Chapter 6 : Modern symmetric encryption.

First, from here on we will work in binary. So we'll have to think in sums of powers of 2 instead of powers of 10 from here on out.

E.g. $37 = 2^5 + 2^2 + 2^0$

$= 100101$

or $7 = 2^2 + 2^1 + 2^0$

$= 111$

Of course, we can still encrypt messages involving letters/punctuation etc, we just have to decide on a correspondence between letters/punctuation and binary numbers. Thankfully this is already done for us.

E.g. ASCII

American Standard Code for Information Interchange. Each letter corresponds to a 5-digit (or 5-bit) binary string.

E.g: HI becomes $\underbrace{00111}_{H}\ \underbrace{01000}_{I}$.

The benefit of binary lies in the simplicity of operations on binary numbers and strings.

There's good old addition:

$$\begin{array}{r} {\scriptstyle 1\ \ 1} \\ 1101 \\ +\ \ 101 \\ \hline 10010 \end{array}$$

or in more familiar notation:

$$\begin{array}{r} 13 \\ +\ 5 \\ \hline 18 \end{array}$$

$= 2^4 + 2^1 = 18$ (it checks out).

There's also "bitwise addition" where we just add the entries mod 2, with no carrying (ie use: $0+0=0$, $1+0=0+1=1$, $1+1=0$).

Also called XOR (eXclusive Or). It's like this; written $\oplus$:

$$\begin{array}{r} 0010111101 \\ \oplus\ 1011111100 \\ \hline 1001000001 \end{array}$$

One of our first goals will be to use binary to generate from a relatively small key a long random-seeming string of 0's and 1's, useful in stream ciphers (for example).

§6.2 Linear feedback shift registers.

Our notation will be the same as the book, meaning we have to think right-to-left.

An LFSR is a way of creating a new string of 0's and 1's from a set of "transition rules".

Example: We are going to transform the 4-bit string (b_4', b_3', b_2', b_1') into (b_4, b_3, b_2, b_1). We write this as

$$(b_4, b_3, b_2, b_1) \longleftarrow (b_4', b_3', b_2', b_1').$$

To describe the change, we write "transition rules":

$b_3 \longleftarrow b_4'$ $\quad$ $b_2 \longleftarrow b_3'$

$b_1 \longleftarrow b_2'$ $\quad$ $b_4 \longleftarrow b_1' + b_2'$

To make this notation make sense, we can read the arrow "$\longleftarrow$" as "comes from". So, with the rules above,

1011 corresponds to $b_4' = 1$, $b_3' = 0$, $b_2' = 1$, $b_1' = 1$

and it is transformed into

0101

$b_3 \longleftarrow b_2'$, etc...

$b_4 \longleftarrow b_1' + b_2'$

In general, a LFSR is a single formula, such as

$$b_{\textcircled{5}} \longleftarrow b_1' + b_2' + b_4'$$

where the subscript on the left indicates the size of the string, and the formula indicates the only "non-trivial" transition rule. I.e. all other transition rules are of the form $b_i \longleftarrow b_{i+1}'$.

An initial value used to compute subsequent strings is called the <u>seed</u> for the register.

<u>Example continued</u>: Using the LFSR $b_4 \longleftarrow b_1' + b_2'$ above, and a seed of 0010, we can compute a table of subsequent strings:

Step	string			
0	0	0	1	0 ← seed.
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	1	0	1	1
5	0	1	0	1
6	1	0	1	0
				⋮

note this column may seem totally random to someone who does not know the rule $b_4 \leftarrow b_1' + b_2'$.

The column on the right will eventually repeat, since there are at most $1111 = 2^3 + 2^2 + 2^1 + 2^0 + 1 = 16$ arrangements of four zeroes and 1's. We can use these bitstreams for stream ciphers.

Example: Consider the LFSR $b_4 \leftarrow b_1' + b_2' + b_4'$, seeded with 0110. We start computing states, and get:

step	string
0	0 1 1 0
1	1 0 1 1
2	1 1 0 1
3	0 1 1 0
4	1 0 1 1
5	⋮

repeats soon!

Now to encrypt a message like "HELP" we write it in binary using ASCII, get:

	H				E				L				P			
plaintext	0	0	1	1	1	0	0	1	0	0	0	1	0	1	1	0	1	1	1	1
keystream ⊕ (XOR)	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	0	1
ciphertext	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0	0	0	1	0
change back:	k				S				Q				C				.			

To decrypt: Pass the rule $b_4 \longleftarrow b_1' + b_2' + b_4'$ to the intended recipient. They generate the same keystream:

0 1 1 0 1 1 0 1 1 ... etc.

Then XOR the ciphertext with the keystream, taking advantage of the fact that subtraction mod 2 is the same as addition mod 2! So:

ciphertext	0	1	0	1	0	1	0	0	....
keystream ⊕	0	1	1	0	1	1	0	1	....
plaintext	0	0	1	1	1	0	0	1	... etc.

Remark: If we chose a LFSR with many bits, then it might not repeat for a long time, unlike the previous example. In general, a LFSR

$$b_k \longleftarrow \text{some equation}$$

doesn't necessarily repeat before the 2^k-th step (though some might repeat much sooner). So we could potentially communicate only a simple equation and a length k seed, and generate a string $\sim 2^k$ bits long that is "effectively random" to an observer.

In fact, we could make the formula public! The seed alone will generate a keystream for the intended recipient, and an eavesdropper would still be lost as to how to generate the keystream.

§6.3 : A vulnerability: Known plaintext attacks.

Recall we want a system where an eavesdropper that somehow discovers a large chunk of our plaintext will still be thwarted.

So suppose that someone knows the LFSR we're using (ie the formula). We send the ciphertext

WIMSUJ

using $b_4 \leftarrow b_4' + b_1'$. Our seed is secret.

Unfortunately, an eavesdropper finds out that the first two letters are "MR".

ciphertext : 1011001000 ....

they know : 0110010001 ....
← plaintext.

Since addition mod 2 is the same as subtraction, they need only XOR these to get the start of the keystream:

keystream: 1101011001

So if we imagine filling in our LFSR table: