Example 48. (bonus!) $p = 29137$ is an example of a left-truncatable prime: the number itself as well as all truncations 9137, 137, 37, 7 are prime. By simply exhausting all possibilities (start with a single digit and keep adding (nonzero) digits on the left until no choice results in a prime), we find that there is a largest left-truncatable prime, namely, 357686312646216567629137.

https://www.youtube.com/watch?v=azL5ehbw_24

Challenge. Find the largest left-truncatable prime which does not have 9 as a digit.

Send me the prime, and an explanation how you found it, by Feb 3 for a bonus point!

Comment. You can play the same game in bases different from 10. We expect that (based on the prime number theorem), for any base, there is always just going to be a finite number of truncatable primes (an extra bonus if you can point me to a proof of that claim!), though the number tends to increase with larger bases. The largest truncatable prime for base 30, for instance, is not known (it is estimated to have about 82 digits in base 30).

https://oeis.org/A103463

Example 49. One thing that makes the one-time pad difficult to use is that the key needs to be the same length as the plaintext. What if we have a shorter key and just repeat it until it has the length we need?

That's essentially the Vigenere cipher (in a different alphabet).

Solution. Assuming the attacker knows the length of our key (if she doesn't she can just try all possibilities), this is equivalent to using the one-time pad several times with the same key. That should never be done! Even using a key twice means that we become susceptible to a ciphertext only attack (see Example 46).

So, repeating the key is a terrible idea. However, the idea to create a longer (random) key out of a shorter (random) key is not (these are pseudorandom generators, to be discussed next).

Let us emphasize that, in order to be perfectly confidential, the key for a one-time pad must be chosen completely at random (otherwise, an attacker can make assumptions on the used keys).

Indeed, the need to generate random numbers shows in every modern cipher.

Stream ciphers

Once we have a way to generate **pseudorandom numbers**, we can use the idea of the one-time pad to create a **stream cipher**.

Start with key of moderate size (say, 128 bits).

Use the key k and a PRG (**pseudorandom generator**) to generate a much longer **pseudorandom keystream** $\mathrm{PRG}(k)$. Then encrypt $E_k(m) = m \oplus \mathrm{PRG}(k)$.

We lost perfect confidentiality. Security relies on choice of PRG (must be unpredictable).

As with the one-time pad, we must never reuse the same keystream! That does not mean that we cannot reuse the key: we can do that using 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 make sure that we never reuse the same keystream, we must never use the same nonce with the same key.

How to generate random numbers?

Natural randomness is surprisingly difficult to harness.

You can for instance play around with a Geiger counter but our department is short on these and getting lots of random numbers is again challenging.

Linear congruential generators

(linear congruential generator) Let a, b, m be chosen parameters.
From the seed x_0, we produce the sequence $x_{n+1} \equiv a x_n + b \pmod{m}$.

The choice of a, b, m is crucial for this to generate acceptable pseudorandom numbers.

For instance, glibc uses $a = 1103515245$, $b = 12345$, $m = 2^{31}$. (This is one of two implementations.) In that case, each x_i is represented by precisely 31 bits. [Note that the choice of m makes this very fast.]

https://en.wikipedia.org/wiki/Linear_congruential_generator

Linear congruential generators (LCG) are easy to predict and must not be used for cryptographic purposes. More generally, all polynomial generators are cryptographically insecure. They are still used in practice, because they are fast and easy to implement and have decent statistical properties. (For instance, our online homework is generated using random numbers, and there is no need for crypto-level security there.)

Statistical trouble. Can you see why the sequences produced by the glibc LCG alternate between even and odd numbers? (Similarly, other low bits are much less "random" than the higher bits.) Because of this defect, some programs (and other implementations of `rand()` based on LCGs) throw away the low bits entirely.

Example 50. Generate values using the linear congruential generator $x_{n+1} \equiv 5x_n + 3 \pmod{8}$, starting with the seed $x_0 = 6$.

Solution. $x_1 \equiv 1$, $x_2 \equiv 0$, $x_3 \equiv 3$, $x_4 \equiv 2$, $x_5 \equiv 5$, $x_6 \equiv 4$, $x_7 \equiv 7$, $x_8 \equiv 6$. This is the value x_0 again, so the sequence will now repeat. Note that we went through all 8 residues before repeating. Period 8.

Note. Because $8 = 2^3$ we can represent each x_i using exactly 3 bits. Then $x_1, x_2, x_3, \ldots = 1, 0, 3, \ldots$ corresponds to the bit stream $(001\ 000\ 011\ \ldots)_2$.

Example 51. (extra) Observe that the sequence produced by the linear congruential generator $x_{n+1} \equiv a x_n + b \pmod{m}$ must repeat, at the latest, after m terms. (Why?!)

One can give precise conditions on a, b, m to achieve a full period m. Namely, this happens if and only if $\gcd(b, m) = 1$ and $a - 1$ is divisible by all primes (as well as 4) dividing m.

