Sketch of Lecture 1

Mon, 1/8/2024

Review: The calculus of congruences

Example 1. Today is Monday. What day of the week will it be a year (366 days!) from now?

Solution. Since $366 \equiv 2 \pmod 7$, it will be Wednesday on 1/8/2025.

$$a \equiv b \pmod n \qquad \text{means} \qquad a = b + mn \quad (\text{for some } m \in \mathbb{Z})$$

In that case, we say that "a is congruent to b modulo n".

In other words: $a \equiv b \pmod n$ if and only if $a - b$ is divisible by n.

Example 2. $17 \equiv 5 \pmod{12}$ as well as $17 \equiv 29 \equiv -7 \pmod{12}$

We say that $5, 17, 29, -7$ all represent the same **residue** modulo 12.

There are exactly 12 different residues modulo 12.

Example 3. Every integer x is congruent to one of $0, 1, 2, 3, 4, \ldots, 11$ modulo 12.

We therefore say that $0, 1, 2, 3, 4, \ldots, 11$ form a **complete set of residues** modulo 12.

Another natural complete set of residues modulo 12 is: $0, \pm 1, \pm 2, \ldots, \pm 5, 6$

[-6 is not included because $-6 \equiv 6$ modulo 12.]

Online homework. When entering solutions modulo n for online homework, your answer needs to be from one of the two natural sets of residues above.

Example 4. Modulo 7, we have the complete sets of residues $0, 1, 2, 3, 4, 5, 6$ and $0, \pm 1, \pm 2, \pm 3$.
A less obvious set is $0, 3, 3^2, 3^3, 3^4, 3^5, 3^6$.

Review. Note that $3^6 \equiv 1 \pmod 7$ by **Fermat's little theorem**. Because 6 is the smallest positive exponent such that $3^k \equiv 1 \pmod 7$, we say that the **multiplicative order** of $3 \pmod 7$ is 6. This makes $3 \pmod 7$ a **primitive root**.

On the other hand, the **multiplicative order** of $2 \pmod 7$ is 3. (Why?!)

Example 5. $67 \cdot 24 \equiv 4 \cdot 3 \equiv 5 \pmod 7$

The point being that we can (and should!) reduce the factors individually first (to avoid the large number we would get when actually computing $67 \cdot 24$ first). This idea is crucial in the computations we (better, our computers) will later do for cryptography.

Example 6. (but careful!) If $a \equiv b \pmod n$, then $ac \equiv bc \pmod n$ for all integers c.
However, the converse is not true! We can have $ac \equiv bc \pmod n$ without $a \equiv b \pmod n$ (even assuming that $c \not\equiv 0$).

For instance. $2 \cdot 4 \equiv 2 \cdot 1 \pmod 6$ but $4 \not\equiv 1 \pmod 6$

However. $2 \cdot 4 \equiv 2 \cdot 1 \pmod 6$ means $2 \cdot 4 = 2 \cdot 1 + 6m$. Hence, $4 = 1 + 3m$, or, $4 \equiv 1 \pmod 3$.

The issue is that 2 is not invertible modulo 6.

$$a \text{ is invertible modulo } n \iff \gcd(a, n) = 1$$

Similarly, $ab \equiv 0 \pmod n$ does not always imply that $a \equiv 0 \pmod n$ or $b \equiv 0 \pmod n$.

For instance. $4 \cdot 15 \equiv 0 \pmod 6$ but $4 \not\equiv 0 \pmod 6$ and $15 \not\equiv 0 \pmod 6$

Armin Straub
straub@southalabama.edu

Good news. These issues do not occur when n is a **prime** p.

- If $ab \equiv 0 \pmod{p}$, then $a \equiv 0 \pmod{p}$ or $b \equiv 0 \pmod{p}$.
- Suppose $c \not\equiv 0 \pmod{p}$. If $ac \equiv bc \pmod{p}$, then $a \equiv b \pmod{p}$.

Example 7. Determine $4^{-1} \pmod{13}$.

Recall. This is asking for the **modular inverse** of 4 modulo 13. That is, a residue x such that $4x \equiv 1 \pmod{13}$.

Brute force solution. We can try the values $0, 1, 2, 3, \ldots, 12$ and find that $x = 10$ is the only solution modulo 13 (because $4 \cdot 10 \equiv 1 \pmod{13}$).

This approach may be fine for small examples when working by hand, but is not practical for serious congruences. On the other hand, the Euclidean algorithm, reviewed below, can compute modular inverses extremely efficiently.

Glancing. In this special case, we can actually see the solution if we notice that $4 \cdot 3 = 12$, so that $4 \cdot 3 \equiv -1 \pmod{13}$ and therefore $4^{-1} \equiv -3 \pmod{13}$.

Example 8. Solve $4x \equiv 5 \pmod{13}$.

Solution. From the previous problem, we know that $4^{-1} \equiv -3 \pmod{13}$.
Hence, $x \equiv 4^{-1} \cdot 5 \equiv -3 \cdot 5 = -2 \pmod{13}$.

(Bézout's identity) Let $a, b \in \mathbb{Z}$ (not both zero). There exist $x, y \in \mathbb{Z}$ such that

$$\gcd(a, b) = ax + by.$$

The integers x, y can be found using the **extended Euclidean algorithm**.
In particular, if $\gcd(a, b) = 1$, then $a^{-1} \equiv x \pmod{b}$ (as well as $b^{-1} \equiv y \pmod{a}$).

Here, $\mathbb{Z}$ denotes the set of all integers $0, \pm 1, \pm 2, \ldots$

Example 9. Find $d = \gcd(17, 23)$ as well as integers r, s such that $d = 17r + 23s$.

Solution. We apply the extended Euclidean algorithm:

$$\begin{array}{lll} \gcd(17,23) & \boxed{23} = 1 \cdot \boxed{17} + 6 & \text{or: } \boxed{A}\ \ 6 = 1 \cdot \boxed{23} - 1 \cdot \boxed{17} \\ = \gcd(6,17) & \boxed{17} = 3 \cdot \boxed{6} - 1 & \boxed{B}\ \ 1 = -1 \cdot \boxed{17} + 3 \cdot \boxed{6} \\ = 1 & & \end{array}$$

Backtracking through this, we find that:

$$1 \underset{\boxed{B}}{=} -1 \cdot \boxed{17} + 3 \cdot \boxed{6} \underset{\boxed{A}}{=} -4 \cdot \boxed{17} + 3 \cdot \boxed{23}$$

That is, **Bézout's identity** takes the form $1 = -4 \cdot 17 + 3 \cdot 23$.

Comment. Note how our second step was $\boxed{17} = 3 \cdot \boxed{6} - 1$ rather than $\boxed{17} = 2 \cdot \boxed{6} + 5$. The latter works as well but requires a third step (do it!). In general, we save time by allowing negative remainders if they are smaller in absolute value.

Example 10. Determine $17^{-1} \pmod{23}$.

Solution. By the previous example, $1 = -4 \cdot 17 + 3 \cdot 23$. Reducing modulo 23, we get $-4 \cdot 17 \equiv 1 \pmod{23}$. Hence, $17^{-1} \equiv -4 \pmod{23}$. [Or, if preferred, $17^{-1} \equiv 19 \pmod{23}$.]

Armin Straub
straub@southalabama.edu

Example 11. Determine $16^{-1} \pmod{25}$.

