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 verification. 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 design and implement the power and square root math functions 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 conduct individual code reviews via GitHub to help you improve the quality of the code you write. You will also use unit testing to carefully verify that your design is robust (“Will my code handle all situations as intended?”). Just as in the tutorial, we will leverage the CMake/CTest framework for unit testing, TravisCI for continuous integration testing, and Codecov.io for code coverage analysis.

After your code is functional and verified, you will write a four page report that describes your implementations, discusses your testing strategy, and evaluates the performance (and other trade-offs) of the iterative and recursive implementations. While the final code and report are all due at the end of the assignment, we also require meeting an incremental milestone of pushing your iterative implementations to GitHub on the date specified by the instructors. You should consult the programming assignment assessment rubric for more information about the expectations for all programming assignments and how they will be assessed.

This handout assumes that you have read and understand the course tutorials. To get started, log in to an ecelinux machine, 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>
```

Where <netid> should be replaced with your NetID. 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 running all of the tests. You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece2400/<netid>
% git pull --rebase
```
All of the tests should fail since you have not implemented the programming assignment yet. For this assignment, you will work in the pa1-math subproject, which includes the following files:

- `CMakeLists.txt` - CMake configuration script to generate Makefile
- `src/pow-iter.c` - Source code for iterative `pow`
- `src/pow-iter.h` - Header file for iterative `pow`
- `src/pow-iter-eval.c` - Evaluation program for iterative `pow`
- `src/pow-iter-main.c` - Ad-hoc test program for iterative `pow`
- `src/pow-recur.c` - Source code for recursive `pow`
- `src/pow-recur.h` - Header file for recursive `pow`
- `src/pow-recur-eval.c` - Evaluation program for recursive `pow`
- `src/pow-recur-main.c` - Ad-hoc test program for recursive `pow`
- `src/sqrt-iter.c` - Source code for iterative `sqrt`
- `src/sqrt-iter.h` - Header file for iterative `sqrt`
- `src/sqrt-iter-eval.c` - Evaluation program for iterative `sqrt`
- `src/sqrt-iter-main.c` - Ad-hoc test program for iterative `sqrt`
- `src/sqrt-recur.c` - Source code for recursive `sqrt`
- `src/sqrt-recur.h` - Header file for recursive `sqrt`
- `src/sqrt-recur-eval.c` - Evaluation program for recursive `sqrt`
- `src/sqrt-recur-main.c` - Ad-hoc test program for recursive `sqrt`
- `src/pow-std-eval.c` - Evaluation program for standard `pow`
- `src/sqrt-std-eval.c` - Evaluation program for standard `sqrt`
- `tests/pow-iter-basic-tests.c` - Basic test cases for iterative `pow`
- `tests/pow-iter-directed-tests.c` - Directed test cases for iterative `pow`
- `tests/pow-iter-random-tests.c` - Random test cases for iterative `pow`
- `tests/pow-recur-basic-tests.c` - Basic test cases for recursive `pow`
- `tests/pow-recur-directed-tests.c` - Directed test cases for recursive `pow`
- `tests/pow-recur-random-tests.c` - Random test cases for recursive `pow`
- `tests/sqrt-iter-basic-tests.c` - Basic test cases for iterative `sqrt`
- `tests/sqrt-iter-directed-tests.c` - Directed test cases for iterative `sqrt`
- `tests/sqrt-iter-random-tests.c` - Random test cases for iterative `sqrt`
- `tests/sqrt-recur-basic-tests.c` - Basic test cases for recursive `sqrt`
- `tests/sqrt-recur-directed-tests.c` - Directed test cases for recursive `sqrt`
- `tests/sqrt-recur-random-tests.c` - Random test cases for recursive `sqrt`
- `tests/utst.h` - Helper macros and functions used for unit testing
- `src/pow.dat` - Input dataset for `pow` evaluation
- `src/sqrt.dat` - Input dataset for `sqrt` evaluation
- `scripts` - Scripts for the build system and generating datasets
2. Implementation Specifications

In this project, you will be implementing both iterative and recursive algorithms to compute the power and square root math functions. The algorithms used for the iterative implementations of pow and sqrt are simple but slow. The recursive algorithms are more sophisticated but can potentially improve performance by an order of magnitude compared to their iterative versions.

The name of our power function will be pow, and the name of our square root function will be sqrt. The pow function takes two input arguments: (1) the base value \( b \), and (2) the exponent value \( e \). The function returns the base raised to the power of the exponent (i.e., \( b^e \)). Specifically, the corresponding C function has the following function signature:

\[
pow( \text{double base, int exponent} )
\]

Notice that base is a floating-point double (i.e., a real number), exponent is an integer, and the returned value is also a floating-point double. A basic implementation of the pow function expressed in a mathematical formula is:

\[
b^e = \begin{cases} 
1 & \text{if } e = 0 \\

b \times b \times \cdots \times b & \text{if } e > 0 \\
1/(b \times b \times \cdots \times b) & \text{if } e < 0 
\end{cases}
\]

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

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

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.

