ECE 2400 Computer Systems Programming, Fall 2021
PA1: Math Functions

School of Electrical and Computer Engineering
Cornell University

revision: 2021-09-25-15-07

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 significantly improve performance. We will leverage the CMake/CTest framework for unit testing, and GitHub Actions for continuous integration testing.

After your code is functional and tested, you will write a four-page report that includes a discussion of your testing strategy and a quantitative evaluation of the performance across three implementations. **You should consult the programming assignment logistics document 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. Requirements specific to this PA for the incremental milestone and the final submission 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, which includes the following files:

• `CMakeLists.txt`	– CMake configuration script to generate Makefile
• `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`
• `eval/sqrt-iter-eval.c`	– Evaluation program for iterative `sqrt`
• `eval/sqrt-recur-eval.c`	– Evaluation program for recursive `sqrt`
• `eval/sqrt-std-eval.c`	– Evaluation program for standard C library `sqrt`
• `test/sqrt-iter-directed-test.c`	– Directed test cases for iterative `sqrt`
• `test/sqrt-iter-random-test.c`	– Random test cases for iterative `sqrt`
• `test/sqrt-recur-directed-test.c`	– Directed test cases for recursive `sqrt`
• `test/sqrt-recur-random-test.c`	– Random test cases for recursive `sqrt`

The programming assignment is divided into the following steps. Complete each step before moving on to the next step.

- Step 1. Implement and test `sqrt_iter`
- Step 2. Implement and test `sqrt_recur`
- Step 3. Evaluate all implementations

2. Interface and Implementation Specifications

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

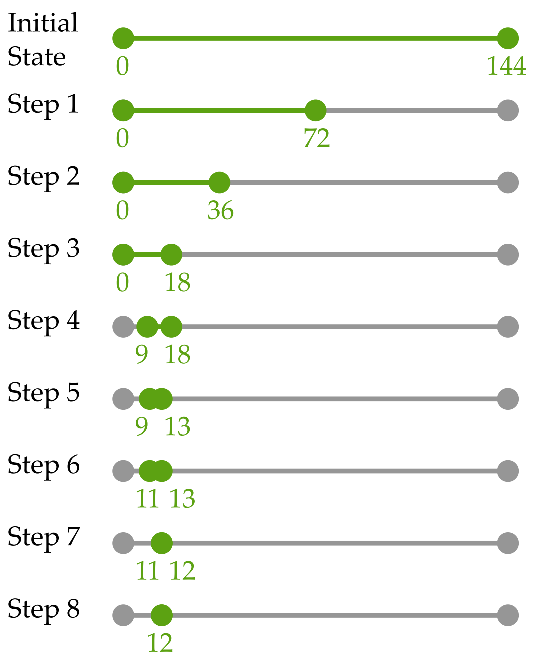
Note that your implementations cannot use anything from the Standard C library except for the `printf` function defined in `stdio.h`, the MIN/MAX macros defined in `limits.h`, and the `assert` macro defined in `assert.h`.

`sqrt` **Interface**

The `sqrt` function will have the following prototype:

```
int sqrt( int x );
```

The function takes one input argument (x) and returns its square root (i.e., $\sqrt{x}$). 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. Your implementations should not use any magic



- Split the range $[0, 144]$ into $[0, 72]$ and $[72, 144]$
- Choose the lower range
- Split the range $[0, 72]$ into $[0, 36]$ and $[36, 72]$
- Choose the lower range
- Split the range $[0, 36]$ into $[0, 18]$ and $[18, 36]$
- Choose the lower range
- Split the range $[0, 18]$ into $[0, 9]$ and $[9, 18]$
- Choose the upper range
- Split the range $[9, 18]$ into $[9, 13]$ and $[13, 18]$
- Choose the lower range
- Split the range $[9, 13]$ into $[9, 11]$ and $[11, 13]$
- Choose the upper range
- Split the range $[11, 13]$ into $[11, 12]$ and $[12, 13]$
- Choose the lower range
- Search the range $[11, 12]$
- 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.

numbers. Note that you can use `INT_MAX` and `INT_MIN` from `limits.h` if you need to determine the largest and smallest value that can be stored in a variable of type `int`.

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 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., $72 \times 72 = 5184$) 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 "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.

$$b^e = \begin{cases} 1 & \text{if } e = 0 \\ b \times b \times \overset{e}{\cdots} \times b & \text{if } e > 0 \end{cases}$$

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

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.