Solution. We apply the extended Euclidean algorithm:

$$\begin{aligned} &\gcd(16,25) && \boxed{25} = 2\cdot\boxed{16} - 7 && \text{or:} && \boxed{A}\quad 7 = -1\cdot\boxed{25} + 2\cdot\boxed{16} \\ &= \gcd(7,16) && \boxed{16} = 2\cdot\boxed{7} + 2 && && \boxed{B}\quad 2 = 1\cdot\boxed{16} - 2\cdot\boxed{7} \\ &= \gcd(2,7) && \boxed{7} = 3\cdot\boxed{2} + 1 && && \boxed{C}\quad 1 = \boxed{7} - 3\cdot\boxed{2} \\ &= 1 \end{aligned}$$

Backtracking through this, we find that:

$$1 \underset{\boxed{C}}{=} \boxed{7} - 3\cdot\boxed{2} \underset{\boxed{B}}{=} 7\cdot\boxed{7} - 3\cdot\boxed{16} \underset{\boxed{A}}{=} -7\cdot\boxed{25} + 11\cdot\boxed{16}$$

That is, **Bézout's identity** takes the form $-7\cdot 25 + 11\cdot 16 = 1$.

Reducing modulo 25, we get $11\cdot 16 \equiv 1 \pmod{25}$. Hence, $16^{-1} \equiv 11 \pmod{25}$.

Application: credit card numbers

Have you ever thought about the numbers on your credit card? Usually, these are 16 digits. For instance, 4266 8342 8412 9270.

No worries (or false hopes...). While close, this is not exactly my credit card number.

- The first digit(s) of a credit card identify the issuer of the card. For instance, a leading 4 is typically Visa, 51 to 55 indicate Mastercard, and 34, 37 indicate American Express. The above credit card is indeed a Visa card.

  More information at: https://en.wikipedia.org/wiki/Payment_card_number

- The last digit is a **check digit**, and a valid credit card number must pass the **Luhn check** (patented by IBM scientist Hans Peter Luhn in 1954/60; now in public domain).

  This works as follows: every second digit, starting with the first, is doubled. If that results in a two-digit number, we take the sum of those two digits.

$$\begin{bmatrix} & 4 & 2 & 6 & 6 & 8 & 3 & 4 & 2 & 8 & 4 & 1 & 2 & 9 & 2 & 7 & 0 \\ \times 2 & 8 & & 12 & & 16 & & 8 & & 16 & & 2 & & 18 & & 14 & \\ & 8 & 2 & 3 & 6 & 7 & 3 & 8 & 2 & 7 & 4 & 2 & 2 & 9 & 2 & 5 & 0 \end{bmatrix}$$

  The other half of the digits is left unchanged. We then add all these digits and reduce modulo 10:

$$8+2+3+6+7+3+8+2+7+4+2+2+9+2+5+0 \equiv 0 \pmod{10}$$

  The result of that computation must be 0. Otherwise, the credit card number fails the Luhn check and is invalid.

Example 12. (extra exercise)

(a) Check that the number 4266 8342 8412 9280 fails the Luhn check.

(b) How do we have to change the last digit to turn this into a valid credit card number?

Armin Straub
straub@southalabama.edu

The purpose of the Luhn check is to detect accidental errors.

[A random credit card number has a 90% chance of failing the Luhn check. Why?!]

On the other hand, as the previous example shows, it provides basically no protection against malicious attacks (except against amateur criminals not aware of the Luhn check).

The Luhn check was designed before online banking (patent filed in 1954). So a special focus is on detecting accidental errors that occur frequently when writing down (things like) credit card numbers by hand.

- For instance, it is common that a single digit gets messed up. Every such error is detected by the Luhn check. (Why?!)
- Another common error is to transpose two digits. Every such error (with the exception of 09 versus 90) is detected.

  For instance. A 82 at the beginning contributes $7+2=9$ to the check sum, while a 28 contributes $4+8\equiv 2$ to the sum. Hence, replacing one with the other will result in the Luhn check failing.

  Advanced comment. An alternative checksum formula that can detect all single digit changes as well as all transpositions is the Verhoeff algorithm (1969). It is, however, much more complicated and cannot be readily performed by hand.

Example 13. The doubling and sum-of-digits procedure permutes the digits as follows:

original digit	0	1	2	3	4	5	6	7	8	9
adjusted digit	0	2	4	6	8	1	3	5	7	9
difference $(\mathrm{mod}\,10)$	0	1	2	3	4	6	7	8	9	0

Note. Looking at the differences modulo 10, we can see why the Luhn check is able to detect all transposition errors (except 09 versus 90).

Example 14. The Luhn check has the somewhat complicated feature that every second digit has to be doubled. Why do we not just add all the original digits and reduce the sum modulo 10?

Solution. One reason is that this simplified check does not catch the transposition of two digits. Why?!

[On the other hand, that simplified check does also detect if just a single digit is incorrect.]

Example 15. (extra) The International Standard Book Number ISBN-10 consists of nine digits $a_1a_2\ldots a_9$ followed by a tenth check digit a_{10} (the symbol X is used if the digit equals 10), which satisfies

$$a_{10}\equiv\sum_{k=1}^{9} ka_k \pmod{11}.$$

The ISBN 0-13-186239-? is missing the check digit (printed as "?"). Compute it!

Solution. $1\cdot 0+2\cdot 1+3\cdot 3+4\cdot 1+5\cdot 8+6\cdot 6+7\cdot 2+8\cdot 3+9\cdot 9=210\equiv 1 \pmod{11}$

Hence, the full ISBN is 0-13-186239-1.

Comment. The check digit is designed so that it is always possible to detect when a single digit is messed up. It is also always possible to detect when two digits are transposed.

This is another example of **error checking**, which is standard practice for all sorts of identification numbers (such as bank account numbers, VIN, ...). With a little more effort **error correction** is also possible.

Euler's phi function

Definition 16. Euler's phi function $\phi(n)$ gives the number of integers in $\{1, 2, ..., n\}$ that are relatively prime to n.

In other words, $\phi(n)$ counts how many residues are invertible modulo n.

Example 17. Compute $\phi(n)$ for $n = 1, 2, ..., 8$.

Solution. $\phi(1) = 1$, $\phi(2) = 1$, $\phi(3) = 2$, $\phi(4) = 2$, $\phi(5) = 4$, $\phi(6) = 2$, $\phi(7) = 6$, $\phi(8) = 4$.

Observation. $\phi(n) = n - 1$ if and only if n is a prime.
This is true because $\phi(n) = n - 1$ if and only if n is coprime to all of $\{1, 2, ..., n-1\}$.

Observation. If p is a prime, then $\phi(p^k) = p^k - p^{k-1} = p^k\left(1 - \frac{1}{p}\right)$.

This is true because, if p is a prime, then $n = p^k$ is coprime to all $\{1, 2, ..., p^k\}$ except $p, 2p, 3p, ..., p^k$ (the multiples of p, of which there are $p^k / p = p^{k-1}$ many).

If the prime factorization of n is $n = p_1^{k_1} \cdots p_r^{k_r}$, then $\phi(n) = n\left(1 - \frac{1}{p_1}\right) \cdots \left(1 - \frac{1}{p_r}\right)$.

Why is this true?

- We observed above that the formula is true if $n = p^k$ is a prime power.
- On the other hand, for composite n, say $n = ab$, we have: $\boxed{\phi(ab) = \phi(a)\phi(b) \text{ if } \gcd(a, b) = 1}$
  This is a consequence of the Chinese remainder theorem. (Review if necessary! We'll use it later but will only review it briefly then.)

