COMS W3157 Advanced Programming, Lab #1 (100 points total)

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.

All lab submissions must include at least five git commits with meaningful comments.

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 must contain the following info:

- your name
- your UNI
- lab assignment number
- description of your solution
- answers to academic honesty questions (see below)

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.

Academic Honesty

To ensure an effective and fair learning environment, it is essential for every student to know what constitutes academic dishonesty and uphold the class/university policy. The following questions are intended to help each student reinforce a thorough understanding of that policy. These scenarios are inspired by actual academic dishonesty cases that have been encountered in the past. Pleaser refer to Prof Jae Woo Lee's Academic Honesty Policy and the CS Department's Academic Honesty Policy to help answer these questions.

For each of the following scenarios, write YES or NO as to whether they constitute cheating and justify your response.

[1] You hire a tutor to assist you with labs. You find that your solution has lots of errors and you feel overwhelmed and unsure where to begin. The tutor addresses this by asking leading questions and pointing you to areas in the code that may have caused these errors.

[2] You're struggling to fix an error and know that a friend enrolled in the class has finished the lab. You reach out to explain your error and they send you a screenshot of the relevant portion of their code. You look at it to understand your error but do not copy any lines.

[3] At the beginning of the semester you are concerned that you might not be able to complete all the labs on your own. For peace of mind you ask a friend from a previous semester for all solutions. You wind up not needing them and never consult them once.

[4] After lecture, you're eating dinner with a friend and discussing that week's lab assignment. Determine whether each of the following scenarios is cheating:

  [a] Your friend finds a concept from lecture unclear and asks you to explain.
  [b] Your friend asks you to explain a small conceptual element of their solution.
  [c] Your friend asks you what you think the lab assignment is trying to teach you.

[5] You are stuck on a lab in which you have to build a server. You know that the lab specifications want you to figure out how to take on multiple requests, but you're not sure exactly how to order your code so that it works. After a few Google searches, you read a post that lays out some code that would be a solution to the lab. You make sure to not use any of the code in your submission.

[6] You are working on the lab. You try to compile your code, but you get an error. You paste the error into Google and find a page that explains the nature of the error. You understand the error, go back to your code and apply what you have learned to fix the error.

[7] You're working on the lab but having difficulty applying concepts learned in class. You search for a lab solution from a previous semester, but only copy in small parts at a time to test the code, look up man pages for functions used and try to fully understand the solution.

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 notes 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

Answer the following questions in your README.txt.

[0.1] What git command(s) 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

[0.3] Assume that you made some changes to hello.c. The changes has not been staged yet. What command do you use to discard your change?

[0.4] Assume that you made some changes to hello.c, and staged the change. What command do you use to unstage your change?

Part 1

Write a C program that reads 2 positive integers (of type int) 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 should 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

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 indentation. Make sure the numbers start at the same column and there is a space between every four binary digits as shown above.

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!