Sage

Any serious cryptography involves computations that need to be done by a machine. Let us see how to use the open-source computer algebra system **Sage** to do basic computations for us.

Sage is freely available at sagemath.org. Instead of installing it locally (it's huge!) we can conveniently use it in the cloud at cocalc.com from any browser.

Sage is built as a **Python** library, so any Python code is valid. For starters, we will use it as a fancy calculator.

Example 55. Let's start with some basics.

```
Sage] 17 % 12
```

5

```
Sage] (1 + 5) % 2  # don't forget the brackets
```

0

```
Sage] inverse_mod(17, 23)
```

19

```
Sage] xgcd(17, 23)
```

$(1, -4, 3)$

```
Sage] -4*17 + 3*23
```

1

```
Sage] euler_phi(84)
```

24

Example 56. Why is the following bad?

```
Sage] 3^1003 % 101
```

27

The reason is that this computes 3^{1003} first, and then reduces that huge number modulo 101:

```
Sage] 3^1003
```

$$35695912125981779196042292013307897881066394884308000526952849942124372128361032287601\\
01447396641767302556399781555972361067577371671671062036425358196474919874574608035466\\
17047063989041820507144085408031748926871104815910218235498276622866724603402112436668\\
09387969298949770468720050187071564942882735677962417251222021721836167242754312973216\\
80102291029227131545307753863985171834477895265551139587894463150442112884933077598746\\
04125161734774642865878855686737747603770909400 27$$

We know how to efficiently avoid computing huge intermediate numbers (binary exponentiation!). Sage does the same if we instead use something like:

```
Sage] power_mod(3, 1003, 101)
```

27

Review.

- A **pseudorandom generator** (PRG) takes a seed x_0 and produces a stream $\mathrm{PRG}(x_0) = x_1\, x_2\, x_3 \ldots$ of numbers, which should be close to random numbers.

  For cryptographic purposes, these numbers should be indistinguishable from random numbers. Even for somebody who knows everything about the PRG except the seed. (See Example 54.)

- Once we have a PRG, we can use it as a **stream cipher**: Using the key k, we encrypt $E_k(m) = m \oplus \mathrm{PRG}(k)$. [Here, the key stream $\mathrm{PRG}(k)$ is assumed to be in bits.]

  As with the one-time pad, we must never reuse the same keystream!

- To reuse the key, we can use a **nonce**: $E_k(m) = m \oplus \mathrm{PRG}((\mathrm{nonce}, k))$, where the seed is produced by combining the nonce and k (for instance, just concatenating them).

  The nonce is then passed (unencrypted) along with the message.

  To never reuse the same keystream, we must never use the same nonce with the same key.

Linear feedback shift registers

Here is another basic idea to generate pseudorandom numbers:

(linear feedback shift register (LFSR) Let ℓ and $c_1, c_2, \ldots, c_\ell$ be chosen parameters. From the seed $(x_1, x_2, \ldots, x_\ell)$, where each x_i is one bit, we produce the sequence

$$x_{n+\ell} \equiv c_1 x_{n+\ell-1} + c_2 x_{n+\ell-2} + \ldots + c_\ell x_n \pmod 2.$$

This method is particularly easy to implement in hardware (see Example 58), and hence suited for applications that value speed over security (think, for instance, encrypted television).

Example 57. Which sequence is generated by the LFSR $x_{n+2} \equiv x_{n+1} + x_n \pmod 2$, starting with the seed $(x_1, x_2) = (0, 1)$?

Solution. $(x_1, x_2, x_3, \ldots) = (0, 1, 1, 0, 1, 1, \ldots)$ has period 3.

Note. Observe that the two previous values determine the state, so there is $2^2 = 4$ states of the LFSR. The state $(0, 0)$ is special (it generates the zero sequence $(0, 0, 0, 0, \ldots)$), so there is 3 other states. Hence, it is clear that the generated sequence has to repeat after at most 3 terms.

