

COMP 321: Introduction to Computer Systems

Project 1: Factors

Assigned: 1/10/19, **Due: 1/17/19, 11:55 PM**

Important: This project must be done individually. Be sure to carefully read the course policies for assignments (including the honor code policy and the slip day policy) on the assignments page of the course web site:

<http://www.clear.rice.edu/comp321/html/assignments.html>

Overview

You will write C code to count the prime factors of a number. The goals of this project are as follows:

- Write a small amount of C code.
- Become familiar with compiling C code.
- Understand the perils of writing a deeply recursive program.

Summary

You will write three versions of code to count the prime factors of a positive integer greater than one¹:

- Count all of the prime factors recursively. For example, 12 has 3 prime factors: 2, 2, and 3.
- Count all of the prime factors iteratively.
- Count the distinct prime factors iteratively. For example, 12 has 2 distinct prime factors: 2 and 3.

You will write a different procedure to handle each version. You do not need to write a particularly efficient program, although it should complete in sixty seconds on any input. In particular, a truly efficient version would use some data structure to store previously discovered primes, but you have not yet learned about data structures in C.

You will not write an entire program. We provide a basic `main` function to handle the I/O. First, you will write a recursive solution, but this solution might not work for certain large inputs, for example, 2,000,000,001. Second, you will write an iterative solution that is guaranteed to work for all valid inputs.

¹Please remember, "1" is *NOT* a prime number.

Deeply Recursive Programs

When a function is called from a C program, it is allocated memory to store its parameters and local variables. This allocation typically occurs on a region called the *program stack*. You will learn more about program stacks later in the semester.

A recursive function is one that calls itself repeatedly until some condition is satisfied. Depending on how deep the recursion is, the program stack may consume a lot of memory. Eventually the program will run out of memory and crash. For example, a very simple recursive solution to count all prime factors is likely to crash with certain large inputs, for example, 2,000,000,001. You should learn to write programs which efficiently use their stack. This is especially important if the program will be executed on a platform with limited memory, such as embedded systems or mobile phones. **In this assignment, you have to write both a recursive solution (that does not have to work on all inputs) and an iterative solution that uses the stack efficiently and works for all valid inputs.** A valid input is any number that can be represented by an unsigned `int` on `CLEAR`.

The `limit` command can be used to check and change the maximum stack size which can be allocated by your program. We encourage you to read the related man pages to understand how to use this command. The default maximum stack size per program is 10 MB on `CLEAR` and your solution will be graded using only that default size.

Notes

- The compiler generates warnings for a reason. You should fix any code that generates a warning to minimize the chances of bugs. Experienced C programmers may understand what they are doing well enough to decide that a warning is unimportant. However, code that causes warnings can be an indication of actual program errors, so it is a good idea to fix any code that causes a warning regardless of your experience level. Part of your coding style grade for this project will be determined by whether or not your code compiles without any warnings. (We will use the `-Wall` and `-Wextra` flags to `clang`.)
- Testing is critical in all programming. You need to be sure to comprehensively test the procedures that you write. Specifically, you should think about the possible types of programming (or algorithmic) mistakes that might be made, and use these possible mistakes to design your testing strategy. This will maximize your correctness score, and a description of your testing strategy is also part of the writeup score. Just because your program happens to work correctly does not mean that you tested it well. This is especially true for relatively simple programs such as this one. The projects will get more complicated, so establishing a habit of thoughtful and thorough testing now will be valuable.
- We provide a `main` function that you must not change. However, when `-t` is specified on the command line, the function `test_factors` will be called with no arguments instead of running the program normally. You may put whatever testing code you would like in this function. We will not use the `-t` argument when grading your program, so this is solely for you to use in your testing as you see fit.
- When `-r` is specified on the command line, the function `count_factors_recursive` will be called. This function should have a recursive implementation. You must not use `for` or `while` loops in its implementation. You may write helper functions, as needed, but they too must not use any loops. As previously discussed, this function does not need to handle all inputs without crashing due to stack overflow. (That said, it is fine if you are able to design an implementation that does handle all inputs without crashing).

- You may also find it helpful to use some simple `printf` statements within your procedures to see intermediate values during the execution of your program as you are testing and debugging your code. The `printf` statement in C has a format string followed by a list of zero or more additional arguments. The “%” characters in the format string specify the locations in the output where the additional arguments must be printed, and the character following each “%” must match the type of the corresponding argument to `printf`. For example, a single unsigned integer can be printed in C using `printf` as follows:

```
printf("Print an unsigned int, %u, between the commas.\n", number);
```

You can also learn from the other `printf` statements in the code given to you. Using `printf` statements is an effective method of debugging simple programs and for narrowing down errors in more complex programs. However, make sure that you comment out all such `printf` statements when you turn in your code, as the final program should only perform the input and output that is provided in the `main` function.

