Lab 5a – XOR Cipher and LFSR

Learning Goals

1) Develop your ability to create object like functionality in C using a combination of structs, typedef, and pointers.
2) Develop your ability to use malloc and free to allocate/deallocate memory on the heap.
3) Develop your skills with arrays.

Background – XOR Cipher

A common form of cryptography is an XOR cipher, which takes advantage of the fact that if for any three bytes A, B and C, A ^ B = C implies C ^ B = A.

This also holds for arbitrarily long byte-strings: by using a bitwise xor (^) between pairs of bytes from some plaintext message (A) and a secret key (B) of the same length, we get an encrypted message (C). The same process can be used on the encrypted message (C) with the key (B) to reproduce the original plaintext (A). Two parties who know a secret shared key can communicate securely using this method.

Of course, messages tend to be much longer than secret keys. A simple solution is to repeat the key as many times as necessary. For example, if we wanted to encrypt the message "hello" with the key "yo", we would do:

```
    h        e        l        l        o
    01101000 01100101 01101100 01101100 01101111
^
    y        o        y        o        y
    01111001 01101111 01111001 01101111 01111001
_____________________________________________

    00010001 00001010 00010101 00000011 00010110
```

(Note that the encrypted bytes are not even printable ASCII characters)

We implemented this simple XOR cipher in **crypto_simple.c**; the key is taken as a command line argument, and the plaintext/encrypted message is read in as standard input. Compile and run a quick test:

```
gcc crypto_simple.c -o crypto
./crypto samplekey
```

Terminate your input with Command-D or Ctrl-D, which sends an EOF signal. Hitting Enter will only flush the input and produce encrypted output for the input so far, including the encrypted \n, and then await more input.

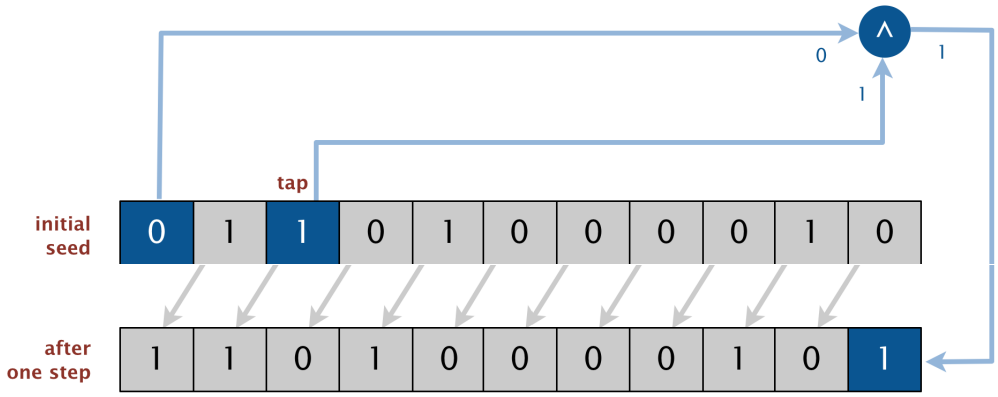
The message input/output can also be redirected or piped. See the class notes for examples.

Background – LFSR

A short, repeating key is not nearly as secure as a long, non-repeating one. One way to produce arbitrarily long, pseudorandom byte strings is a *linear-feedback shift register* (LFSR).

An LFSR is a register (fixed size list) of bits that can be used to produce a series of pseudorandom bits by a series of discrete "steps." Every "step" does three things:

- Shifts the bits one position to the left
- Replaces the vacated bit by the XOR of the bit shifted off and the bit previously at a given tap position in the register.
- Outputs the newly generated bit



one step of an 11-bit LFSR with initial seed 01101000010

Given a particular "seed" (the initial contents of the register) and tap position, an LFSR's output is deterministic but pseudorandom, and can go for a long time without repeating. Thus, they can be used to produce arbitrarily long keys for XOR ciphers, improving security while only requiring a secret, shared "seed" and tap index.