(a) Generate values using a linear congruential generator $x_{n+1} \equiv 2x_n + 1 \pmod{10}$, starting with the seed $x_0 = 5$. When do they repeat? Is that consistent with the mentioned condition?

(b) What are possible values for a so that the LCG $x_{n+1} \equiv a x_n + 11 \pmod{100}$ has period 100?

(c) glibc uses $a = 1103515245$, $b = 12345$, $m = 2^{31}$. After how many terms will the sequence repeat?

Solution.

(a) $x_1 \equiv 1$, $x_2 \equiv 3$, $x_3 \equiv 7$, $x_4 \equiv 5$. This is the value x_0 again, so the sequence will repeat. Period 4. [The period is less than 10. This is as predicted by the mentioned condition, because $a - 1$ is not divisible by 2 and 5.]

(b) We need that $a - 1$ is divisible by 4 and 5. Equivalently, $a \equiv 1 \pmod{20}$. Hence, possible values are $a = 1, 21, 41, 61, 81$.

(c) Clearly, $\gcd(b, m) = 1$. Also, $a - 1$ is divisible by 4 (and no primes other than 2 divide m). Hence, for every seed, values repeat only after going through all 2^{31} residues.

Example 52. Let's use the PRG $x_{n+1} \equiv 5x_n + 3 \pmod 8$ as a stream cipher with the key $k = 4 = (100)_2$. The key is used as the seed x_0 and the keystream is $\mathrm{PRG}(k) = x_1 x_2 \ldots$ (where each x_i is 3 bits). Encrypt the message $m = (101\ 111\ 001)_2$.

Solution. We first use the PRG with seed $x_0 = k$ to produce the keystream $\mathrm{PRG}(k) = 7,\ 6,\ 1,\ \ldots = (111\ 110\ 001\ \ldots)_2$.

We then encrypt and get $c = E_k(m) = m \oplus \mathrm{PRG}(k) = (101\ 111\ 001)_2 \oplus (111\ 110\ 001)_2 = (010\ 001\ 000)_2$.

Decryption. Observe that decryption works in the exact same way:

$D_k(c) = c \oplus \mathrm{PRG}(k) = (010\ 001\ 000)_2 \oplus (111\ 110\ 001)_2 = (101\ 111\ 001)_2$.

Note. The keystream continues as $\mathrm{PRG}(k) = 7, 6, 1, 0, 3, 2, 5, 4, \ldots$ At this point it repeats itself because we obtained the value 4, which was our seed. Since the state of this PRG only depends on the value of x_n, and there is 8 possible values for x_n, the period 8 is the longest possible. The previous (extra) example gave conditions on the PRG that guarantee that the period is as long as possible.

Example 53. Can you think of a way in which the numbers produced by a linear congruential generator differ from truly random ones?

Solution. An easy observation for our small examples is the following: by construction, $x_{n+1} \equiv a x_n + b \pmod m$, individual values don't repeat unless a full period is reached and everything repeats. Truly random numbers do repeat every now and then (however, if m is large, then this observation is not exactly practical).

Of course, knowing the parameters a, b, m, the numbers generated by the PRG are terribly **predictable**. Knowing just one number, we can produce all the next ones (as well as the ones before). A PRG that is safe for cryptographic purposes should not be predictable like that! (See next example.)

The next example illustrates the vulnerability of stream ciphers, based on predictable PRGs.

Recall that it is common to know or guess pieces of plaintexts; for instance every PDF begins with $\%\mathrm{PDF}$.

Example 54. Eve intercepts the ciphertext $c = (111\ 111\ 111)_2$. It is known that a stream cipher with PRG $x_{n+1} \equiv 5x_n + 3 \pmod 8$ was used for encryption. Eve also knows that the plaintext begins with $m = (110\ 1\ldots)_2$. Help her crack the ciphertext!

Solution. Since $c = m \oplus \mathrm{PRG}$, we learn that the initial piece of the keystream is $\mathrm{PRG} = m \oplus c = (110\ 1\ldots)_2 \oplus (111\ 1\ldots)_2 = (001\ 0\ldots)_2$. Since each x_n is 3 bits, we conclude that $x_1 = (001)_2 = 1$.

Because the PRG is predictable, we can now recreate the entire keystream! Using $x_{n+1} \equiv 5x_n + 3 \pmod 8$, we find $x_2 \equiv 0$, $x_3 \equiv 3$, ... In other words, $\mathrm{PRG} = 1, 0, 3, \ldots = (001\ 000\ 011\ \ldots)_2$.

Hence, Eve can decrypt the ciphertext and obtain $m = c \oplus \mathrm{PRG} = (111\ 111\ 111)_2 \oplus (001\ 000\ 011)_2 = (110\ 111\ 100)_2$.