- 20% of your grade on this assignment will be based on your coding style, which includes commenting your code for clarity. Use `requires/effects` comments for each procedure, as in the provided code. These procedure comments should be about *what* the procedure does, not how it does it. In other words, a procedure `sum` might have the following *effects*: Returns the sum of the elements in the array. An inappropriate comment would be: loops over the elements in the array using a for loop, adds each element to a running sum, and then returns the running sum after the for loop completes. Comments inside the procedure can be used to document what the procedure is doing when the code is not relatively obvious. However, commenting the statement `i += 1;` with “Add one to i.” is pointless. It is only adding clutter to your program.
- When you are modifying an existing program, *the first and foremost rule of good coding style* is that the style of your code, *e.g.*, its indentation, should match the style of the surrounding code. In effect, it should not be obvious to a reader where different people have edited the code. Also, keep in mind that if you consistently indent your code, it will make it easier for someone else (*i.e.*, the graders) to understand your code. A comprehensive description of our required C coding style can be found on the course web site at:

<http://www.clear.rice.edu/comp321/html/style.html>

Getting Started

To get started on this assignment, please visit the web page at <https://classroom.github.com/a/X3T6vWJx>. (If you are copying this URL, do not include the period at the end of the sentence, as it is not part of the URL.) This page should be titled “RICE-COMP321-SPRING19”. Moreover, it should say, “Accept the Factors assignment”, and have a button labeled “Accept the invitation”². Please accept the invitation.

Upon accepting the invitation, you will be redirected to another web page. This page confirms that you have accepted the assignment and provides a link to your personal repository for the assignment. Click the link to go to your personal repository.

The web page for your personal repository has a button labeled “Clone or download”. Click this button. You should now see a text field with a URL. Copy or remember this URL.

²You may have to login to `github` to see the invitation. If so, you will be prompted for your username and password at the top of the page.

Login to the CLEAR system if you have not already done so. Type the following:

```
git clone [Copy the URL for your repo here]
```

You will be prompted for your github username and password.

Once the clone operation is complete, you will have a directory named

```
factors-[YOUR github ID]
```

Please `cd` into this directory, and run the command `ls`. You should see the following files:

- `factors.c` – provided code
- `Makefile` – specification for building `factors` using `make`
- `writeup.txt` – a skeleton writeup file for you to complete

If you do *NOT* see these files, contact the course staff immediately!

Compiling and Running Your Code

To build the program, use the Unix command:

```
UNIX% make
```

Note that the “UNIX%” above represents the prompt from your shell. You should only type into the shell the part after the prompt (in this case, “make”).

This will compile your code and build the `factors` program. The `factors` program could then be run to determine the number of factors of 12 as follows:

```
UNIX% ./factors
Enter number:
12
12 has 3 prime factors, 2 of them distinct.
```

Make sure that the iterative version of your program works for all valid inputs. However, the recursive version of your program may crash with large inputs. For example, on executing the recursive solution to count all prime factors with input 2,000,000,001, the following will likely occur:

```
UNIX% ./factors -r
Enter number:
2000000001
Segmentation fault (core dumped)
```

Your program must only handle valid inputs. If an invalid input is given to your program, it is acceptable for your program to output nonsense. In addition, it is acceptable for the recursive version of your program to crash on an invalid input.

Be sure to document your testing strategy in your writeup. Specifically, you should list your test cases, explain how you selected test cases, and argue that your test cases are sufficient. Your sufficiency argument will be a non-trivial portion of your writeup score. A good sufficiency argument makes a convincing case as to why the potential mistakes you identified are the only possible ones.

Turning in Your Assignment

To turn in your assignment, you *must* use `git push` to copy your work to the `github` remote repository. We will *only* look at the last version that you pushed before the deadline. As a precaution against accidental loss of your code or writeup, we encourage you to push periodically. Please note, the *only* files that you need to turn in are `factors.c` and `writeup.txt`. In other words, these are the only two files on which you should ever perform `git add`. For grading your submission, we will use the `Makefile` that was originally provided to compile your code. Therefore, your code should not rely on any modifications to the `Makefile` for correct compilation.

As a sanity check, you should use your web browser to visit your assignment repo. Make sure that what you see in the browser is consistent what you think you have pushed.

Grading

This project will be graded as follows:

- Writeup: 20%
- Coding style: 20%
- Correctness: 60%