Comment. Of course, if we don't reduce modulo 2, then the sequence $x_{n+2} = x_{n+1} + x_n$ generates the Fibonacci numbers $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

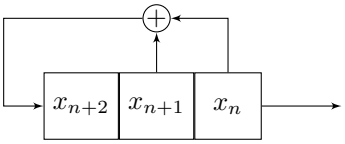
Example 58. Which sequence is generated by the LFSR $x_{n+3} \equiv x_{n+1} + x_n \pmod 2$, starting with the seed $(x_1, x_2, x_3) = (0, 0, 1)$? What is the period?

[Let us first note that the LFSR has $2^3 = 8$ states. Since the state $(0, 0, 0)$ remains zero forever, 7 states remain. This means that the generated sequence must be periodic, with period at most 7.]

Solution. $(x_1, x_2, x_3, \ldots) = (0, 0, 1, 0, 1, 1, 1, 0, 0, 1, \ldots)$ has period 7.

Again, this is not surprising: 3 previous values determine the state, so there is $2^3 = 8$ states. The state $(0, 0, 0)$ is special, so there is 7 other states.

Note that this LFSR can be implemented in hardware using three registers (labeled x_n, x_{n+1}, x_{n+2} in the sketch to the right). During each cycle, the value of x_n is read off as the next value produced by the LFSR.



Note. In the part $0, 0, 1, 0, 1, 1, 1$ that repeats, the bit 1 occurs more frequently than 0.

The reason for that is that the special state $(0, 0, 0)$ cannot appear.

For the same reason, the bit 1 will always occur slightly more frequently than 0 in LFSRs. However, this becomes negligible if the period is huge, like $2^{31} - 1$ in Example 59.

Example 59. The recurrence $x_{n+31} \equiv x_{n+28} + x_n \pmod 2$, with a nonzero seed, generates a sequence that has period $2^{31} - 1$.

Note that this is the maximal possible period: this LFSR has 2^{31} states. Again, the state $(0, 0, ..., 0)$ is special (the entire sequence will be zero), so that there is $2^{31} - 1$ other states. This means that the terms must be periodic with period at most $2^{31} - 1$.

Comment. glibc (the second implementation) essentially uses this LFSR.

Advanced comment. One can show that, if the characteristic polynomial $f(T) = x^\ell + c_1 x^{\ell-1} + c_2 x^{\ell-2} + ... + c_\ell$ is irreducible modulo 2, then the period divides $2^\ell - 1$. Here, $f(T) = T^{31} + T^{28} + 1$ is irreducible modulo 2, so that the period divides $2^{31} - 1$. However, $2^{31} - 1$ is a prime, so that the period must be exactly $2^{31} - 1$.

We have seen two simple examples of PRGs so far:

- linear congruential generators $x_{n+1} \equiv a x_n + b \pmod m$
- LFSRs $x_{n+\ell} \equiv c_1 x_{n+\ell-1} + c_2 x_{n+\ell-2} + ... + c_\ell x_n \pmod 2$

Of course, we could also combine LFSRs and linear congruential generators (i.e. look at recurrences like for LFSRs but modulo any parameter m).

However, much of the appeal of an LFSR comes from its extremely simple hardware realization, as the sketch in Example 58 indicates.

Example 60. (extra) One can also consider nonlinear recurrences (it mitigates some issues). Our book mentions $x_{n+3} \equiv x_{n+2} x_n + x_{n+1} \pmod 2$. Generate some numbers.

Solution. For instance, using the seed $0, 0, 1$, we generate $\overbrace{0, 0, 1}^{\text{seed}}, 0, 1, 1, 1, 0, 1, ...$ which now repeats (with period 4) because the state $1, 0, 1$ appeared before. Observe that the generated sequences is only what is called eventually periodic (it is not strictly periodic because $0, 0, 1$ never shows up again).