LFSR's are typically digital circuits, but you will simulate one in C code. LFSR's are usually also registers of **bits**, but your LFSR will instead be a register of **chars** (which are each 1 byte) instead, which is much easier to work with than individual bits. Each "step" of our simulated LFSR will then produce a pseudorandom char/byte that can be used to encrypt/decrypt the next char/byte of the message.

Your Task

You will implement an LFSR that generates pseudorandom bytes at every "step" using an array (**lfsr_arr.c**). You will also write an updated version of **crypto_simple.c** that utilizes your LFSR to perform the XOR cipher (**crypto_lfsr.c**).

1) First, create the file **lfsr_arr.c;** include and implement the interface defined in **lfsr.h**, which has been provided for you. You will need to define the `struct lfsr` (which is `typedef`'d in **lfsr.h** as `LFSR`), which should utilize

an array to hold the register's contents.

As you implement each function, note the performance requirements noted for certain functions. Remember that CONSTANT time generally implies no loops; LINEAR time implies loop(s) usage, but no nesting.

2) To test your implementation, the program **lfsr_test.c** has been provided for you. It needs to be compiled with your **lfsr_arr.c**.

```
gcc lfsr_test.c lfsr_arr.c -o lfsr_test
./lfsr_test
```

See the Sample Output below for what you should expect on success.

3) Now, copy **crypto_simple.c** over to a new file called **crypto_lfsr.c.** Include **lfsr.h** at the top, and then update the code to require 2 command line arguments: a seed (index 1), and a tap position (index 2). The tap argument can be converted from a string to an int using the **sscanf** function, which does formatted scanning from a string, rather than stdin (like scanf). See below:

```
int tap;
sscanf(argv[2],"%i", &tap);
```

With the given seed and tap position, create an LFSR and use it to generate random bytes for every step of the encryption/decryption process.

4) To test it, compile **crypto_lfsr.c** with **lfsr_arr.c.** You can run it similarly to the **crypto_simple** program, only you must provide both the seed and the tap as command line arguments.

```
gcc crypto_lfsr.c lfsr_arr.c -o crypto
./crypto sampleseed 7
```

Try to encrypt and decrypt various messages with different seeds and tap positions. Make sure that different seeds/taps produce different encryptions, and that the decryption results in the same as your original message!

5) You should further confirm that your cipher is correctly implemented by encrypting various message files in the **plain_message_files** directory with various seeds in the **seed_files** directory and various tap positions (5,7,or 9).

You can pass the contents of a file as a command line argument with "`$(<FILEPATH)`". Here's an example of how to pass a file as the seed:

```
./crypto "$(<seed_files/1_tiny.seed)" 5 <plain_message_files/1_tiny.txt >out.txt
```

The directory **encrypted_message_files** contains the expected result for each combination, and can be checked against your own output with the **`diff`** command. Here's an example:

```
diff out.txt encrypted_message_files/1_tiny.seed-tap_5-1_tiny.txt.crypt
```

WARNING: Start with the smaller seed files and plain message files, and work your way up.

Discussion Break! What's with the slowness???

If you tried some of the larger message/key files above, you may have noticed that run time was beginning to get long. Each size up on either the seed or message file increases its size 10 fold, and the time grows accordingly. For example: using **4_large.seed** to encrypt **5_huge.txt** will likely take 5-10 seconds. Increasing the size of either the seed or message will take over a minute; the biggest sizes **5_huge.seed** and 6**_enormous.txt** will take 10 or more minutes! (If its already very fast for you, you may have already done the extension below – so good job!)

This is because the time complexity of the `lfsrStep` function is LINEAR: the time required to do a step scales with the size of the seed - namely, the "shift" part of the step. Our choice of an array to represent the register is to blame here: "shifts" of all elements in an array involve touching every single position. *Is there a way (or two) to fix this…?*

EXTENSION OPPORTUNITY: Research "circular array" and update **lfsr_array.c** so that `lfsrStep` actually works in CONSTANT time.

