PS1: 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 FibLFSR class
- implement unit tests using the Boost test framework

Average time to complete assignment ~5 hours.

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 bits previously at a given ***taps*** position in the register.

In general, 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 diagram illustrates the 16-bits *Fibonacci LFSR* with initial seed 1011011000110110 and tap positions 10, 12, and 13.

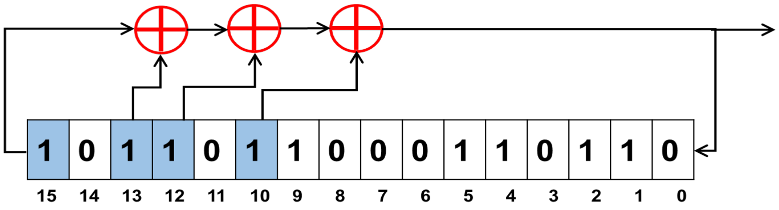

Diagram 1

Fibonacci LFSR Data Type

Your first task is to write a data type that simulates the operation of a 16 bits Fibonacci LFSR by implementing the following API:

```
class FibLFSR {
public:
      FibLFSR(string seed);   // constructor to create LFSR with
                              // the given initial seed
      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: ...
}
```

This assignment is based on Princeton CS assignment created by Robert Sedgewick.

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 XORing the leftmost bit and the tap bit, when using resulting bit as one of the inputs for the next XOR gate, and so on. There are 3 tabs for this assignment in positions 13, 12, and 10 (see diagram 1). For example, the following code should create the FibLFSR described above.

```
FibLFSR flfsr("1011011000110110");
```

- ***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 and print the content of the register and the output of `step()` function and you should see:

```
0110110001101100  0
1101100011011000  0
1011000110110000  0
0110001101100001  1
1100011011000011  1
1000110110000110  0
0001101100001100  0
0011011000011001  1
0110110000110011  1
1101100001100110  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 `0`, then `0`, then `0`, then `1`, then `1`, the variable takes on the values 0, 0, 0, 1, and 3, which is the binary representation of the bit sequence `00011`. For example, call `generate(5)` should return:

```
1100011011000011 3
1101100001100110 6
0000110011001110 14
1001100111011000 24
0011101100000001 1
0110000000101101 13
0000010110111100 28
```

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

This assignment is based on Princeton CS assignment created by Robert Sedgewick.

- ***Testing***. Implement unit tests using the Boost test framework.
  - Install Boost into your development environment. From the shell:

    ```
    sudo apt-get install libboost-test-dev
    ```

  - 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/

What to turn in

- The implementation must be contained in files named `FibLFSR.cpp` and `FibLFSR.h`.
- Two additional unit tests in Boost, in a file `test.cpp` (attached).
- You must have *two more sets of tests* in additional BOOST_AUTO_TEST_CASE blocks. ach block should be commented with a short discussion.
- Create a `Makefile` to build your project. You must compile `FibLFSR.cpp` and `test.cpp`, and link them together with the boost_unit_test_framework library into an executable named ps1a.
- Your Makefile should have the targets `all`, `FibLFSR.o`, `test.o`, `ps1a`, and `clean`, and make sure all prerequisites are correct (e.g., `LFSR.o` should have `LFSR.cpp` and `LFSR.h` as prerequisites).
- Submit a `ps1a-readme.txt` file that includes:
  1. name,
  2. an explanation of the representation you used for the register bits (how it works, and why you selected it)
  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 `ps1a`
- If you additionally have a `main.cpp` file with some printf-style tests, you may include that too.

How to turn it in

Submit on Blackboard.

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
ps1a-readme.txt	2	complete and discusses work
Total	**10**	

This assignment is based on Princeton CS assignment created by Robert Sedgewick.