1. Introduction

Your first programming assignment is a warmup designed to give you experience with two important aspects of computer systems programming: software design and software testing. In this assignment, you will leverage the basic concepts from lecture, ranging from variables and operators to conditional and iteration statements. More advanced concepts such as recursion will also play a key role in optimizing the performance of your code.

You will implement the `square root` math function twice: first using a simple but naive implementation, and then using a more sophisticated algorithm that can potentially improve performance by an order of magnitude. Throughout the assignment, you will carefully design your code to be maintainable (“Will somebody else be able to understand my code and extend it?”). We will leverage the CMake/CTest framework for unit testing, GitHub Actions for continuous integration testing, and Codecov.io for code coverage analysis.

After your code is functional and tested, you will write a four-page report that provides a discussion of your testing strategy and quantitatively evaluates the performance across three implementations. You should consult the programming assignment assessment rubric for more information about the expectations for all programming assignments and how they will be assessed. While the final code and report are all due at the end of the assignment, we also require meeting an incremental milestone in this PA. Specific requirements for this milestone are described at the end of this handout.

This handout assumes that you have read and understand the course tutorials and that you have attended the discussion sections. To get started, log in to an ecelinux server, source the setup script, and clone your individual remote repository from GitHub:

```
% source setup-ece2400.sh
% mkdir -p ${HOME}/ece2400
% cd ${HOME}/ece2400
% git clone git@github.com:cornell-ece2400/netid
% cd ${HOME}/ece2400/netid/pa1-math
% tree
```
Where netid should be replaced with your NetID. You can both pull and push to your individual remote repository. You should never fork your individual remote repository! If you need to work in isolation then use a branch within your individual remote repository. If you have already cloned your individual remote repository, then use git pull to ensure you have any recent updates before working on your programming assignment.

% cd $(HOME)/ece2400/netid
% git pull
% tree pa1-math

For this assignment, you will work in the pa1-math subproject. The initial version of the programming assignment includes the following files:

- src/ece2400-stdlib.h – Header file for course standard library
- src/ece2400-stdlib.c – Source code for course standard library
- src/sqrt-iter.h – Header file for iterative sqrt
- src/sqrt-iter.c – Source code for iterative sqrt
- src/sqrt-iter-adhoc.c – Ad-hoc test program for iterative sqrt
- src/sqrt-recur.h – Header file for recursive sqrt
- src/sqrt-recur.c – Source code for recursive sqrt
- src/sqrt-recur-adhoc.c – Ad-hoc test program for recursive sqrt

The programming assignment is divided into the two steps, and you should complete the first step before moving on to second. The first step is to implement and test the sqrt_iter function using an iterative algorithm. The second step is to implement and test the sqrt_recur function using a recursive algorithm.

2. Implementation Specifications

You will be implementing both iterative and recursive algorithms to compute the square root (\(\sqrt{x}\)) math function. The algorithm used for the iterative implementation of \(\sqrt{x}\) is simple but slow. The recursive algorithm is more complex but can potentially significantly improve performance compared to the iterative version.

The \(\sqrt{x}\) function takes one input argument \(x\) and returns its square root (i.e., \(\sqrt{x}\)). The corresponding C function has the following function signature:

\[
\text{int sqrt( int x )}
\]

Notice that the variant of sqrt that we will use in this assignment takes an integer input and returns another integer. The return value is the square root of \(x\) rounded down to the nearest integer. For example, calling sqrt(5) will return 2. If \(x\) is a negative value, the sqrt function must return -1 to report an invalid input. Your implementations should work correctly when the input is zero and when the input is any valid positive integer. Your implementations should not use any floating point arithmetic since this can introduce unnecessary performance overhead.

