TEST 1 STUDY GUIDE

MAT 685, C++ FOR MATHEMATICIANS

The test will cover **through Chapter 4**.

COMPUTER SCIENCE

General.

- Purpose of programming
- Kinds of programming languages
  - compiled v. interpreter v. bytecode
  - strong v. weak typing
  - which category(-ies) C++ falls under
  - versions of C++
  - why C++
- Compiling and linking
  - which compiler we're using and why
  - source files
    - header
    - implementation
  - object files
    - library
    - executable
  - compiler flags
  - basic compiler errors and how to fix them
- Debugging: tool and basic commands
- Monte Carlo v. Las Vegas algorithms

Basic structure of a C++ program.

- preprocessor directives
  - how/why *we* use them
  - `#define`, `#ifndef`, `#endif`
- include files
  - how to create (requirements as well as best practices)
  - how to use
  - useful/necessary libraries
    cstdlib: I/O
    cmath: math
    cstdlib: system services
    ctime: timer
  - using an object from a namespace
- comments

Date: Spring 2017.

- procedures
  - difference from mathematical function
  - arguments
    - call-by-value
    - call-by-reference
  - returning values
    - how to "return" more than one
  - local v. global scope
  - overloading: when it's allowed, when it's not
  - recursion
- loops
  - indefinite v. definite (concept, not keywords)
  - `while` v. `for`
    - why `for` is really `while`
- decisions (`if...else`)
- data, limited to machine types for now
  - boolean, character types
    - character v. string
  - numeric types
    - integer types (`short`, `int`, `long`, `long long`)
    - floating-point types (`float`, `double`)
  - `const`
  - standard operations on types
  - pointer (basic definition, `nullptr`)
  - `static` variables

MATHEMATICS

Greatest Common Divisor.

- relatively prime numbers
- Euclidean algorithm
- Bézout's identity
- extended Euclidean algorithm

sequences.

- Fibonacci sequence (recursive and non-recursive computation)
- Lucas sequence (generalized Fibonacci sequence)

pseudo-random numbers.

- difference between random, pseudo-random
- linear congruential generator
- Box-Muller method

TAKE-HOME PROGRAMMING ASSIGNMENT

Another algorithm to compute the gcd of two numbers is the **binary method**, which works especially well on computers.

given $a, b \in \mathbb{N}$
output $\gcd(a, b)$
do

- let $d = 0$
- **while** $a \neq b$
  - **if** a, b both even
    - divide a, b by two
    - increment d
  - **else if** a even, b odd
    - divide a by two: why does this not change the gcd?
  - **else if** a odd, b even
    - divide b by two: why does this not change the gcd?
  - **else**
    - suppose $a > b$ (if not, reverse the two in what follows)
    - let $c = a - b$: $\gcd(a, b) = \gcd(b, c)$ — why?
    - replace c by $c/2$: $\gcd(a, b) = \gcd(b, c/2)$ — why?
    - replace a by c
- **return** $a \times 2^d$

Questions.

1. Use the binary algorithm to compute the gcd of several numbers by hand. Try it at least with the pairs you had to do from the book. Also choose some other pairs to compute the gcd. You want to make sure you understand the algorithm very, very well before moving on.
2. Implement the binary algorithm in C++. You already have a header file, so you only need an implementation file; call it `gcd_binary.cpp`. Save it in the same directory as your other gcd implementations. Be sure to include at least some comments.
3. Compile and, if necessary, debug the program until it works correctly.
4. Implement the following optimizations that are possible only because everything involves division and divisibility by 2. Suppose that `n` is an integer type, then:
   - You can test for divisibility by 2 by replacing `n % 2 != 0` by `n & 1`. (This is a bit operation.)
   - You can multiply by 2 by replacing `n * 2` by `n << 1`. (This is a bitwise "shift" left.)
   - You can divide by 2 by replacing `n / 2` by `n >> 1`. (This is a bitwise "shift" right.)
   - You can multiply or divide by the dth power of 2 using `n << d` to multiply and `n >> d` to divide.

   Check to make sure your program still produces the same output. Which version do you think is easier to read and maintain: the first or the second? (Both optimizations make sense if you understand think about the binary representation of the numbers. You do not need to worry too much about this at this time.)
5. *(Extra credit)* Can you extend the binary algorithm to find Bézout's Identity? If so, implement it. If not, why not?