Sample Output of lfsr_test.c

```
Running Test set 1:

LFSR created with seed aBcDeFg and tap 3
        | toString: aBcDeFg | lsfrHead: 61 (a) | lsfrTap: 44 (D)
Step 1:
        lsfrStep: 25 (%) | toString: BcDeFg% | lsfrHead: 42 (B) | lsfrTap: 65 (e)
Step 2:
        lsfrStep: 27 (') | toString: cDeFg%' | lsfrHead: 63 (c) | lsfrTap: 46 (F)
Step 3:
        lsfrStep: 25 (%) | toString: DeFg%'% | lsfrHead: 44 (D) | lsfrTap: 67 (g)
Step 4:
        lsfrStep: 23 (#) | toString: eFg%'%# | lsfrHead: 65 (e) | lsfrTap: 25 (%)
Step 5:
        lsfrStep: 40 (@) | toString: Fg%'%#@ | lsfrHead: 46 (F) | lsfrTap: 27 (')
Step 6:
        lsfrStep: 61 (a) | toString: g%'%#@a | lsfrHead: 67 (g) | lsfrTap: 25 (%)
Step 7:
        lsfrStep: 42 (B) | toString: %'%#@aB | lsfrHead: 25 (%) | lsfrTap: 23 (#)
Step 8:
        lsfrStep: 6 (□) | toString: '%#@aB□ | lsfrHead: 27 (') | lsfrTap: 40 (@)
Step 9:
        lsfrStep: 67 (g) | toString: %#@aB□g | lsfrHead: 25 (%) | lsfrTap: 61 (a)
Step 10:
        lsfrStep: 44 (D) | toString: #@aB□gD | lsfrHead: 23 (#) | lsfrTap: 42 (B)
destroyLFSR does not cause segFault, although correctness is not guaranteed
Running Test set 2:

LFSR created with seed AbCdEfG?! and tap 6
        | toString: AbCdEfG?! | lsfrHead: 41 (A) | lsfrTap: 47 (G)
Step 1:
```

```
        lsfrStep: 6 (□) | toString: bCdEfG?!□ | lsfrHead: 62 (b) | lsfrTap: 3f (?)
Step 2:
        lsfrStep: 5d (]) | toString: CdEfG?!□] | lsfrHead: 43 (C) | lsfrTap: 21 (!)
Step 3:
        lsfrStep: 62 (b) | toString: dEfG?!□]b | lsfrHead: 64 (d) | lsfrTap: 6 (□)
Step 4:
        lsfrStep: 62 (b) | toString: EfG?!□]bb | lsfrHead: 45 (E) | lsfrTap: 5d (])
Step 5:
        lsfrStep: 18 (□) | toString: fG?!□]bb□ | lsfrHead: 66 (f) | lsfrTap: 62 (b)
Step 6:
        lsfrStep: 4 (□) | toString: G?!□]bb□□ | lsfrHead: 47 (G) | lsfrTap: 62 (b)
Step 7:
        lsfrStep: 25 (%) | toString: ?!□]bb□□% | lsfrHead: 3f (?) | lsfrTap: 18 (□)
Step 8:
        lsfrStep: 27 (') | toString: !□]bb□□%' | lsfrHead: 21 (!) | lsfrTap: 4 (□)
Step 9:
        lsfrStep: 25 (%) | toString: □]bb□□%'% | lsfrHead: 6 (□) | lsfrTap: 25 (%)
Step 10:
        lsfrStep: 23 (#) | toString: ]bb□□%'%# | lsfrHead: 5d (]) | lsfrTap: 27 (')
destroyLFSR does not cause segFault, although correctness is not guaranteed
```

Grading

This assignment will be scored out of 100 points.

70 pts for **lfsr_arr.c**

- 10pts: createLSFR works
- 10pts: lsfrHead and lsfrTap work in CONSTANT time
- 20pts: lsfrStep works in LINEAR time
- 20pts: lsfrToString and lsfrDebugPrint work
- 10pts: destoryLSFR works

30 points for **crypto_lfsr.c**

- 10 pts: Reads in seed and tap command line arguments and creates an LFSR
- 20 pts: Uses LFSR correctly to encrypt/decrypt message from std in.