Note that your implementations cannot use anything from the Standard C library except for the printf function defined in stdio.h, the MIN/MAX macros defined in limits.h, the NULL macro defined in stddef.h, and the assert macro defined in assert.h.
ECE 2400 Computer Systems Programming, Fall 2021  PA1: Math Functions (Ad-Hoc Version)

Initial State
- Split the range [0, 144] into [0, 72] and [72, 144]
- Choose the lower range

Step 1
- Split the range [0, 72] into [0, 36] and [36, 72]
- Choose the lower range

Step 2
- Split the range [0, 36] into [0, 18] and [18, 36]
- Choose the lower range

Step 3
- Split the range [0, 18] into [0, 9] and [9, 18]
- Choose the upper range

Step 4
- Split the range [9, 18] into [9, 13] and [13, 18]
- Choose the lower range

Step 5
- Split the range [9, 13] into [9, 11] and [11, 13]
- Choose the upper range

Step 6
- Split the range [11, 13] into [11, 12] and [12, 13]
- Choose the lower range

Step 7
- Search the range [11, 12]
- Identify the square root of 144 to be 12

Step 8
- Identify the square root of 144 to be 12

Figure 1: Example of Recursive sqrt Algorithm – The range of integers that can contain the square root is halved at each step.

Iterative sqrt Implementation

The iterative implementation of the sqrt function should be implemented using an iteration statement. Let i range from zero to x. For each i, compute i × i and compare the result with x. If i × i is smaller than x, then i is less than the square root of x. If i × i is larger than x, then i is greater than the square root of x. By gradually checking all values of i, you will be able to find the square root of x rounded down to the nearest integer. Write your iterative implementation for sqrt inside of src/sqrt-iter.c.

Recursive sqrt Implementation

The iterative implementation of sqrt is particularly slow when x is large because: (1) the computer executes multiplication operations more slowly compared to simpler operations (e.g., addition, subtraction); and (2) the number of multiply operations increases linearly with \( \sqrt{x} \) since we are doing an exhaustive search. We can use a more sophisticated search to reduce the number of multiply operations. Consider the situation when x is 144. We can divide the search space into two ranges:

- Range of integers from 0 to \( \frac{x}{2} \), which is [0, 72] when x is 144
- Range of integers from \( \frac{x}{2} \) to x, which is [72, 144] when x is 144

We can quickly determine which half the square root of x lies in by squaring the midpoint (i.e., 72 × 72 = 5184) and comparing to x. Observing 5184 > 144 tells us that our guess of 72 was much too high, so the answer must be in the lower half (i.e., somewhere in the range [0, 72]), which is true since we know in this example that the square root is 12. We can continue applying the same approach on the smaller range, dividing the search space into smaller and smaller ranges. Figure 1 illustrates an example execution when x is 144. This approach allows us to quickly “zoom in” on the square root of x. We can capture this algorithm iteratively, but a recursive solution is also possible.
and may be more elegant and concise. The general approach of repeatedly halving the search space is known as a *binary search*. We will learn more about this class of algorithms in the future. Write your recursive implementation for sqrt inside of src/sqrt-recur.c. You may add additional helper functions inside this file as needed.

### 3. Testing Strategy

You are responsible for developing an effective testing strategy to ensure both implementations are correct. Writing tests is one of the most important and challenging aspects of software programming. Software engineers often spend far more time implementing tests than they do implementing the actual program.

Note that while there are limitations on what you can use from the Standard C library in your implementations there are no limitations on what you can use from the Standard C library in your testing strategy. You should feel free to use the Standard C library in your golden reference models and/or for random testing.

#### 3.1. Ad-hoc Testing

To help students start testing, we provide an ad-hoc test program for each implementation (e.g., src/sqrt-iter-adhoc.c). Students are encouraged to start compiling and running these ad-hoc test programs directly in the src directory without using any build or test frameworks (e.g., CMake or Make). You can build and run the given ad-hoc test programs like this:

```
% cd ${HOME}/ece2400/netid/pa1-math/src

% gcc -Wall -o sqrt-iter-adhoc ece2400-stdlib.c sqrt-iter.c sqrt-iter-adhoc.c
% ./sqrt-iter-adhoc

% gcc -Wall -o sqrt-recu-adhoc ece2400-stdlib.c sqrt-recu.c sqrt-recu-adhoc.c
% ./sqrt-recu-adhoc
```

The -Wall flag will ensure that gcc reports all warnings.

#### 3.2. Systematic Unit Testing

This section will be released after students learn more about C build and test frameworks in the discussion section.

#### 3.3. Code Coverage

This section will be released after students learn more about C build and test frameworks in the discussion section.

### 4. Evaluation

This section will be released after students learn more about C build and test frameworks in the discussion section.