Applications of Bit Manipulation

Due: Wed March 14 at 11:59pm
Extension: Fri March 16 11:59pm
(No submissions after Friday March 16 will be accepted)

A. Pseudo-random number generator (prng)

The type of computers we are programming in CSC190 are deterministic: given instructions in a particular sequence, there is only ONE result that is possible.

This may lead you to wonder how we were able to exploit a "random" function back in earlier labs. The answer to this: these functions were not random, but rather pseudo-random.

The functions maintained an internal variable (a state variable, like your variables in a class that maintain state), and generated a new number based on the old number and a particular state evolution function. The state evolution function was chosen so that the numbers produced seemed to have the statistics of a random variable --- but the numbers were produced via a deterministic method. This can be seen by looking at a sequence of random numbers from a random function, and searching for the period of that random function: that is, where the random sequence begins to repeat. You'll quickly see that your random function is nothing more than a good simulation of randomness.

How are these random functions constructed? There is a mathematical theory (finite fields) that enables one to construct long-period sequence generators. The math is out of scope, but we can use the results. The operating principle here is that of the "linear feedback shift register", or LFSR.

[A possible reference that goes into far too much detail is:
https://en.wikipedia.org/wiki/Linear-feedback_shift_register
You do not need to read this.]

The key ingredients here are the concepts of:
* feedback shift registers
* linear feedback

Recall the operation of shifting: ">>" is the shift right operator and "<<" is the shift left operator. It is clear what happens to the internal bits (i.e., the non-extremum bits) when the shift operator is applied.

Consider an 8-bit variable, x, defined in C as an unsigned char. When we apply ">>" to it:
* for the internal bits (bit 6 to bit 1)
the value at position i moves to position i-1
Now what about the extrema: bit 7 and bit 0.
* bit 7 gets a value of 0
* bit 0 is lost

Given any arbitrary value, with enough shifts (i.e., 8), the value of that variable will eventually become 0.

Similar logic can be applied to the case of shifting left; try it, and make sure you're clear on this.

What a feedback shift register does is take the bit[0] that would have gotten lost, and feed it into bit 7. Now you can see (and if you can not, try this on paper) that with every 8 shifts

the original pattern will be restored.

How do you do this programmatically?

```
unsigned char FSR(unsigned char x) {
   unsigned char oldbit0 = x & 0x1; /* extract bit 0 */
   unsigned char r = x >> 1;        /* shift right   */
   unsigned char newbit7 = oldbit0 << 7; /* move bit0 to the bit7 pos */
   r = r | newbit7; /* rotate the old value of bit 0 into the bit 7 pos */
   return r;
}
```

Test this function with arbitrary values and confirm it works as advertised. Test this function and show that it repeats with a period of 8.

This isn't very random however: every value is related by a factor of 2. (Don't see this? Test it, print the values.)

So we've covered the FSR part of LFSR:

```
   y[n+1] = f(y[n])
```

where f() is the composition of shifting and rotation.

We can make the f() a bit more complex, via a linear function, yielding a "linear feedback shift register".

We're going to use a Galois function here, which is produced via the aforementioned finite fields mathematical theory.

All you need to know is that after you generate the shifted and rotated value, you must XOR the resulting value with a particular pattern prior to returning it.

Exercise 1:
Write the function:

```
unsigned char prng(unsigned char x, unsigned char pattern) {
 ...
}
```

that XORs the shifted and rotated value of x (as per the above discussion) with "pattern".

Test this code with pattern=0xb8, and look at the values produced. What is the period? Can you identify any simple relationship between adjacent elements of the generated sequence?

B. Cryptography

Cryptography is the science behind the hiding of data (from our Hellenic friends: crypto="hidden", graphy="writing").

The basic idea is:

* user inputs some secret password to do with the data he/she wishes to hide
* the password is somehow "mixed" with the data to generate some new data that "hides" the original

The above is called "encryption". For this to be useful, one must also be able to do the reverse. That is, the user who is privy to the data should be able to provide the secret password and obtain the original data. As well, one should be able to share the mixed data with a general user, and that general user should not be able to easily recover the data.

[The qualifier "easily" is important here: it is always possible for a non-permitted user to brute force attack the hidden message

and generate a series of candidate secret passwords, unmix
the data with each secret password, and see if there's an
intelligible message. It is a design decision for the user of
cryptographic systems as to how hard to make this brute force
attack. Again: all of this is out of scope, but good to know.]

Decryption is:
   * user inputs some secret password
   * encrypted data is unmixed with the secret password
     to generate a decrypted data stream which should be
     identical to the original if the user entered
     the correct password

We're going to prototype this with a very simple algorithm:

 * user will enter a secret password that will be INITIAL VALUE
   of your prng stream
 * the encrypt and decrypt functions will take as input this
   password, as well as a char * that represents the string the
   user wishes to process.

Write the function:

```
int crypt(char *data,unsigned int size,unsigned char password) {
}
```

"data" will be a caller-allocated char array of size "size+1"
(+1 because we need to include an end '\0' char)
"password" will be the starting value for the prng and CAN NOT
be zero.

The function will return 0 on success, and -1 on failure.

"data" holds the input string, and will also be used to store the output.

One last piece of data: how do we mix the password and the data?
For each character of the data stream, starting from idx 0, upto the end
of the string, XOR the char of the string with the current value out of
the prng:

```
    prngVal = <password>
    for i in range(0,size,1):
       prngVal = prng(prngVal,0xb8)
       new char at i = (old char at i) ^ prngVal
```

Note: this is only a very basic example of cryptogeaphy. It is highly insecure.
Don't use this for data you actually want to protect!

Submit a file called crypt.c which will have the prng and crypt functions above.
Test you code: put in some string and see how the transormed data looks
(both encrypted and unencrypted). Submit will take an arg of 6.