The above formula follows from combining these two observations. Can you fill in the details?

Example 18. Compute $\phi(35)$.

Solution. $\phi(35) = \phi(5 \cdot 7) = \phi(5)\phi(7) = 4 \cdot 6 = 24$

Example 19. Compute $\phi(100)$.

Solution. $\phi(100) = \phi(2^2 \cdot 5^2) = \phi(2^2)\phi(5^2) = (2^2 - 2^1) \cdot (5^2 - 5^1) = 40$

[Alternatively: $\phi(100) = \phi(2^2 \cdot 5^2) = 100\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{5}\right) = 40$]

Example 20. Compute $\phi(1000)$.

Solution. $\phi(1000) = \phi(2^3) \cdot \phi(5^3) = (8 - 4)(125 - 25) = 400$

[Alternatively: $\phi(1000) = \phi(2^3 \cdot 5^3) = 1000\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{5}\right) = 400$.]

Armin Straub
straub@southalabama.edu

Historical examples of symmetric encryption

Alice wants to send a secret message to Bob.

What Alice sends will be transmitted through an unsecure medium (like the internet), meaning that others can read it. However, it is important to Alice and Bob that no one else can understand it.

The original message is referred to as the **plaintext** m. What Alice actually sends is called the **ciphertext** c (the encrypted message).

Symmetric encryption algorithms rely on a secret key k (from some **key space**) shared by Alice and Bob (but unknown to anyone else).

Our ultimate goal will be to secure messaging against both:

- eavesdropping (goal: **confidentiality**)
- tampering (goal: **integrity** and, even stronger, **authenticity**)

  The symmetric encryption approach, by itself, cannot fully protect against tampering. For instance, an attacker can collect previously sent messages, resend them, or use them to replace new messages. (You could preface each message with something like a time stamp to address these issues. But that's getting ahead of ourselves; and there are better ways.)

Shift cipher

The alphabet for our messages will be $A, B, ..., Z$, which we will identify with $0, 1, ..., 25$.

So, for instance, C is identified with the number 2.

Example 21. (shift cipher) A key is an integer $k \in \{0, 1, ..., 25\}$. Encryption works character by character using

$$E_k: \quad x \mapsto x + k \pmod{26}.$$

Obviously, the decryption D_k works as $x \mapsto x - k \pmod{26}$.

The **key space** is $\{0, 1, ..., 25\}$. It has size 26. [Well, $k = 0$ is a terrible key. Maybe we should exclude it.]

For instance. If $k = 1$, then the message $HELLO$ is encrypted as $IFMMP$.

If $k = 2$, then the message $HELLO$ is encrypted as $JGNNQ$.

Historic comment. Caesar encrypted some private messages with a shift cipher (typically using $k = 3$). The shift cipher is therefore also often called Caesar's cipher.

While completely insecure today, it was fairly secure at the time (with many of his enemies being illiterate).

Modern comment. Many message boards on the internet "encrypt" things like spoilers or solutions using a shift cipher with $k = 13$. This is called ROT13. What's special about the choice $k = 13$?

Solution. Since $-13 \equiv 13 \pmod{26}$, for ROT13, encryption and decryption are the same!

Armin Straub
straub@southalabama.edu

Example 22. (affine cipher) A slight upgrade to the shift cipher, we encrypt each character as

$$E_{(a,b)}: \quad x \mapsto ax + b \pmod{26}.$$

How does the decryption work? How large is the key space?

Solution. Each character x is decrypted via $x \mapsto a^{-1}(x-b) \pmod{26}$.
The key is $k = (a, b)$. Since a has to be invertible modulo 26, there are $\phi(26) = \phi(2) \cdot \phi(13) = 12$ possibilities for a. There are 26 possibilities for b. Hence, the key space has size $12 \cdot 26 = 312$.

Vigenere cipher (vector shift cipher)

See Section 2.3 of our book for a full description of the Vigenere cipher.

This cipher was long believed by many (until early 20th) to be secure against ciphertext only attacks (more on the classification of attacks shortly).

Example 23. Let us encrypt $HOLIDAY$ using a Vigenere cipher with key BAD (i.e. $1, 0, 3$).

	H	O	L	I	D	A	Y
$+$	B	A	D	B	A	D	B
$=$	I	O	O	J	D	D	Z

Hence, the ciphertext is $IOOJDDZ$.

An encrypted message

Example 24. (bonus challenge!) You find a post-it with the following message:

$$ZHOFRPH \quad WR \quad FUBSWR$$

Can you make any sense of it?

(To collect a bonus point, send me an email before next class with the plaintext and how you found it.)

Example 25. The challenge from Example 24 was encrypted using .... The key space has size ..., so a brute-force attack results in immediate success: we find that the plaintext is ...

This is the worst kind of vulnerability: we successfully mounted a **ciphertext only attack**.

That is, just knowing the encrypted message, we were able to decrypt it (and discover the key that was used).

Example 26. (warmup) What a terrible blunder... Explain what is wrong!

$$\text{(incorrect!)} \quad 10^9 \equiv 3^2 = 9 \equiv 2 \pmod 7$$

Solution. $10^9 = 10 \cdot 10 \cdot \ldots \cdot 10 \equiv 3 \cdot 3 \cdot \ldots \cdot 3 = 3^9$. Hence, $10^9 \equiv 3^9 \pmod 7$.

However, there is no reason, why we should be allowed to reduce the exponent by 7 (and it is incorrect).

Corrected calculation. $3^2 \equiv 2$, $3^4 \equiv 4$, $3^8 \equiv 16 \equiv 2$. Hence, $3^9 = 3^8 \cdot 3^1 \equiv 2 \cdot 3 \equiv -1 \pmod 7$.

By the way, this approach of computing powers via exponents that are $2,\ 4,\ 8,\ 16,\ 32,\ \ldots$ is called **binary exponentiation**. It is crucial for efficiently computing large powers.

Corrected calculation (using Fermat). $3^6 \equiv 1$ just like $3^0 = 1$. Hence, we are allowed to reduce exponents modulo 6. Hence, $3^9 \equiv 3^3 \equiv -1 \pmod 7$.

Theorem 27. (Fermat's little theorem) Let p be a prime, and suppose that $p \nmid a$. Then

$$a^{p-1} \equiv 1 \pmod p.$$

Proof. (beautiful!) Since a is invertible modulo p, the first $p-1$ multiples of a,

$$a, 2a, 3a, \ldots, (p-1)a$$

are all different modulo p. Clearly, none of them is divisible by p.

Consequently, these values must be congruent (in some order) to the values $1, 2, \ldots, p-1$ modulo p. Thus,

$$a \cdot 2a \cdot 3a \cdot \ldots \cdot (p-1)a \equiv 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1) \pmod p.$$

Cancelling the common factors (allowed because p is prime!), we get $a^{p-1} \equiv 1 \pmod p$. □

Remark. The "little" in this theorem's name is to distinguish this result from Fermat's last theorem that $x^n + y^n = z^n$ has no integer solutions if $n > 2$ (only recently proved by Wiles).

Example 28. (bonus challenge!) Eve, can you crack the following message?

$$OIWW \;\; PIHX \;\; RR \;\; PSQHDC$$

Word on the street is that Alice was using the Vigenere cipher with a key of size 2.

