Homework Assignment 9

Due by 5:30 pm on Friday, December 8, 2017.

1: Introduction

The goals of this problem set are:

(a) To learn how to use the GNU multiple precision arithmetic library in cryptographic applications.

(b) To produce working code for an incremental version of the Blum-Blum-Shub cryptographically secure pseudorandom sequence generator presented in lecture 22, slide 8.

2: The Assignment

The problem set consists of two problems.

(a) Everyone *must* do the first problem in order to receive full credit for the assignment.

(b) Graduate students (and others registered under CPSC 567) *must* do both problems for full credit. Undergraduates *may* do problem 2 for a maximum of 50 points extra credit.

Problem 1: Implementation of BBS generator

Implement the BBS pseudorandom sequence generator in C++ using the Gnu MultiPrecision (GMP) library. I have furnished a C++ header file `bbs.hpp` on the Zoo in the folder `/c/cs467/assignments/hw9/`. Your job is to create an implementation file `bbs.cpp` for the functions declared there.

The header file `bbs.hpp` describes an incremental version of BBS. An object of class BSS contains two `mpz_class` data members: A modulus `n`, which must be a Blum integer, and a `state`, which is the current state of the generator. The BBS constructor initializes `n` and `state`. Private member function `BBS::lsb()` returns the least significant bit of `state`. Public member function `BBS::srand()` is a setter function for `state`, allowing it to be changed at any time. Public member function `BBS::rand()` generates the required number of output bits, which are packed together into a big integer of type `mpz_class` and then returned. For each bit generated, the state is updated *before* the least significant bit is extracted from it.

You may find it helpful to use the `mpz_tstbit()` function in your implementation. Remember also that the familiar C++ operators `*, %, =, <<, |` work for numbers of type `mpz_class` as well as for ordinary `int`s.

Problem 2: Implementation of function to find Blum integers

[This problem is required for graduate students and others taking the course under the CPSC 567 number. It is optional and extra credit for CPSC 467 students.]

Implement functions to generate random Blum primes and random Blum integers of a specified size (number of bits). You will find a header file `Blum.hpp` in the zoo folder `/c/cs467/assignments/hw9/`. Your job is to create an implementation file `Blum.cpp` for the functions declared there.

To generate a random Blum prime, you should use the GMP random number generator `gmp_randclass`. It contains a member function `get_z_bits()` which returns a random `mpz_class` integer of the specified length. You can then use `mpz_probab_prime()` with an iteration count of 20 trials in order to determine if a random candidate for being a Blum prime really is a prime (with high likelihood).

Once you know how to generate a random Blum prime of a given length, it is an easy matter to generate a random Blum integer of a desired length.

You should also write a driver command `genBlum seed size num` that calls your function `randBlumInt()` to generate `num`-many `size`-bit random Blum integers n. The generator should be seeded using the argument `seed`.

Your driver program should write `num` lines of text to standard output. Each line should consist of three space-delimited hex numbers `p q n`, where `p` and `q` are the Blum primes used to produce the Blum integer `n` returned by `randBlumInt()`.

In order to do this, you may modify the furnished header file `Blum.hpp` by adding data members `p` and `q` and member functions `void showp()` and `void showq()` that will print `p` and `q` to standard output, respectively.

3: GNU Multiple Precision Arithmetic (GMP)

GMP is a highly optimized package for calculating with big numbers. It was originally designed for use with C . More recently, C++ extensions have been added that make it much easier to use for ordinary calculations, but some of the more sophisticated functions are still only available through the C interface. See the GMP manual at `http://gmplib.org/manual/index.html` for documentation. Since you will be using the C++ interface, you should pay special attention to section 12 of the GMP manual. You can also read the manual on the Zoo by typing `info gmp`.

Please feel free to ask for help with C++. You might find the C++ tutorial at `http://www.learncpp.com` helpful. The site `http://www.cplusplus.com` has a wealth of information useful to the C++ programmer. Finally, look at the code that I furnished for `BBSTest` and `Tester`. While you don't need to understand the details of my code in order to use it, it does give you several examples of how to use GMP big integers.

