ECE 209
Fall 2021

Program 1: `roots`

Due Friday, Sep 17 @ 11:59pm

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of 0 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

DO NOT copy code from the Internet, or use programs found online or in textbooks as a "starting point" for your code. Your job is to design and write this program from scratch, on your own. Evidence of using external code from any source will be investigated as a potential academic integrity violation.

This program uses the *sliding-digit* algorithm to compute the *n*-th root of an integer to a specified number of digits past the decimal point. We use integer arithmetic for the calculation. Similar to long division, one digit of the root is produced with each iteration.

The learning objectives of the program are:

- Write a complete C program.
- Use loops and conditional statements.
- Use printf and scanf to perform I/O.
- Call standard C library functions.

Background

Given a number A, an n-th root y is a number such that $x^n = A$. There are many algorithms to compute the root, but we will use the *sliding-digit* algorithm. Similar to long division, we choose one digit at a time of the answer. This is not the most efficient or most effective way to calculate, but it's an interesting algorithm to implement, and there will be a few learning opportunities along the way. The following Wikipedia article provides an in-depth explanation:

https://en.wikipedia.org/wiki/Shifting_nth_root_algorithm

Problem Description

For this problem, the starting number A must be between 2 and 1,000,000. (The roots of 1 are all 1, so that's not interesting.) The value of n will also be an integer, greater than or equal to 2. There is no specific upper limit, but we'll see that practically it cannot be very large.

The root x is in generally not an integer, but we choose to only use integer arithmetic to avoid issues with precision and rounding. The user will specify k, the desired number of digits to the right of the decimal point. The calculated solution y will be $x \cdot 10^k$, truncated to an integer.

For example, the square root of 2 is approximately 1.414214. With k=3, the calculated result y = 1414, which corresponds to x = 1.414. The bottom k digits of y are interpreted as being to the right of the decimal point. This is known as a fixed-point representation.

The algorithm works with any base B, but we will always use base-10 arithmetic.

In summary, the algorithm uses the following values:

Value	*Description*	*Type*	*Source*
A	The number to take the root of.	Integer	User
n	Which root to calculate.	Integer	User
k	Desired precision, number of decimal places.	Integer	User
x	The estimated n-th root, with k decimal places. (Truncated, not rounded.)	Floating-point	Calculated by program.
y	x times 10^k	Integer	Calculated by program.

The integer A is divided into an integral number of n-digit groups, with leading zeroes added as needed. For example, if we are taking the 3rd root of 1234, the input number is grouped as 001 234. The algorithm starts with the left-most grouping, which would be 001 in this example.

Each group of A will generate one digit of the solution. Therefore, if there are g groups in A, and k decimal places, the total number of digits in the solution y will be g+k. This is also the number of iterations required in the algorithm.

The algorithm is as follows:

```
Initialize $y = 0$, and $r = 0$.
Repeat until the desired number of digits is produced:
    $\alpha$ = the next group of $n$ digits. (If no more digits in $A$, then $\alpha$ = 0.)
    $r = 10^n r + \alpha$
    Choose $\beta$ as the largest integer such that $0 \le \beta \le 9$ and $(10y + \beta)^n - (10y)^n \le r$
    $y = 10y + \beta$
    $r = r - ((10y + \beta)^n - (10y)^n)$
    If $r < 0$, set $r = 0$. (This indicates an underflow condition, where $r$ is not large enough for
    the calculation.)
```

First, note that $10y$ is the same as a left shift of y by one digit in base-10. Likewise, $10^n r$ is equivalent to shifting r to the left by n digits. Therefore, the second step inside the loop is the same as "appending" the next n-digit group to r.

Let's walk through the algorithm with a couple of examples.

Square root of 2: $A = 2$, $n = 2$, $k = 3$

$y = 0$ and $r = 0$

There is one 2-digit group (02) and three decimal digits, so there will be 4 iterations of the loop.

Step 1:

α is 02 (or just 2), so r becomes $100 \times 0 + 2 = 2$
Since y is zero, we're just looking for $\beta^2 \leq 2$, so $\beta = 1$.
$y = 0 + 1$
$r = 2 - 1 = 1$

Step 2:

Since there are no more 2-digit groups, $\alpha = 0$ and $r = 100r = 100$
$\beta = 4$, because $(14)^2 - (10)^2 \leq 100$.
$y = 14$
$r = 100 - (196 - 100) = 4$

Step 3:

$r = 400$
$\beta = 1$, because $(141)^2 - (140)^2 \leq 400$
$y = 141$
$r = 400 - (19{,}881 - 19{,}600) = 119$

Step 4:

$r = 11{,}900$
$\beta = 4$, because $(1{,}414)^2 - (1410)^2 \leq 19{,}900$
$y = 1{,}414$
(don't care about *r*, because this is the last step)

So our computed value is 1414, meaning that the estimated root is 1.414.