(To collect a bonus point, send me an email before next class with the plaintext and how you found it.)

Attacks

So far, we considered the weakest kind of attack only: namely, a **ciphertext only attack**. And, even then, the historical ciphers prove to be terribly insecure.

However, we need to also worry about attacks where our enemy has additional insight.

- In a **known plaintext attack**, the enemy somehow has knowledge of a plaintext-ciphertext pair (m, c).
- In a **chosen plaintext attack**, the enemy can, herself, compute $c = E(m)$ for a chosen plaintext m ("gained some sort of access to our encryption device").
- In a **chosen ciphertext attack**, the enemy can, herself, compute $m = D(c)$ for a chosen ciphertext c ("gained some sort of access to our decryption device").

There exist many variations of these. Sometimes, the attacker can make several choices (maybe even adaptively), sometimes she only has partial information.

Example 29. Alice sends the ciphertext $BKNDKGBQ$ to Bob. Somehow, Eve has learned that Alice is using the Vigenere cipher and that the plaintext is $ALLCLEAR$. Next day, Alice sends the message $DNFFQGE$. Crack it and figure out the key that Alice used! (What kind of attack is this?)

Solution. This is a known plaintext attack.

Since $m + k = c$ (to be interpreted characterwise, modulo 26, and with k repeated as necessary), we can find k simply as $k = c - m$.

For instance, since A (value 0!) got encrypted to B, the first letter of the key is B.

c			B	K	N	D	K	G	B	Q
m		$-$	A	L	L	C	L	E	A	R
k		$=$	B	Z	C	B	Z	C	B	Z

We conclude that the key is $k = BZC$.

Note. Now, we can decrypt any future message that Alice sends using this key. For instance, we easily decrypt $DNFFQGE$ to $CODERED$ (using $m = c - k$).

All of the historical ciphers we have seen, including the substitution cipher that we will discuss shortly, fall apart completely under a known plaintext attack.

Euler's theorem

Example 30. Compute $3^{1003} \pmod{101}$.

Solution. Since 101 is a prime, $3^{100} \equiv 1 \pmod{101}$ by Fermat's little theorem.
Because $3^{100} \equiv 3^0 \pmod{101}$, this enables us to reduce exponents modulo 100.
In particular, since $1003 \equiv 3 \pmod{100}$, we have $3^{1003} \equiv 3^3 = 27 \pmod{101}$.

Fermat's little theorem is a special case of Euler's theorem :

Theorem 31. (Euler's theorem) If $n \geqslant 1$ and $\gcd(a,n) = 1$, then $a^{\phi(n)} \equiv 1 \pmod{n}$.

Proof. Euler's theorem can be proved along the lines of our earlier proof of Fermat's little theorem. The only adjustment is to only start with multiples ka where k is invertible modulo n. There are $\phi(n)$ such residues k, and so that's where Euler's phi function comes in. Can you complete the proof? □

Example 32. What are the last two (decimal) digits of 3^{7082}?

Solution. We need to determine $3^{7082} \pmod{100}$. $\phi(100) = \phi(2^2 5^2) = \phi(2^2)\phi(5^2) = (2^2 - 2^1)(5^2 - 5^1) = 40$.
Since $\gcd(3,100) = 1$ and $7082 \equiv 2 \pmod{40}$, Euler's theorem shows that $3^{7082} \equiv 3^2 = 9 \pmod{100}$.

Binary exponentiation

Example 33. Compute $3^{25} \pmod{101}$.

Solution. Fermat's little theorem is not helpful here.
Instead, we do **binary exponentiation**:
$3^2 = 9$, $3^4 = 81 \equiv -20$, $3^8 \equiv (-20)^2 = 400 \equiv -4$, $3^{16} \equiv (-4)^2 \equiv 16$, all modulo 101
$25 = 16 + 8 + 1$ [Every integer $n \geqslant 0$ can be written as a sum of distinct powers of 2 (in a unique way).]
Hence, $3^{25} = 3^{16} \cdot 3^8 \cdot 3^1 \equiv 16 \cdot (-4) \cdot 3 = -192 \equiv 10 \pmod{101}$.

Example 34. (extra practice) Compute $2^{20} \pmod{41}$.

Solution. $2^2 = 4$, $2^4 = 16$, $2^8 = 256 \equiv 10$, $2^{16} \equiv 100 \equiv 18$. Hence, $2^{20} = 2^{16} \cdot 2^4 \equiv 18 \cdot 16 = 288 \equiv 1 \pmod{41}$.
Or: $2^5 = 32 \equiv -9 \pmod{41}$. Hence, $2^{20} = (2^5)^4 \equiv (-9)^4 = 81^2 \equiv (-1)^2 = 1 \pmod{41}$.
Comment. Write $a = 2^{20} \pmod{41}$. It follows from Fermat's little theorem that $a^2 = 2^{40} \equiv 1 \pmod{41}$. The argument below shows that $a \equiv \pm 1 \pmod{41}$ [but we don't know which until we do the calculation].
The equation $x^2 \equiv 1 \pmod{p}$ is equivalent to $(x-1)(x+1) \equiv 0 \pmod{p}$ [b/c $(x-1)(x+1) = x^2 - 1$]. Since p is a prime and $p \mid (x-1)(x+1)$, we must have $p \mid (x-1)$ or $p \mid (x+1)$. In other words, $x \equiv \pm 1 \pmod{p}$.

Armin Straub
straub@southalabama.edu

Representations of integers in different bases

We are commonly using the **decimal system** of writing numbers. For instance:

$$1234 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0.$$

10 is called the base, and $1, 2, 3, 4$ are the digits in base 10. To emphasize that we are using base 10, we will write $1234 = (1234)_{10}$. Likewise, we write

$$(1234)_b = 1 \cdot b^3 + 2 \cdot b^2 + 3 \cdot b^1 + 4 \cdot b^0.$$

In this example, $b > 4$, because, if b is the base, then the digits have to be in $\{0, 1, ..., b-1\}$.

Comment. In the above examples, it is somewhat ambiguous to say whether 1 or 4 is the first or last digit. To avoid confusion, one refers to 4 as the **least significant digit** and 1 as the **most significant digit**.

Example 35. $25 = 16 + 8 + 1 = \boxed{1} \cdot 2^4 + \boxed{1} \cdot 2^3 + \boxed{0} \cdot 2^2 + \boxed{0} \cdot 2^1 + \boxed{1} \cdot 2^0$.
Accordingly, $25 = (11001)_2$.

While the approach of the previous example works well for small examples when working by hand (if we are comfortable with powers of 2), the next example illustrates a more algorithmic approach.

Example 36. Express 49 in base 2.

Solution.

- $49 = 24 \cdot 2 + \boxed{1}$. Hence, $49 = (...1)_2$ where ... are the digits for 24.
- $24 = 12 \cdot 2 + \boxed{0}$. Hence, $49 = (...01)_2$ where ... are the digits for 12.
- $12 = 6 \cdot 2 + \boxed{0}$. Hence, $49 = (...001)_2$ where ... are the digits for 6.
- $6 = 3 \cdot 2 + \boxed{0}$. Hence, $49 = (...0001)_2$ where ... are the digits for 3.
- $3 = 1 \cdot 2 + \boxed{1}$. Hence, $49 = (...10001)_2$ where ... are the digits for 1.
- $1 = 0 \cdot 2 + \boxed{1}$. Hence, $49 = (110001)_2$.