4: Testing Your Code

Random number generators are notoriously difficult to test and debug because, well, the correct answers look random. 

I am furnishing a test program that you can use to test that your solution is correct. My program consists of three files: `BBSTest.cpp`, `Tester.hpp`, and `Tester.cpp`. These files should be compiled and linked with `bbs.o` (from problem 1) and `Blum.o` (from problem 2) to produce the executable command `tester`.

`tester` provides a command interface to the BBS generator and the Blum integer finder. Its commands allow for exercising and testing the code you are to write. The furnished makefiles simply copy and link to my `.o` files. When you solve problem 1, you should create a file `bbs.cpp` and modify the makefile to compile and use your code instead of mine. Comments inside the makefile tell you want to do. Similar remarks apply to `Blum.o` if you do problem 2.

The code involves two different random number generators, which can get a bit confusing. First, you will be using the GMP random number generator `gmp_randclass`, initialized with `gmp_randinit_default`. Like familiar random number generators, it must be seeded before it can be used. `tester` takes the seed for this generator as an optional command line argument. Specifying the seed makes sure that each run of your code is reproducible, which can be a great aid in debugging. If the argument is omitted, `tester` seeds the generator with the current clock time. (This is insecure, but it's useful for exploring the behavior of your code.)

The other random number generator is BBS, which is described in problem 1.

The command processor `tester` takes five subcommands:

- `genBI size`: Generates a random `size`-bit Blum integer.
- `readBI file`: Reads a Blum integer from `file`.
- `srand [seed]`: Creates and seeds a BBS generator using the specified seed and the current Blum integer set by the most recent `genBI` or `readBI` subcommand. The current clock time is used if the `seed` argument is omitted.
- `rand num`: Generates `num` bits using the current BBS generator.
- `quit`: Exit program.

To use `tester` to run the BBS generator, you must first obtain a Blum integer and set the modulus to it. You can either use `genBI` to generate a new random Blum integer, or you can use `readBI` to read a Blum integer from a file. I have furnished data files `bi20`, `bi40`, `bi80`, and `bi160`. Each contains a single Blum integer of length 20, 40, 80, and 160 bits, respectively.

Once the modulus has been set, you should call the `srand` command to set the BBS seed[1].

You can now use the `rand` command to produce output from the BBS generator.

5: How to Use My Files

The Zoo directory `/c/cs467/assignments/hw9/` contains several parts:

(a) Makefiles for both linux and osx to build an executable `tester` to be used to exercise the BBS generator and the optional random Blum integer generator.

(b) Data files `bi20`, `bi40`, `bi80`, and `bi160`. described above.

(c) Header files `bbs.hpp` and `Blum.hpp` for problems 1 and 2.

(d) Code files `BBSTest.cpp`, `Tester.hpp`, `Tester.cpp` for the `tester` program.

(e) Object (`.o`) files for my implementations of `bss.cpp` and `Blum.cpp`. I have compiled them for both linux and osx and have placed them in their respective subdirectories.

[1] Note that this is not the same as the seed given to `tester`, which is the seed given to the GMP random number generator.

(f)

Begin by copying the Zoo directory `/c/cs467/assignments/hw9` to your home directory. Copy the `Makefile.`*`system`* file appropriate for your *system* to `Makefile`. Type `make`, and make sure it builds the executable `tester`. You will need the GMP libraries and development packages installed on your system for this to work. They're already preinstalled on the Zoo for your convenience. If you choose to work on another machine, it's up to you to configure it properly.

6: Submission

Submit all code and makefiles necessary to build and run your programs to the Canvas course site. After downloading your files, the grader should be able to type

```
> make tester
```

in order to generate a version of `tester` that runs your BBS implementation and, if you are solving problem 3 as well, your Blum integer implementation.

Good luck, and don't be afraid to ask for help!