Cube root of 4,913: A = 4,913, n = 3, k = 1

$y = 0$ and $r = 0$

There are two 3-digit groups and one decimal digits, so there will be 3 iterations of the loop.

Step 1:

$\alpha = 4, r = 4$
$\beta^3 \leq 4$, so $\beta = 1$.
$y = 0 + 1$
$r = 4 - 1 = 3$

Step 2:

$\alpha = 913, r = 3{,}913$
$\beta = 7$, because $(17)^3 - (10)^3 \leq 3{,}913$
$y = 17$
$r = 3{,}913 - (4{,}913 - 1{,}000) = 0$

Step 3:

$\alpha = 0, r = 0, \beta = 0$
$y = 170$

Therefore, the estimated root is 17.0

There's no clever way to compute the next digit (β). Just start at zero and increment until you've found the largest digit that meets the condition. If you get to $\beta = 10$, stop looking and set $\beta = 9$. (This can happen when *r* underflows.)

Program Specification

First, the program must prompt the user to enter the number, the root to be calculated, and the number of decimal digits. The user prompts must look exactly like the text shown below. There must be exactly one space (and no linefeed) after the colon. The **printf** for the prompt does not include a linefeed; we want the user to type the value on the same line, and the user will type a linefeed as part of the input, which result in the next prompt being printed on the next line. In the example below, the numbers in bold are entered by the user, not printed by the program. The user types a linefeed (the Enter key) after each number.

```
Number: 2
Root: 2
Digits: 3
```

Next, print a restatement of the problem, in the following form:

```
Compute root 2 of 2 to 3 digits.
```

Of course, the numbers will depend on what is entered by the user. There is a period at the end of the sentence, and a linefeed.

Next, print an empty line, and then print the number of n-digit groups that are in the integer.

```
Number has 1 groups of 2 digits.
```

Note again the period and the linefeed at the end. This will give you some information about how many iterations are needed for the calculation. As described above, the number of groups plus the number of decimal digits gives the total number of iterations.

Print another empty line. For each iteration, print two lines. The first line gives the values of α and β for this iteration, and the second line gives the calculated values of y and r. This is how we check that you are following the algorithm correctly. An example output from the first example above would be:

```
alpha = 2, beta = 1
y = 1, r = 1
alpha = 0, beta = 4
y = 14, r = 4
alpha = 0, beta = 1
y = 141, r = 119
alpha = 0, beta = 4
y = 1414, r = 604
```

The number of lines will depend on the number of iterations, and of course the numbers depend on the value being computed.

Next, print an empty line and then print the estimated root as a floating-point number. Just use the default format for **printf** for a double variable; don't try to print only the number of decimal places specified by the user. The estimated root is found by dividing *y* by 10^k.

```
Estimated root = 1.414000
```

Finally, perform a calculation to show how closely the estimated root matches the actual root. Raise the estimated root to the *n*-th power[1] and print the result, followed by the original *integer* entered by the user. Again, the text in your program should exactly match what is below. Only the numbers will change. Print a linefeed at the end of this line.

```
1.414000 to the 2 = 1.999396 (orig = 2)
```

End the program by returning 0 from the **main** function.

Complete examples of the program output are shown in the Appendix.

Implementation

Do not use floating-point arithmetic during the calculation. You will find that large numbers, large roots, and a large number of decimal places may cause the integer values to exceed the size of an int variable. That is expected behavior, and not something you need to fix.

Do not use the **pow** function to raise 10 (or any integer) to a power. The **pow** function is for floating-point values, not for integers, and writing a loop is one of the learning objectives of this program. Use a loop to raise an integer to the *n*-th power.

Points will be deducted if you violate the conditions above.

You may use **pow** when performing the check operation at the end.

You are not required to define your own functions in this program, although you are allowed to do so. The code is not very long, so there is not much motivation to modularize.

Do not use any global variables. If you do, you will lose points.

Follow the (minimal) programming style guidelines posted on the Moodle page. Include comments, and include a program header that describes what the program does.

Your program does not need to check for valid input from the user. You may assume that the number will be less than 1,000,000 and that the root and digits will be reasonable.

You can, of course, use additional variables, other than the ones used in the algorithm description above. And you may use different names than the ones used above. The implementation is up to you, as long as it performs properly, passes the tests, and does not violate the program specifications.

Developing and Submitting the Program

You will submit this code as a zyBook lab, but you will develop the program outside of the zyBook. It is expected that you will use CLion, but you are free to use whatever development environment that you want.

1. In CLion, create a new project. Specify a C Executable, and use the C11 standard. Name the project whatever you like, but I recommend something meaningful like “prog1” or “roots”. This will create a directory in the place where your CLion projects are kept.
2. Now use CLion to complete the program, using the editor, compiler, and debugger to meet the program specifications.