Other bases.
What is 49 in base 3? $49 = 16 \cdot 3 + \boxed{1}$, $16 = 5 \cdot 3 + \boxed{1}$, $5 = 1 \cdot 3 + \boxed{2}$, $1 = 0 \cdot 3 + \boxed{1}$. Hence, $49 = (1211)_3$.
What is 49 in base 5? $49 = (144)_5$.
What is 49 in base 7? $49 = (100)_7$.

Example 37. Bases 2, 8 and 16 (binary, octal and hexadecimal) are commonly used in computer applications.

For instance, in JavaScript or Python, `0b...` means $(...)_2$, `0o...` means $(...)_8$, and `0x...` means $(...)_{16}$.
The digits $0, 1, ..., 15$ in hexadecimal are typically written as $0, 1, ..., 9, A, B, C, D, E, F$.
Example. FACE value in decimal? $(FACE)_{16} = 15 \cdot 16^3 + 10 \cdot 16^2 + 12 \cdot 16 + 14 = 64206$
Practical example. `chmod 664 file.tex` (change file permission)

664 are octal digits, consisting of three bits: $1 = (001)_2$ execute (x), $2 = (010)_2$ write (w), $4 = (100)_2$ read (r)
Hence, 664 means rw,rw,r. What is rwx,rx,-? 750
By the way, a fourth (leading) digit can be specified (setting the flags: setuid, setgid, and sticky).

Armin Straub
straub@southalabama.edu

Example 38. (substitution cipher) In a substitution cipher, the key k is some permutation of the letters $A, B, \ldots, Z$. For instance, $k = FRA\ldots$. Then we encrypt $A \to F$, $B \to R$, $C \to A$ and so on. How large is the key space?

Solution. Key space has size $26! \approx 10^{26.6} \approx 2^{88.4}$, so a key can be stored using 89 bits. That's actually a fairly large key space (for instance, DES has a key size of 56 bits only). Too large to go through by brute force.

However, still easy to break. Since each letter is always replaced with the same letter, this cipher is susceptible to a **frequency attack**, exploiting that certain letters (and, more generally, letter combinations!) occur much more frequently in, say, English text than others. For instance, Lewand's book on Cryptology lists the following frequencies:

E: 12.7%, T: 9.1%, A: 8.2%, O: 7.5%, I: 7%, N: 6.7%, S: 6.3%, H: 6.1%, R: 6%, D: 4.3%, L: 4%, C: 2.8%, ...

The rarest letters are Q and Z with a frequency of about 0.1% only. (The exact frequencies and precise ordering various between different sources and the body of text that the frequencies were obtained from.)

The most common letter pairs (digrams) are TH HE AN RE ER IN ON AT ND ST ES EN OF TE ED OR TI HI AS TO.

More information at: https://en.wikipedia.org/wiki/Letter_frequency

Comment. Note that the frequencies and even the ranking depend considerably on the source of text. For instance, using government telegrams, a military resource lists EN followed by RE, ER as the most frequent digrams. That same manual suggests SENORITA as a mnemonic to remember the most frequent letters.

http://www.umich.edu/~umich/fm-34-40-2/ (Field Manual 34-40-2, Department of the Army, 1990)

Example 39. It seems convenient to add the space as a 27th letter in the historic encryption schemes. Can you think of a reason against doing that?

Solution. In most texts, the space occurs more frequently and more regularly than any other letter. Adding it to the encryption schemes would make them even more susceptible to attacks.

Example 40. (bonus challenge!) You intercept the following message from Alice:

WHCUHFWXOWHUQXOMOMQVSQWAMWHCUHFXOLNWXQMQVSQWAWMQLN

Your experience tells you that Alice is using a substitution cipher. You also know that this message contains the word "secret". Can you crack it?

Note. In modern practice, it is not uncommon to know (or suspect) what a certain part of the message should be. For instance, PDF files start with "%PDF" (0x25504446).

See https://en.wikipedia.org/wiki/Magic_number_(programming) for more such instances.

(To collect a bonus point, send me an email within the next week with the plaintext and how you found it.)

Modern ciphers

Example 41. For modern ciphers, we will change the alphabet from $A, B, \ldots, Z$ to $0, 1$. One of the most common ways of encoding text is **ASCII**.

In ASCII (American Standard Code for Information Interchange), each letter is represented using 8 bits (1 byte). Among the $2^8 = 256$ many characters are the usual letters, as well as common symbols.

For instance: space $= (20)_{16}$, "0"$=(30)_{16}$, $A = (41)_{16} = (0100, 0001)_2 = 65$, $a = (61)_{16} = (0110, 0001)_2 = 97$

See, for instance, http://www.asciitable.com for the full table.

Example 42. The new (8/2018) insignia of **FinCEN** features binary digits. What do they mean?

01000110 01101001 01101110 01000011 01000101 01001110 https://www.fincen.gov

By the way. If you ever have more than $\$10,000$ in foreign accounts, you must file a report to FinCEN.

One-time pad

Definition 43. The "exclusive or" (XOR), often written $\oplus$, is defined bitwise:

	0	0	1	1
$\oplus$	0	1	0	1
$=$	0	1	1	0

Note. On the level of individual bits, this is just addition modulo 2.

By the way. Best thing about a boolean: even if you are wrong, you are only off by a bit.

Example 44. $1011 \oplus 1111 = 0100$

Example 45. Observe that $a \oplus b \oplus b = a$.

One way to see that is to think bitwise in terms of addition modulo 2. Then, $a + b + b = a + 2b \equiv a \pmod 2$.

A **one-time pad** works as follows. We use a key k of the same length as the message m. Then the ciphertext is

$$c = E_k(m) = m \oplus k.$$

To decipher, we use $m = D_k(c) = c \oplus k$.

As the name indicates, we must never use this key again!

Note. Observe that encryption and decryption are the same routine.

Comment. If that is helpful, a one-time pad is doing exactly the same as the Vigenere cipher if we use a key of the same length as the message (also, we use $0, 1$ as our letters instead of the classical $A, B, ..., Z$).

Example 46. Using a one-time pad with key $k = 1100, 0011$, what is the message $m = 1010, 1010$ encrypted to?

Solution. $c = m \oplus k = 0110, 1001$

If a one-time pad (with perfectly random key) is used exactly once to encrypt a message, then **perfect confidentiality** is achieved (eavesdropping is hopeless).

Meaning that Eve intercepting the ciphertext can draw absolutely no conclusions about the plaintext (because, without information on the key, every text of the right length is actually possible and equally likely), see next example.

Example 47. A ciphertext only attack on the one-time pad is entirely hopeless. Explain why!

Solution. The attacker only knows $c = m \oplus k$. The attacker is unable to get any information on m, because every other message m' (of the right length) could have resulted in the same ciphertext c.

Indeed, the key $k' = m' \oplus c$ encrypts m' to c as well (because $m' \oplus k' = m' \oplus (m' \oplus c) = c$). Moreover, every plaintext m' is equally likely because it corresponds to a unique key.

The next example highlights the importance of only using the key once.

Example 48. (attack on the two-time pad) Alice made a mistake and encrypted the two plaintexts m_1, m_2 using the same key k. How can Eve exploit that?

Solution. Eve knows the two ciphertexts $c_1 = m_1 \oplus k$ and $c_2 = m_2 \oplus k$.

Hence, she can compute $c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2$.