2.1. Iterative pow Implementation

Your iterative implementation of the pow function should directly correspond to the above definition and should use an iteration statement with a few carefully chosen conditional statements.

In this assignment, we will allow certain cases in the implementation of pow to remain undefined:

- A base of 0 raised to a negative exponent is undefined
- A base of 0 raised to the 0\(^{th} \) power is undefined
- Any cases involving overflowing a double are undefined

We will not test these cases when we grade the assignment, and students do not need to handle these special cases in their implementation. Write your iterative implementation for pow inside of src/pow-iter.c.

2.2. 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 \( i \times i \) and compare the result with \( x \). If \( i \times i \)
is smaller than $x$, then $i$ is less than the square root of $x$. If $i \times i$ is larger than $x$, then $i$ is greater than the square root of $x$. By gradually testing 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`.

2.3. Recursive `pow` Implementation

The iterative implementation of `pow` is particularly slow when the exponent 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 $e$. We can exploit structure in the iterative implementation to reduce the required number of multiply operations. Consider the following example:

$$3^8 = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$$

We can reduce the number of multiply operations by reusing intermediate results.

$$a = 3 \times 3$$
$$b = a \times a$$
$$c = b \times b$$

We first calculate $3^2 = 3 \times 3$, and we can then reuse $3^2$ to calculate $3^4 = (3^2)^2$, and we can then reuse $3^4$ to calculate $3^8 = ((3^2)^2)^2$. The iterative implementation would require seven multiply operations, while the recursive implementation requires only three multiply operations. Here is the recursive implementation expressed as a mathematical formula:

$$b^e = \begin{cases} 
1 & \text{if } e = 0 \\
(b^2)^{e/2} & \text{if } e > 0 \text{ and } e \text{ is even} \\
b \times b^{e-1} & \text{if } e > 0 \text{ and } e \text{ is odd} \\
1/(b^{|e|}) & \text{if } e < 0 
\end{cases}$$

Notice that this approach for computing `pow` has multiple recursive cases. Although an iterative solution using loops is possible, we can more elegantly and concisely capture this algorithm using a recursive helper function. Write your recursive implementation for `pow` inside of `src/pow-recur.c`. You may add additional helper functions inside this file as needed.

2.4. Recursive `sqrt` Implementation

The iterative implementation of `sqrt` is particularly slow when $x$ is large because: (1) as mentioned above, 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 $x$ 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., $72 \times 72 = 5184$) 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 "zero 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 \( \text{sqrt} \) inside of \texttt{src/sqrt-recur.c}. You may add additional helper functions inside this file as needed.

When squaring large integers, it is possible to exceed the maximum value an integer type can hold. This is called integer overflow and can corrupt the data in your program. It is relatively straightforward to detect overflow before it happens. We have provided a short helper function named \texttt{detect\_overflow\_before\_squaring} that takes an integer and returns 1 if \( x^2 \) will cause overflow and returns 0 if it will not. Please use this helper function before squaring integers to avoid corrupting the data in your program.

While you are required to implement the recursive implementations described in this section, students should also feel free to experiment with additional implementations. These implementations should be kept separate by using \texttt{pow-extra} and \texttt{sqrt-extra} prefixes. Students will have to modify the \texttt{CMakeLists.txt} accordingly, and they will also need to ensure that any additional implementations are both tested and evaluated.
3. Testing Strategy

You are responsible for developing an effective testing strategy to ensure all 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.

3.1. Ad-hoc Testing

To help students start testing, we provide one ad-hoc test program per implementation (e.g., src/pow-iter-main.c). Students are strongly encouraged to start compiling and running these ad-hoc test programs directly in the src/ directory without using any build-automation tool (e.g., CMake and Make).

You can build and run the given ad-hoc test program for pow-iter like this:

```
% cd ${HOME}/ece2400/<netid>/pa1-math/src
% gcc -Wall -o pow-iter-main pow-iter.c pow-iter-main.c
% ./pow-iter-main
```

The -Wall command line option will ensure that gcc reports all warnings.

3.2. Systematic Unit Testing

While ad-hoc test programs help you quickly see results of your implementations, they are often too simple to cover most scenarios. We need a systematic unit testing strategy to hopefully test all possible scenarios efficiently.