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-recur-adhoc ece2400-stdlib.c sqrt-recur.c sqrt-recur-adhoc.c
% ./sqrt-recur-adhoc
```

The `-Wall` flags will ensure that `gcc` reports most warnings.

3.2. Systematic Unit Testing

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

In this course, we are using CMake/CTest as our build and test automation framework. For each implementation, we provide a directed test program that should include several test cases to target different categories and a random test program that should test that your implementation works for random inputs. **We only provide a very few directed tests and no random tests. You must add many more directed and random tests to thoroughly test your implementations!**

A directed test case involves manually pre-computing the output for a specific set of inputs, and then verifying that your implementation produces this desired output. Start by writing as many directed test cases as you can for some simple and more complex inputs. 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 directed test case and verify your implementation now passes that test case. 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 to increase your confidence in the correctness of your implementation. 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 `sqrt` function in the standard C library (include `math.h`) as a golden reference model 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 `sqrt` function in the standard C library for your implementation, only for testing.

We provide you a testing framework you should use for your directed and random testing. See the provided test programs in the `test` subdirectory for how to use this framework. The ECE 2400 standard library in `ece2400-stdlib.h` contains the following macros you should use to check the correctness of your implementations:

- `ECE2400_CHECK_FAIL()` – check program does not reach this point
- `ECE2400_CHECK_TRUE( expr_ )` – check `expr_` is always true
- `ECE2400_CHECK_FALSE( expr_ )` – check `expr_` is always false
- `ECE2400_CHECK_INT_EQ( expr0_, expr1_ )` – check `expr0_` equals `expr1_`

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
```

If you are failing a test program, then you can "zoom in" and run all of the unit tests for a single test program (e.g., directed tests for `sqrt-iter`) like this:

```
% cd ${HOME}/ece2400/netid/pa1-math/build
% make sqrt-iter-directed-test
% ./sqrt-iter-directed-test
```

You can then "zoom in" to a specific test case by passing in the index of that test case like this:

```
% cd ${HOME}/ece2400/netid/pa1-math/build
% make sqrt-iter-directed-test
% ./sqrt-iter-directed-test 1
% ./sqrt-iter-directed-test 2
```

3.3. Test-Case Crowd Sourcing

While a comprehensive test suite provides strong evidence that your implementation has the correct functionality, it is particularly challenging to write high-quality test cases for all of your implementations. **Students can use test-case crowd-sourcing after the milestone to reduce the workload of constructing a comprehensive test suite.** Test-case crowd-sourcing will use a Canvas discussion page; students cannot see any of the currently posted test cases until they post one of their own. Focus on uploading one or two very strong directed or random test cases. Do not upload more than two test cases. Avoid uploading simple directed test cases since students will have already developed such test cases for the milestone. Posting the basic test case provided by the course instructors, posting an obviously too simple test case, and/or posting something which is obviously meant to "game" the system is not allowed. Let's all work together to crowd-source a great test suite that every student can take advantage of!

You can use test cases posted in the Canvas discussion page in your test programs as long as you acknowledge the author, so be sure to include the comment in your source code which describes the test case and includes the author's name. You will need to renumber the test cases and call them correctly from `main()`. **Make sure you understand the test case and that you feel it is testing correct behavior before including it in your test suite!**

3.4. 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
% rm -rf build-coverage
% mkdir -p build-coverage
% cd build-coverage
% cmake ..
% make check
% make coverage
```

Note that these code coverage results will reflect *all* prior runs of the test and evaluation programs in the build directory. That is why in the above example, we do a fresh build in a separate `build-coverage` build directory.

If you want to drill down and explore the coverage of each line in a program you use use the elinks web browser like this:

```
% cd ${HOME}/ece2400/netid/pa2-dstruct/build-coverage
% elinks coverage-html/index.html
```

Code coverage is just one more piece of evidence you can use to make a compelling case for the correct functionality of your implementations. It is not required that students achieve 100% code coverage. It is far more important that students simply use code coverage as a way to guide their test-driven design than to overly focus on the specific code coverage number.

4. Evaluation

Once you have tested the functionality of the iterative and recursive implementations, you can then start to evaluate the performance of these implementations. We provide you an evaluation program for each implementation in the `eval` subdirectory. In addition, we also provide you an evaluation program for the `sqrt` function provided in the standard `math` library. You should not need to modify the evaluation programs. The ECE 2400 standard library in `ece2400-stdlib.h` contains the following functions that are used in the evaluation programs to measure the execution time:

- `ece2400_timer_reset()` – reset the global timer
- `ece2400_timer_get_elapsed()` – return the elapsed time in seconds since last reset

You can build the evaluation programs like this:

```
% cd ${HOME}/ece2400/netid/pa1-math
% rm -rf build-eval
% mkdir -p build-eval
% cd build-eval
% cmake -DCMAKE_BUILD_TYPE=eval ..
% make eval
```

Note how we are working in a separate `build-eval` build directory, and that we are using the `-DCMAKE_BUILD_TYPE=eval` command line option to the `cmake` script. This tells the build system to create optimized executables without any extra debugging information. **You must do your quantitative evaluation using an eval build. Using a debug build for evaluation produces meaningless results.**

To run an evaluation, you simply specify the inputs on the command line. For example, the following runs an evaluation for one of the `sqrt` implementations to find the square root of 100.

```
% cd ${HOME}/ece2400/netid/pa1-math/build-eval
% make eval
% ./sqrt-iter-eval 100
```

The evaluation programs apply your math functions to the input you specify at the command line in a loop and report the total wall-clock runtime. 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.

You should quantitatively evaluate all three evaluations for a range of values. We suggest evaluating your `sqrt` implementations from zero to one million with a reasonable number of intermediate points as long as the implementation doesn't take too long to run. Record all of this performance data.

5. Milestone and Final Submission

This section includes critical information about the incremental milestone, final code submission, and the final report specific to this PA. **The programming assignment logistics document provides general details about the requirements for the milestone and final submission.** You must actually read the document to ensure you know how we will access your milestone and final submission.

5.1. Incremental Milestone

While the final code and report are all due at the end of the assignment, we also require you to complete an incremental milestone and push your code to GitHub by the date specified by the instructor. In this PA, to meet the incremental milestone, you will need to first complete the iterative implementation of `sqrt` and then write an extensive test suite including many directed and random tests for this implementation. Here is how we will be testing your milestone:

```
% mkdir -p ${HOME}/ece2400/submissions
% cd ${HOME}/ece2400/submissions
% rm -rf repo
% git clone git@github.com:cornell-ece2400/netid

% cd ${HOME}/ece2400/submissions/netid/pa1-math
% mkdir -p build
% cd build
% cmake ..
% make check-milestone
```

5.2. Final Code Submission

Your code quality score will be based on the way you format the text in your source files, proper use of comments, deletion of instructor comments, and uploading the correct files to GitHub (only source files should be uploaded, no generated build files). To assist you in formatting your code correctly, we have created a make target that will autoformat the code for you. You can use it like this:

```
% cd ${HOME}/ece2400/netid/pa1-math
% mkdir -p build
% cd build
% cmake ..
% make autoformat
% git diff
# ... check all changes ...
% git commit -a -m "autoformat"
```

Note that the `autoformat` target will only work if you have already committed all of your work. This way you can easily use `git diff` to view the changes made by the autoformatting and commit those changes when you are happy with them. Since we provide students an automated way to format their code correctly, students have no excuse for not following the course coding conventions!

Note that students must remove unnecessary comments that are provided by instructors in the code distributed to students. Students must not commit executable binaries or any other unnecessary files. The `autoformat` target will not take care of these issues for you.

To submit your code you simply upload it to GitHub. Your code will be assess both in terms of functionality and code quality. Your functionality score will be determined by running your code against a series of tests developed by the instructors to test its correctness. Here is how we will be testing your final code submission:

```
% mkdir -p ${HOME}/ece2400/submissions
% cd ${HOME}/ece2400/submissions
% rm -rf repo
% git clone git@github.com:cornell-ece2400/netid

% cd ${HOME}/ece2400/submissions/netid/pa1-math
% mkdir -p build
% cd build
% cmake ..
% make check
% make eval
# ... run the eval programs ...
```

5.3. Final Report

The final report must be uploaded to Canvas. The date you upload your report will indicate how many slip days you are using for the assignment. For this PA, we require you to include four sections: introduction, testing strategy, quantitative evaluation, and conclusion.

The quantitative evaluation section of your report must include a performance plot. The plot should have the input to `sqrt` on the x-axis and the wall-clock runtime on the y-axis. Plot a line for each of the three implementations of `sqrt`. Ensure your plot is easy to read with a legend, reasonable font sizes, and appropriate colors/markers for black-and-white printing. You must discuss these results in the quantitative evaluation section.

Acknowledgments

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