This means that Eve knows $m_1 \oplus m_2$, which is information about the original plaintexts (no key involved!). That's a cryptographic disaster: Eve should never be able to learn *anything* about the plaintexts.

In fact. If the plaintexts are, say, English text encoded using ASCII then Eve very possibly can (almost) reconstruct both m_1 and m_2 from $m_1 \oplus m_2$. The reason for that is that the messages are expressed in ASCII, which means 8 bits per character of text. However, the **entropy** (a measure for the amount of information in a message) of (longer) typical English text is frequently below 2 bits per character.

Some details and beautiful graphical illustrations are given at:

http://crypto.stackexchange.com/questions/59/taking-advantage-of-one-time-pad-key-reuse

We saw in Example 47 that ciphertext only attacks on the one-time pad are entirely hopeless. What about other attacks?

Attacks like known plaintext or chosen plaintext don't apply if the key is only to be used once.

Yet, the one-time pad by itself provides **little protection of integrity**. The next example shows how tampering is possible without knowledge about the key.

Example 49. Alice sends an email to Bob using a one-time pad. Eve knows that and concludes that, per email standard, the plaintext must begin with `To: Bob`. Eve wants to tamper with the message and change it to `To: Boo`, for a light scare.

- Eve wants to change the 7th letter of the plain text m from b to o.
- Since b is $0x62$ and o is $0x6F$, we have $b \oplus o = 0x0D$. Hence, $b \oplus 0x0D = o$.
- Therefore, if $e = 0x\underbrace{000000000000}_{6 \text{ characters}}0D00\ldots$, then $\underbrace{\text{“TO: Bob...”}}_{m} \oplus e = \underbrace{\text{“TO: Boo...”}}_{m'}$.
- Alice sends $c = m \oplus k$. If Eve changes the ciphertext c to $c' = c \oplus e$, then Bob receives c' and decrypts it to $c' \oplus k = \overbrace{\underbrace{m \oplus k}_{=c} \oplus e}^{c'} \oplus k = m \oplus e = m'$, which is what Eve intended.

Using the one-time pad presents several challenges, including:

- keys must not be reused (see Example 48)
- while perfectly protecting against eavesdropping (if done correctly), the one-time pad is not secure against tampering (see Example 49)
- key distribution and management

  Alice and Bob have to somehow exchange huge amounts of keys, so that, at a later time, they are able to communicate securely.
- for perfect confidentiality, the key must be perfectly random

  But how can we produce huge amounts of random bits?

  Especially, how to teach a deterministic machine like a computer to do that? Think about it! This is much more challenging that it may seem at first...

These issues make one-time pads difficult to use in practice.

Historic comment. During the Cold War, the “hot line” between Washington and Moscow apparently used one-time pads for secure communication.

Example 50. 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 48).

So, repeating the key is a terrible idea. However, the idea to create a longer (random) key out of a shorter (random) key is good (we will discuss pseudorandom generators 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.

Remark. A nonce can only be used once, as is in its name. Apparently, according to Urban Dictionary, it is also common as a British insult, roughly equivalent to wanker.

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.

Comment. The particular choices of a and b in glibc are somewhat mysterious. See, for instance:

https://stackoverflow.com/questions/8569113/why-1103515245-is-used-in-rand

Example 51. 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 52. (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 53. 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 are 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 54. 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 55. 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 $m = c \oplus \mathrm{PRG} = (111\ 111\ 111)_2 \oplus (001\ 000\ 011)_2 = (110\ 111\ 100)_2$.

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 "look like" 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 59.)

- 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 57), and hence suited for applications that value speed over security (think, for instance, encrypted television).

Example 56. 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 are $2^2 = 4$ states of the LFSR. The state $(0, 0)$ is special (it generates the zero sequence $(0, 0, 0, 0, \ldots)$), so there are 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 $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

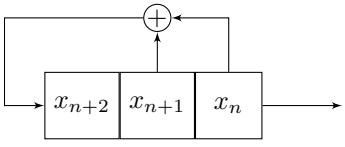
Example 57. Which sequence is generated by the LFSR $x_{n+3} \equiv x_{n+1} + x_n \pmod 2$, starting with the seed $(x_1, x_2, x_3) = (0, 0, 1)$? What is the period?

[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 are $2^3 = 8$ states. The state $(0, 0, 0)$ is special, so there are 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 58.

Armin Straub
straub@southalabama.edu

Example 58. 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 $(0,0,\ldots,0)$ is special (the entire sequence will be zero), so that there are $2^{31}-1$ 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 $f(T) = x^\ell + c_1 x^{\ell-1} + c_2 x^{\ell-2} + \ldots + c_\ell$ 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 $2^{31}-1$.

Example 59. Eve intercepts the ciphertext $c = (1111\ 1011\ 0000)_2$ from Alice to Bob. She knows that the plaintext begins with $m = (1100\ 0\ldots)_2$. Eve thinks a stream cipher using a LFSR with $x_{n+3} \equiv x_{n+2} + x_n \pmod 2$ was used. If that's the case, what is the plaintext?

Solution. The initial piece of the keystream is $\mathrm{PRG} = m \oplus c = (1100\ 0\ldots)_2 \oplus (1111\ 1\ldots)_2 = (0011\ 1\ldots)_2$.

Each x_n is a single bit, and we have $x_1 \equiv 0$, $x_2 \equiv 0$, $x_3 \equiv 1$. The given LFSR produces $x_4 \equiv x_3 + x_1 \equiv 1$, $x_5 \equiv x_4 + x_2 \equiv 1$, $x_6 \equiv 0$, $x_7 \equiv 1$, and so on. Continuing, we obtain $\mathrm{PRG} = x_1 x_2 \ldots = (0011\ 1010\ 0111)_2$.

Hence, the plaintext would be $m = c \oplus \mathrm{PRG} = (1111\ 1011\ 0000)_2 \oplus (0011\ 1010\ 0111)_2 = (1100\ 0001\ 0111)_2$.

A PRG is **predictable** if, given the stream it outputs (but not the seed), one can with some precision predict what the next bit will be (i.e. do better than just guessing the next bit).

In other words: the bits generated by the PRG must be indistinguishable from truly random bits, even in the eyes of someone who knows everything about the PRG except the seed.

The PRGs we discussed so far are entirely predictable because the state of the PRGs is part of the random stream they output.

For instance, for a given LFSR, it is enough to know any ℓ consecutive outputs $x_n, x_{n+1}, \ldots, x_{n+\ell-1}$ in order to predict all future output.

We have seen two simple examples of PRGs so far:

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

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 57 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 $\overbrace{0, 0, 1}^{\text{seed}}, 0, 1, 1, 1, 0, 1, \ldots$ 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).

Example 61. Suppose we have two PRGs that output bits. The first repeats after 14 bits, the second after 18 bits. After how many bits do they repeat simultaneously?

What if the two PRGs repeat after 13 and 17 bits instead?

Solution. Note that the first PRG again repeats after 28 bits, after 42 bits and, in general after $14m$ bits where m is any positive integer. Likewise, the second PRG repeats after $18m$ bits where m is any positive integer. Therefore, both PRGs repeat simultaneously after $\operatorname{lcm}(14, 18) = \frac{14 \cdot 18}{2} = 126$ bits.

Review. Here, lcm is the **least common multiple**. We can always compute the lcm through the Euclidean algorithm by using $\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$.

