This is the ad-hoc version of the PA1 handout. The full version will include additional information about using the build, test, and evaluation frameworks. These frameworks can be a little overwhelming to new programmers. So the initial version of the PA1 handout guides students through the implementation specification and ad-hoc testing. We want students to initially focus on using this kind of simple ad-hoc testing to get started. We will post the full version of the handout once we introduce the build, test, and evaluation frameworks in the discussion section. Note that students will be required to use these frameworks when submitting their programming assignment.

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 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 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 three-page report that provides a detailed deep-dive discussion on one implementation, qualitatively evaluates trade-offs across four implementations, and quantitatively evaluates the performance across six implementations. 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 later in this handout. You should consult the programming assignment assessment rubric for more information about the expectations for all programming assignments and how they will be assessed. For this PA, we require you to discuss your recursive implementation of square root as your implementation deep dive in the report.

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 machine, source the setup script, and clone your individual remote repository from GitHub:

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

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

- `src/pow-iter.c` – Source code for iterative `pow`
- `src/pow-iter.h` – Header file for iterative `pow`
- `src/pow-iter-adhoc.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-adhoc.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-adhoc.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-adhoc.c` – Ad-hoc test program for recursive `sqrt`

## 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 complex but can potentially improve performance by an order of magnitude compared to their iterative versions.

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:

```
double pow( 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:

\[
b^e = \begin{cases} 
1 & \text{if } e = 0 \\
 b \times b \times \cdots \times b & \text{if } e > 0 \\
 1/(b \times b \times \cdots \times b) & \text{if } e < 0 
\end{cases}
\]
In this assignment, we will allow certain cases in the implementation of `pow` to remain undefined:

- The result for $0^e$ where $e \leq 0$ is undefined
- The result when $b < -1000$ or when $b > 1000$ is undefined
- The result when $e < -400$ or when $e > 400$ is undefined
- The result when $b^e > 10^{300}$ is undefined
- The result when $b^e < -10^{300}$ is 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.

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:

```c
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 implementation should work correctly when the input is zero and when the input is any valid positive integer.

### 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. 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 power 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 $x/2$, which is $[0, 72]$ when $x$ is 144
- Range of integers from $x/2$ 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 sqrt inside of src/sqrt-recur.c. You may add additional helper functions inside this file as needed.

### 2.5. Other Implementations

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 pow-extra and sqrt-extra prefixes. Students will have to modify the CMakeLists.txt accordingly, and they will also need to ensure that any additional implementations are both tested and evaluated.
• 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.

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-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 and Make). You can build and run the given ad-hoc test program like this:

% cd ${HOME}/ece2400/netid/pa1-math/src
% gcc -Wall -Wextra -pedantic -o pow-iter-adhoc pow-iter.c pow-iter-adhoc.c
% ./pow-iter-adhoc

% gcc -Wall -Wextra -pedantic -o pow-recur-adhoc pow-recur.c pow-recur-adhoc.c
% ./pow-recur-adhoc

% gcc -Wall -Wextra -pedantic -o sqrt-iter-adhoc sqrt-iter.c sqrt-iter-adhoc.c
% ./sqrt-iter-adhoc
% gcc -Wall -Wextra -pedantic -o sqrt-recur-adhoc sqrt-recur.c sqrt-recur-adhoc.c
% ./sqrt-recur-adhoc

The -Wall, -Wextra, -pedantic-errors flags will ensure that gcc reports all warnings.

3.2. Systematic Unit Testing

This section will be released after we learn more about testing in the discussion section.

3.3. Code Coverage

This section will be released after we learn about code coverage in the discussion section.

4. Evaluation

This section will be released after we learn more about evaluation in the discussion section.

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 the iterative implementation of pow
• Complete the iterative implementation of sqrt
• Pass all given basic and directed tests for both implementations
• Thoroughly test both implementations by adding your own directed and random tests

Acknowledgments

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