[1] The function `pow(a,b)` returns `a` raised to the `b` power, where `a` and `b` are both **double**, and the returned value is also a **double**. You may use this function during this step only.

3. When you are ready, upload your **main.c** file to the zyBook assignment and submit. This will run the tests.

If some tests fail, go back to Step 2 and work on your program some more. The input for the failed tests will give you some idea of what's not correct, but use the debugger to figure out what's happening with your implementation.

Several iterations of Steps 2 and 3 might be necessary. In fact, it's a reasonable strategy to write a program that only passes the first test (or some tests), then improve it to pass the next test, etc. (This even has a fancy name: test-driven development.)

There is no limit to the number of times you can submit. Each submission will overwrite the earlier ones, so make sure that your new code does not break tests that passed earlier.

Hints and Suggestions

- Don't overcomplicate the program. Do not use anything that we have not covered in class. (For example, don't try to use pointers or string library functions.)
- Work incrementally. Get one part of the code working, and then move to the next. For example, you might do an initial implementation that only includes the zero function. Then you can add the linear function. Each function you add will pass more and more of the tests.
- For compiler errors, look at the source code statement mentioned in the error. Try to figure out what the error is telling you. Try to fix the first error first, and then recompile. Sometimes, fixing the first error will make all the other errors go away. (Because the compiler got confused after the first error.)
- Use a source-level debugger to step through the program if the program behavior is not correct. If you are using CLion on your own computer, the debugger is integrated with the editor and compiler, so there's no excuse for not using it.
- For general questions or clarifications, use Piazza, so that other students can see your question and the answer. For code-specific questions, post a private message to Piazza and attach your code as a file. (Do not copy and paste!)

Administrative Info

Updates or clarifications on Piazza:

Any corrections or clarifications to this program spec will be posted on Piazza. It is important that you read these postings, so that your program will match the updated specification.

What to turn in:

- Submit your **main.c** file to the zyLab assignment in *15.1 Program 1: roots*.

Grading criteria:

20 points: Program compiles. As long as you've made some effort to actually solve the problem and the file compiles with no errors or warnings, you will get these points.

10 points: Proper coding style, comments, and headers. No global variables. No goto. See the Programming Assignments section on Moodle for more style guidelines. (You will not get these points if you only submit trivial code.)

20 points: Prompt, retrieve data, and print the problem statement.

15 points: Print the number of *n*-digit groups required for the algorithm.

25 points: Print the correct values for each iteration.

10 points: Print the correct estimated root, and the correct check calculation.

NOTE: The ZyBook tests will only total 70 points. The first 30 points will be assigned manually by the grader.

NOTE: Points may be deducted for errors, even if all of the zyBook tests pass. This will be rare, but it may happen if it is obvious to the grader that the program is written specifically to pass only these tests, and would not pass other similar tests.

For example: If you know what the estimated root should be for any test, and you include hard-coded values in your program that uses that information, you will have points deducted.

Do not make assumptions. Write your code to pass <u>any</u> test consistent with this specification. We reserve the right to run your code on tests that were not provided ahead of time.

Appendix: Sample Runs

In the following examples, text in bold is entered by the user (or test program).

Example 1:

```
Number: 2
Root: 2
Digits: 3
Compute root 2 of 2 to 3 digits.

Number has 1 groups of 2 digits.

alpha = 2, beta = 1
y = 1, r = 1
alpha = 0, beta = 4
y = 14, r = 4
alpha = 0, beta = 1
y = 141, r = 119
alpha = 0, beta = 4
y = 1414, r = 604

Estimated root = 1.414000
Check: 1.414000 to the 2 = 1.999396 (orig = 2)
```

Example 2:

```
Number: 4913
Root: 3
Digits: 1
Compute root 3 of 4913 to 1 digits.
```

```
Number has 2 groups of 3 digits.

alpha = 4, beta = 1
y = 1, r = 3
alpha = 913, beta = 7
y = 17, r = 0
alpha = 0, beta = 0
y = 170, r = 0

Estimated root = 17.000000
Check: 17.000000 to the 3 = 4913.000000 (orig = 4913)
```

Example 3:

```
Number: 31
Root: 4
Digits: 4
Compute root 4 of 31 to 4 digits.

Number has 1 groups of 4 digits.

alpha = 31, beta = 2
y = 2, r = 15
alpha = 0, beta = 3
y = 23, r = 30159
alpha = 0, beta = 5
y = 235, r = 50199375
alpha = 0, beta = 0
y = 2350, r = 0
alpha = 0, beta = 0
y = 23500, r = 0

Estimated root = 2.350000
Check: 2.350000 to the 4 = 30.498006 (orig = 31)
```

(In this case, *r* becomes negative in step 4 and is therefore set to zero.)