If the two PRGs repeat after 13 and 17 bits instead, then they repeat simultaneously after $\operatorname{lcm}(13, 17) = 13 \cdot 17 = 221$ bits.

Comment. Certain cicadas spend more than 99% of their life underground as nymphs and only emerge as adults for 4-6 weeks. Interestingly, this life cycle is highly synchronized: cicadas of one species in a region appear all at once. In 2024, "Brood XIII" and "Brood XIX" will co-emerge. These emerge every 17 and 13 years, respectively. Therefore this co-emergence is a rare event that only happens every 221 years (though, the same thing happened 2015 with two different broods).

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

Example 62. (bonus!) Eventually the output of the baby CSS in Example 63 has to repeat (though it need not be perfectly periodic; see Example 60). Once it repeats, what is the period?

Note. The state of the system is determined by $3 + 4 + 1 = 8$ bits (3 bits for LFSR-1, 4 bits for LFSR-2, and 1 bit for the carry). Hence, there are $2^8 = 256$ many states. Since the state with everything 0 is again special, that means that after at most 255 steps our PRG will reach a state it has been in before. At that point, everything will repeat.

(To collect a bonus point, send me an email within the next week with the period and how you found it.)

Combining two LFSRs to get the CSS (content scramble system)

A popular way to reduce predictability is to combine several LFSRs (in a nonlinear fashion):

Example 63. The CSS (content scramble system) is based on 2 LFSRs and used for the encryption of DVDs. Before discussing the actual CSS let us consider a baby version of CSS. Our PRG uses the LFSR $x_{n+3} \equiv x_{n+1} + x_n \pmod 2$ as well as the LFSR $x_{n+4} \equiv x_{n+2} + x_n \pmod 2$. The output of the PRG is the output of these two LFSRs added with carry.

Adding with carry just means that we are adding bits modulo 2 but add an extra 1 to the next bits if the sum exceeded 1. This is the same as interpreting the output of each LFSR as the binary representation of a (huge) number, then adding these two numbers, and outputting the binary representation of the sum.

Armin Straub
straub@southalabama.edu

If we use $(0,0,1)$ as the seed for LFSR-1, and $(0,1,0,1)$ for LFSR-2, what are the first 10 bits output by our PRG?

Solution. With seed $0,0,1$ LSFR-1 produces $0,1,1,1,0,0,1,0,1,1,\ldots$
With seed $0,1,0,1$ LSFR-2 produces $0,0,0,1,0,1,0,0,0,1,\ldots$
We now add these two:

	0	1	1	1	0	0	1	0	1	1	⋯
+	0	0	0	1	0	1	0	0	0	1	⋯
carry					1						1
	0	1	1	0	1	1	1	0	1	0	⋯

Hence, the output of our PRG is $0,1,1,0,1,1,1,0,1,0,\ldots$.

Important comment. Make sure you realize in which way this CSS PRG is much less predictable than a single LFSR! A single LFSR with ℓ registers is completely predictable since knowing ℓ bits of output (determines the state of the LFSR and) allows us to predict all future output. On the other hand, it is not so simple to deduce the state of the CSS PRG from the output. For instance, the initial $(0,1,\ldots)$ output could have been generated as $(0,0,\ldots)+(0,1,\ldots)$ or $(0,1,\ldots)+(0,0,\ldots)$ or $(1,0,\ldots)+(1,0,\ldots)$ or $(1,1,\ldots)+(1,1,\ldots)$.
[In this case, we actually don't learn anything about the registers of each individual LFSR. However, we do learn how their values have to match up. That's the correlation that is exploited in **correlation attacks**, like the one described next class for the actual CSS scheme.]

Advanced comment. Is the carry important? Yes! Let $a_1,a_2,\ldots$ and $b_1,b_2,\ldots$ be the outputs of LFSR-1 and LFSR-2. Suppose we sum without carry. Then the output is $a_1+b_1,a_2+b_2,\ldots$ (with addition mod 2). If Eve assigns variables k_1, k_2, ..., k_7 to the $3+4$ seed bits (the key in the stream cipher), then the output of the combined LFSR will be linear in these seven variables (because the a_i and b_i are linear combinations of the k_i). Given just a few more than 7 output bits, a little bit of linear algebra (mod 2) is therefore enough to solve for $k_1,k_2,\ldots,k_7$.
On the other hand, suppose we include the carry. Then the output is $a_1+b_1,a_2+b_2+a_1b_1,\ldots$ (note how a_1b_1 is $1 \pmod 2$ precisely if both a_1 and b_1 are $1 \pmod 2$, which is when we have a carry). This is not linear in the a_i and b_i (and, hence, not linear in the k_i), and we cannot use linear algebra to solve for $k_1,k_2,\ldots,k_7$ as before.

Example 64. In each case, determine if the stream could have been produced by the LFSR $x_{n+5} \equiv x_{n+2}+x_n \pmod 2$. If yes, predict the next three terms.
(STREAM-1) $\ldots,1,0,0,1,1,1,1,0,1,\ldots$ (STREAM-2) $\ldots,1,1,0,0,0,1,1,0,1,\ldots$

Solution. Using the LFSR, the values $1,0,0,1,1$ are followed by $1,1,1,0,\ldots$ Hence, STREAM-1 was not produced by this LFSR.
On the other hand, using the LFSR, the values $1,1,0,0,0$ are followed by $1,1,0,1,1,1,0,\ldots$ Hence, it is possible that STREAM-2 was produced by the LFSR (for a random stream, the chance is only $1/2^4 = 6.25\%$ that 4 bits matched up). We predict that the next values are $1,1,0,\ldots$
Comment. This observation is crucial for the attack on CSS described in Example 65.

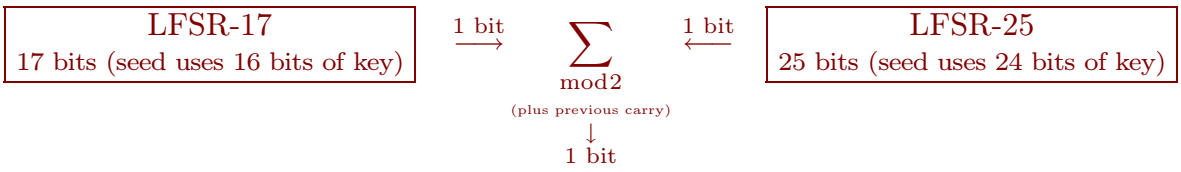
Example 65. (CSS) The CSS (content scramble system) is based on 2 LFSRs and used for the encryption of DVDs. Let us indicate (in a slightly oversimplified way) how to break it.

CSS was introduced in 1996 and first compromised in 1999. One big issue is that its key size is 40 bits. Since $2^{40} \approx 1.1 \cdot 10^{12}$ is small by modern standards, even a direct brute-force attack in time 2^{40} is possible.

However, we will see below that poor design makes it possible to attack it in time 2^{16}.

Historic comment. 40 bits was the maximum allowed by US export limitations at the time.

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



CSS PRG combines one 17-bit LFSR and one 25-bit LFSR. The bits output by the CSS PRG are the sum of the bits output by the two LFSRs (this is the usual sum, including carries).

The 40 bit key is used to seed the LFSRs (the 4th bit of each seed is "1", so we need $16+24=40$ other bits).

