt sample_output/testcase01-output.txt
1c1
< fail whale :(
---
> Hooray!
seansz@eustis:~$ _
```

Super Important: *Remember, 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.*

8. Testing for Memory Leaks with Valgrind

Part of the credit for this assignment will be awarded based on your ability to implement the program without any memory leaks. To test for memory leaks, you can use a program called *valgrind*, which is installed on Eustis.

Valgrind will **not** guarantee that your code is completely free of memory leaks. It will only detect whether any memory leaks occur when you run your program. So, if you have a function called *foo()* that has a nasty memory leak, but you run your program in such a way that *foo()* never gets called, *valgrind* won't be able to find that potential memory leak.

The *test-all.sh* script will automatically run your program through all test cases and use *valgrind* to check whether any of them result in memory leaks. If you want to run *valgrind* manually, simply compile your program with the *-g* flag, and then run it through *valgrind*, like so:

```
gcc SneakyRooks.c testcase01.c -g
valgrind --leak-check=yes ./a.out
```

In the output of *valgrind*, the magic phrase you're looking for to indicate that no memory leaks were detected is:

```
All heap blocks were freed -- no leaks are possible
```

For more information about *valgrind*'s output, see: http://valgrind.org/docs/manual/quick-start.html

9. Style Restrictions (***Super Important!***)

These are the same as in the previous assignment. 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:

- ★ 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 *#include* 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.
- ★ When defining a function that doesn't take any arguments, always put *void* in its parentheses. For example, define a function using *int do_something(void)* instead of *int do_something()*.
- ★ When defining or calling a function, do not leave a space before its opening parenthesis. For example:

use *int main(void)* instead of *int main (void)*. Similarly, use *printf("...")* instead of *printf ("...")*.

- ★ 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)*.
- ★ 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]*.)
- ★ 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.)

10. Special Restrictions (*Super Important!*)

1. As always, you must avoid the use of global variables, mid-function variable declarations, and system calls (such as `system("pause")`).
2. Do not read from or write to any files. File I/O is forbidden in this assignment.
3. Be sure you don't write anything in *SneakyRooks.c* that conflicts with what's given in *SneakyRooks.h*. Namely, do not try to define a *Coordinate* struct in *SneakyRooks.c*, since your source file will already be importing the definition of a node struct from *SneakyRooks.h*.
4. No shenanigans. For example, if you write an *allTheRooksAreSafe()* function that always returns 1, you might not receive any credit for the test cases that it happens to pass.
5. Your *SneakyRooks.c* file **must not** include a *main()* function. If it does, your code might fail to compile during testing, and you will not receive credit for this assignment.
6. Be sure to include your name and NID as a comment at the top of your source file.

11. Deliverable (Submitted via Webcourses, not Eustis)

Submit a single source file, named *SneakyRooks.c*, via Webcourses. The source file must contain definitions for all the required functions listed above. Be sure to include your name and NID as a comment at the top of your source file. Don't forget `#include "SneakyRooks.h"` in your source code (with correct capitalization). Your source file must work on Eustis with the *test-all.sh* script, and it must also compile on Eustis with both of the following:

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

Continued on the following page...

12. Grading

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've released 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:

100%	Passes test cases with correct output in linear time, with no memory leaks. (See notes below.)

Important Note! We are at a point in the semester where we should be able to take for granted that everyone will submit clean, beautiful code that is appropriately commented and abides by all style restrictions listed above. Accordingly, while you won't be *awarded* points for those things, significant point *deductions* may be imposed for poor commenting and whitespace practices or failure to adhere to style restrictions. You should also still include your name and NID in your source code.

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based primarily on your program's ability to compile and produce the *exact* results expected. Even minor deviations will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program thoroughly. 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.

Note also that your functions should not print anything to the screen. If they do, it will interfere with the output we generate while testing, resulting in incorrect test case results and an unfortunate loss of points.

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