PS2: Linear Feedback Shift Register (part A)

In this assignment you will write a program that produces pseudo-random bits by simulating a linear feedback shift register, and then use it to implement a simple form of encryption for digital pictures.

For this portion of the assignment, you will:

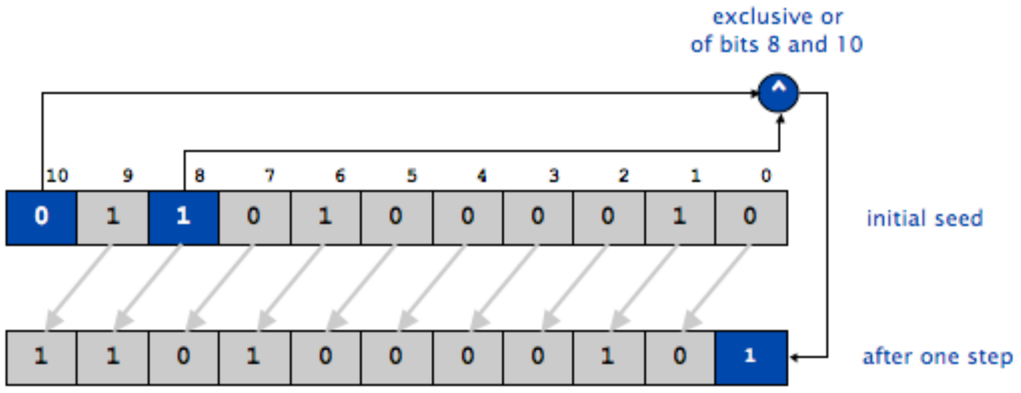
- implement the LFSR class
- implement unit tests using the Boost test framework

What is an LFSR?

A *linear feedback shift register* (LFSR) is a register that takes a linear function of a previous state as an input. Most commonly, this function is a Boolean exclusive OR (XOR). LFSR performs discrete step operations that

- Shifts the bits one position to the left, and
- Replaces the vacated bit by the *exclusive or* of the bit shifted off and the bit previously at a given *tap* position in the register.

A LFSR has three parameters that characterize the sequence of bits it produces: the number of bits *N*, the initial *seed* (the sequence of bits that initializes the register), and the tap position *tap*. The following illustrates one step of an 11-bit LFSR with initial seed 01101000010 and tap position 8.



One step of an 11-bit LFSR with initial seed 01101000010 and tap at position 8

Note: the position 0 is at the right of the diagram.

LFSR Data Type

Your first task is to write a data type that simulates the operation of a LFSR by implementing the following API:

```
class LFSR {
public:
     LFSR(string seed, int t);  // constructor to create LFSR with
                                // the given initial seed and tap
     int step();                // simulate one step and return the
                                // new bit as 0 or 1
     int generate(int k);       // simulate k steps and return
                                // k-bit integer
private: ...
}
```

To do so, you need to choose the internal representation (data members), implement the constructor, and implement the three member functions. These are interrelated activities and there are several viable approaches.

- *Constructor.* The constructor takes the initial seed as a `String` argument whose characters are a sequence of `0`s and `1`s. The length of the register is the length of the seed. We will generate each new bit by *XOr*ing the leftmost bit and the tap bit. The position of the tap bit comes from the constructor argument. For example, the following code should create the LFSR described above.

  ```
  LFSR lfsr("01101000010", 8);
  ```

- *Destructor*. If your constructor dynamically allocates memory, make sure to define a destructor that deallocates it.
- *String representation.* Overload the `<<` stream insertion operator to display its current register value in printable form (see these instructions http://www.learncpp.com/cpp-tutorial/93-overloading-the-io-operators/)
- *Simulate one step.* The `step()` function simulates one step of the LFSR and returns the rightmost bit as an integer (0 or 1). For example, if you call step() 10 times the output should be:

  ```
  11010000101 1
  10100001011 1
  01000010110 0
  10000101100 0
  00001011001 1
  00010110010 0
  00101100100 0
  01011001001 1
  10110010010 0
  01100100100 0
  ```

- *Extracting multiple bits.* The member function `generate()` takes an integer `k` as an argument and returns a `k`-bit integer obtained by simulating `k` steps of the LFSR. This task is easy to accomplish with a little arithmetic: initialize a variable to zero and, for each bit extracted, double the variable and add the bit returned by `step()`. For example, if the first 5 bits extracted are `1`, then `1`, then `0`, then `0`, then `1`, the variable takes on the values 1, 3, 6, 12, and 25, which is the binary representation of the bit sequence `11001`. For example, call `generate(5)` should output:

  ```
  00001011001 25
  01100100100 4
  10010011110 30
  01111011011 27
  01101110010 18
  11001011010 26
  01101011100 28
  01110011000 24
  01100010111 23
  01011111101 29
  ```

  Implement the `generate()` function by calling the `step()` function `k` times and performing the necessary arithmetic.

- *Testing.* Implement unit tests using the Boost test framework.

  - See http://www.boost.org/doc/libs/1_53_0/libs/test/doc/html/utf/tutorials.html for an introduction to using Boost unit testing.

    Additional info:

    https://www.ibm.com/developerworks/aix/library/au-ctools1_boost/index.html
    https://theboostcpplibraries.com/

    If you're working on Mac, take a look at the above link (because it introduces how to use Boost). Then, follow these installation instructions:

    https://www.boost.org/doc/libs/1_50_0/doc/html/quickbook/install.html#quickbook.install.macosx

What to turn in

- The implementation must be contained in files named `LFSR.cpp` and `LFSR.hpp`.

  Note: Your code must work with seed strings up to 32 bits long.

- Two additional unit tests in Boost, in a file `test.cpp` (start with the existing file `~/COMP2040/PS2a/test.cpp` in our Ubuntu VM).

  You must have *two more sets of tests* in additional BOOST_AUTO_TEST_CASE blocks. Each block should be commented with a short description of the test. Try to test some edge cases of your implementation (e.g. very long or short seed strings).
  ☐

- Create a Makefile to build your project. You must compile LFSR.cpp and test.cpp, and link them together with the boost_unit_test_framework library into an executable named ps2a.

  Your Makefile should have the targets `all`, `LFSR.o`, `test.o`, `ps2a`, and `clean`, and make sure all prerequisites are correct (e.g., `LFSR.o` should have `LFSR.cpp` and `LFSR.hpp` as prerequisites).

- Create a `ps2a-readme.txt` file that includes:

  1. your name;
  2. an explanation of the representation you used for the register bits (how it works, and why you selected it); and
  3. a discussion of what's being tested in your two additional Boost unit tests.

- Make sure all your files are in a directory named `ps2a`
- If you additionally have a `main.cpp` file with some printf-style tests, you may include that too.

How to turn it in

- Submit a "tarball" (e.g., a tar archive file compressed using gzip) via the PS2a assignment page on Blackboard. The name of your tarball file should include your name; e.g., `Tom_Wilkes_ps2a.tar.gz` .

Grading rubric

Feature	**Value**	**Comment**
Implementation	4	full & correct implementation
Makefile	2	full & correct implementation
your own `test.cpp`	2	all files packaged in .tar.gz file with correct directory structure
ps2a-readme.txt	2	complete and discusses work
Total	**10**	