In this course, we are using CMake/CTest as a build and test automation tool. For each implementation, we provide a basic test that checks the most basic functionality, a couple of directed tests that target different categories, and random tests. You will need to add more directed and random tests to thoroughly test your implementations.

As you design your implementations, pay careful attention to corner cases and unexpected inputs (e.g., negative inputs) that break the functionality of your code. When you encounter such a case, capture the situation with a test case and a helpful error message and improve your code. Carefully read the implementation specification (i.e., the inputs, the outputs, and the behavior), so you know how your program should respond in all possible scenarios. Convince yourself that your implementations are robust by carefully developing a testing strategy.

In addition to writing directed tests, you should also add random tests. You can randomly generate inputs using the rand function in the standard C library (include stdlib.h). Use the srand function to initialize the random seed to a deterministic value to ensure your random tests are repeatable. You can use the pow and sqrt functions in the standard C library (include math.h) as golden reference models to generate correct reference outputs which you can then compare to the results from your own implementations. Note that you are not allowed to use the pow and sqrt functions in the standard C library for your implementation, only for verification.

When testing the pow function, note that the result can vary by very small amounts because of precision errors that build up as the computer performs arithmetic on real numbers. This is commonly solved by comparing relative amounts (i.e., checking that the two numbers are within at least 99.99% of each other). An example test written in this fashion has been provided for you in tests/pow-iter-basic-tests.c and tests/pow-recur-basic-tests.c. If precision error becomes a problem, please compare numbers relatively.
Before running the tests you need to create a separate build directory and use cmake to create the Makefile like this:

```
% cd ${HOME}/ece2400/<netid>/pa1-math
% mkdir -p build
% cd build
% cmake ..
```

Now you can build and run all unit tests for all implementations like this:

```
% cd ${HOME}/ece2400/<netid>/pa1-math/build
% make check
```

You can focus and run all of the unit tests for a single implementation (e.g., pow-iter) like this:

```
% cd ${HOME}/ece2400/<netid>/pa1-math/build
% make check-pow-iter
```

You can run just a single unit test for a single implementation (e.g., pow-iter-basic-tests) like this:

```
% cd ${HOME}/ece2400/<netid>/pa1-math/build
% make check-pow-iter-basic-tests
```

### 3.3. Code Coverage

After your implementations pass all unit tests, you can evaluate how effective your test suite is by measuring its code coverage. The code coverage will tell you how much of your source code your test suite executed during your unit testing. The higher the code coverage is, the less likely some bugs have not been detected.

You can run the code coverage like this:

```
% cd ${HOME}/ece2400/<netid>/pa1-math
% mkdir -p build
% cd build
% cmake ..
% make coverage
```

### 4. Evaluation

Once you have verified the functionality of the iterative and recursive implementations, you can then start to evaluate the performance of these implementations. We provide you a performance analysis harness for each implementation. In addition, we also provide you two performance analysis harnesses for the pow and sqrt functions provided in the standard math library. You can build and run these evaluation programs like this:

```
% cd ${HOME}/ece2400/<netid>/pa1-math/build-eva
% mkdir -p pa1-math/build-eval
% cd pa1-math/build-eval
% cmake ..
% make eval-pow-iter
% make eval-sqrt-iter
```
Note how we are working in a separate build-eval build directory, and that we are using the `-DCMAKE_BUILD_TYPE=RELEASE` command line option to the `cmake` script. This tells the build system to create optimized executables without any extra debugging information. You can build and run all of the evaluation programs in a single step like this:

```
% cd ${HOME}/ece2400/<netid>/pa1-math/build-eval
% make eval
```

The evaluation programs apply your math functions to 1000 inputs and report the total wall-clock run time. This will enable you to compare the performance between your iterative algorithms, recursive algorithms, and the implementations provided in the standard `math` library. The evaluation programs also ensure that your implementations are producing the correct results, however, you should not use the evaluation programs for testing. If your implementations fail during the evaluation, then your testing strategy is insufficient. You must add more unit tests to effectively test your program before returning to performance evaluation.

5. Incremental Milestone

While the final code and report are all due at the end of the assignment, we also require meeting an incremental milestone of pushing your iterative implementations to GitHub on the date specified by the instructor. More specifically to meet the incremental milestone of this PA, you are expected to:

- Complete two iterative implementations `pow-iter` and `sqrt-iter`
- Pass all given basic and directed tests for both implementations
- Add a few more directed and random test cases for both implementations

Acknowledgments

This programming assignment was created by Christopher Batten, Jose Martínéz, Christopher Torng, Xiaodong Wang, Shuang Chen, Shunning Jiang, and Tuan Ta as part of the ECE 2400 Computer Systems Programming course at Cornell University.