

-----

Please read this assignment carefully and follow the instructions EXACTLY.

### Submission

-----

Please refer to the lab retrieval and submission instruction, which outlines the only way to submit your lab assignments. Please do not email me your code.

If a lab assignment consists of multiple parts, your code for each part must be in a subdirectory (named "part1", "part2", etc.)

Please include README.txt in the top level directory. At a minimum, README.txt should contain the following info:

- your name
- your UNI
- lab assignment number
- description of your solution

The description can be as short as "My code works exactly as specified in the lab." You may also want to include anything else you would like to communicate to the grader such as extra functionalities you implemented or how you tried to fix your non-working code.

### Makefile

-----

If a part asks for a program, you must provide a Makefile. The TAs will make no attempt to compile your program other than typing "make".

The sample Makefile in the lecture note is a good starting point for the Makefile you need to write for this lab. If you understand everything about the sample Makefile, you should have no problem writing this lab's Makefile.

Here are some additional documentations on Make:

- Stanford CS Education Library has a nice short tutorial on Makefiles. See Unix Programming Tools, Section 2, available at <http://cslibrary.stanford.edu/107/>.
- The manual for make is available at <http://www.gnu.org/software/make/manual/make.html>. Reading the first couple of chapters should be sufficient for basic understanding.

There are a few rules that we will follow when writing makefiles in this class:

- Always provide a "clean" target to remove the intermediate and executable files.

- Compile with gcc rather than cc.
- Use "-Wall" option for all compilations.
- Use "-g" option for compiling and linking.

The sample Makefile follows all these rules.

Part 0 (10 points)

-----

Answer the following questions in your README.txt.

- [0.1] What git command should never be used after cloning the lab skeleton code?
- [0.2] Which of the following commands should you run in order to start a lab in this class?
- (A) git init
  - (B) git push
  - (C) git clone
  - (D) All of the above

For [0.3] and [0.4], let's assume that you have made some changes to hello.c, but regret it, and want to go back to the version last committed.

- [0.3] What command do you use if the file has not yet been staged?
- [0.4] What command do you use if the file has been staged?

Part 1 (50 points)

-----

Write a C program that reads 2 positive integers from the user using scanf() function, prints the following information, and then terminates:

- the average of the two (this should be printed as a floating point number.)
- whether each number is a prime number or not
- whether the two numbers are relatively prime or not (see <http://en.wikipedia.org/wiki/Coprime> if you don't know what this means.)

You can assume that the user will input only positive integers, i.e., don't do any error checking.

In order to see if two numbers are relatively prime, you should calculate the GCD using Euclidean algorithm. You are allowed to look up the algorithm and/or code on the Internet, in which case you should cite the source in your README.txt file.

Your code should be organized as follows:

- gcd.h & gcd.c: GCD calculation function header and definition
- prime.h & prime.c: prime number testing function header and definition
- main.c: everything else
- Makefile

All files must be named EXACTLY as above, case-sensitive. When you run "make", it should build an executable file called "main".

The makefile should have correct dependencies. For example, if you build everything, change prime.h, and run make again, only prime.c and main.c should recompile, not gcd.c. (You can simulate changing a file by using 'touch' command.)

You can use

```
printf("You typed in %d and %d\n", x, y);
```

to print integers, and

```
printf("The average is: %f\n", avg);
```

to print a floating point number.

And you can use

```
scanf("%d", &x);
```

to read an integer that the user types in. Don't forget the ampersand in front of the variable.

Part 2 (40 points)

-----

Write a C program called "convert" that reads a signed decimal integer from the user and prints the number in 4 different ways: signed decimal, unsigned decimal, hexadecimal, and binary.

Here are a few example runs of the program:

```
$ ./convert
-1
signed dec:   -1
unsigned dec: 4294967295
hex:         ffffffff
binary:      1111 1111 1111 1111 1111 1111 1111 1111

$ ./convert
256
signed dec:   256
unsigned dec: 256
hex:         100
binary:      0000 0000 0000 0000 0000 0001 0000 0000
```

There is a bash shell trick that you might find useful for testing your program. You can evaluate an arithmetic expression like this:

```
$ echo $(( 1 + 2 + 3 ))  
6
```

In addition, you can take the output of one program, and feed it as a user input to another program by chaining the two programs with '|' character. So, you can input the minimum 32-bit integer like this:

```
$ echo $(( -2 * 1024 * 1024 * 1024 )) | ./convert  
signed dec:  -2147483648  
unsigned dec: 2147483648  
hex:        80000000  
binary:     1000 0000 0000 0000 0000 0000 0000 0000
```

Note that your program must behave EXACTLY the same as the sample runs shown above. Given the same number, your program must generate the EXACT same output--same order, same strings, same spacing.

To help you get started, I gave you a little C program called printf-test.c in the lab 1 skeleton code. Please take a look at it. Good luck!