Here's how we break CSS in time 2^{16}:

- If a movie is encrypted using MPEG then we know the first few, say x (6-20), bytes of the plaintext.
- As in Example 59, this allows us to compute the first x bytes of the CSS keystream.
- We now go through all 2^{16} possibilities for the seed of LFSR-17. For each seed:
  - We generate x bytes using LFSR-17 and subtract these from the known CSS keystream.
  - This would be the output of LFSR-25. As in Example 64, we can actually easily tell if such an output could have been produced by LFSR-25. If yes, then we found (most likely) the correct seed of LFSR-17 and now also have the correct state of LFSR-25.

This kind of attack is known as a correlation attack.

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

Comment. Similar combinations of LFSRs are used in GSM encryption (A5/1,2, 3 LFSRs); Bluetooth (E0, 4 LFSRs). Due to certain details, these are broken or have serious weaknesses; so, of course, they shouldn't be used. However, it is difficult to update things implemented in hachallengerdware...

Sad but important lessons

Review. CSS (content scramble system) is based on 2 LFSRs whose outputs are added with carry (the carry is important because it combines the LFSRs in a nonlinear way).

Combining LFSRs in a nonlinear fashion is a good idea for constructing PRGs for cryptographic purposes (especially because they are simple to implement in hardware). However, as the examples of CSS as well as GSM/Bluetooth encryption show, a lot of attention has to be paid to the details in order not to compromise security.

CSS (and many other examples in recent history) teach us one important lesson:

Do not implement your own ideas for serious crypto!

We will soon see that there exist cryptosystems which are believed to be secure. While none of these beliefs are proven, we do know that certain of these are in fact secure (if implemented correctly) if and only if a certain important mathematical problem cannot be easily solved.

- So, to crack such a system, one has to solve a mathematical problem that many people care about deeply. If this happens, you will most likely read about it in the (academic) news, and you will have an opportunity to update your system in time (most likely, you'll hear about progress much earlier).
- On the other hand, if you use a cryptosystem that is not well-studied, then it may well happen that an adversary breaks your system and keeps exploiting the security leak without you ever learning about it.

Not particularly related but important to keep in mind:

Frequently, security's weakest link are humans. It's very hard to protect against that.
https://en.wikipedia.org/wiki/Social_engineering_(security)

Review: Chinese remainder theorem

Example 66. (warmup)

(a) If $x \equiv 3 \pmod{10}$, what can we say about $x \pmod{5}$?

(b) If $x \equiv 3 \pmod{7}$, what can we say about $x \pmod{5}$?

Solution.

(a) If $x \equiv 3 \pmod{10}$, then $x \equiv 3 \pmod{5}$.
[Why?! Because $x \equiv 3 \pmod{10}$ if and only if $x = 3 + 10m$, which modulo 5 reduces to $x \equiv 3 \pmod{5}$.]

(b) Absolutely nothing! $x = 3 + 7m$ can be anything modulo 5 (because $7 \equiv 2$ is invertible modulo 5).

Example 67. If $x \equiv 32 \pmod{35}$, then $x \equiv 2 \pmod{5}$, $x \equiv 4 \pmod{7}$.

Why?! As in the first part of the warmup, if $x \equiv 32 \pmod{35}$, then $x \equiv 32 \pmod{5}$ and $x \equiv 32 \pmod{7}$.

The Chinese remainder theorem says that this can be reversed!

That is, if $x \equiv 2 \pmod{5}$ and $x \equiv 4 \pmod{7}$, then the value of x modulo $5 \cdot 7 = 35$ is determined.
[How to find the exact value of x, namely $x \equiv 32 \pmod{35}$, is discussed in the next example.]

Armin Straub
straub@southalabama.edu

Example 68. Solve $x \equiv 2 \pmod 5$, $x \equiv 4 \pmod 7$.

Solution. $x \equiv 2 \cdot 7 \cdot \underbrace{7^{-1}_{\bmod 5}}_{3} + 4 \cdot 5 \cdot \underbrace{5^{-1}_{\bmod 7}}_{3} \equiv 42 + 60 \equiv 32 \pmod{35}$

Important. Can you see how we need 5 and 7 to be coprime here?

Brute-force solution. Note that, while in principle we can always perform a brute-force search, this is not practical for larger problems. Here, if x is a solution, then so is $x + 35$. So we only look for solutions modulo 35.
Since $x \equiv 4 \pmod 7$, the only candidates for solutions are $4, 11, 18, \ldots$ Among these, we find $x = 32$.
[We can also focus on $x \equiv 2 \pmod 5$ and consider the candidates $2, 7, 12, \ldots$, but that is even more work.]

Example 69. Solve $x \equiv 1 \pmod 4$, $x \equiv 2 \pmod 5$.

Solution. $x \equiv 1 \cdot 5 \cdot \underbrace{5^{-1}_{\bmod 4}}_{1} + 2 \cdot 4 \cdot \underbrace{4^{-1}_{\bmod 5}}_{-1} \equiv 5 - 8 \equiv -3 \pmod{20}$

Example 70. Solve $x \equiv 1 \pmod 4$, $x \equiv 2 \pmod 5$, $x \equiv 3 \pmod 7$.

Solution. (option 1) By the previous problem, the first two congruences combine to $x \equiv -3 \pmod{20}$.
Using $x \equiv -3 \pmod{20}$, $x \equiv 3 \pmod 7$, we find $x \equiv -3 \cdot 7 \cdot \underbrace{7^{-1}_{\bmod 20}}_{3} + 3 \cdot 20 \cdot \underbrace{20^{-1}_{\bmod 7}}_{-1} \equiv -63 - 60 \equiv 17 \pmod{140}$.

Solution. (option 2) $x \equiv 1 \cdot 5 \cdot 7 \cdot \underbrace{[(5 \cdot 7)^{-1}_{\bmod 4}]}_{-1} + 2 \cdot 4 \cdot 7 \cdot \underbrace{[(4 \cdot 7)^{-1}_{\bmod 5}]}_{2} + 3 \cdot 4 \cdot 5 \cdot \underbrace{[(4 \cdot 5)^{-1}_{\bmod 7}]}_{-1} \equiv 17 \pmod{140}$

Theorem 71. (Chinese Remainder Theorem) Let $n_1, n_2, \ldots, n_r$ be positive integers with $\gcd(n_i, n_j) = 1$ for $i \neq j$. Then the system of congruences

$$x \equiv a_1 \pmod{n_1}, \quad \ldots, \quad x \equiv a_r \pmod{n_r}$$

has a simultaneous solution, which is unique modulo $n = n_1 \cdots n_r$.

In other words. The Chinese remainder theorem provides a bijective (i.e., 1-1 and onto) correspondence

$$x \pmod{nm} \mapsto \begin{bmatrix} x \pmod n \\ x \pmod m \end{bmatrix}$$

provided that m and n are coprime.

For instance. Let's make the correspondence explicit for $n = 2$, $m = 3$:
$0 \mapsto \left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$, $1 \mapsto \left[\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right]$, $2 \mapsto \left[\begin{smallmatrix} 0 \\ 2 \end{smallmatrix}\right]$, $3 \mapsto \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$, $4 \mapsto \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$, $5 \mapsto \left[\begin{smallmatrix} 1 \\ 2 \end{smallmatrix}\right]$

Example 72. Solve $x \equiv 4 \pmod 5$, $x \equiv 10 \pmod{13}$.

Solution. $x \equiv 4 \cdot 13 \cdot \underbrace{13^{-1}_{\bmod 5}}_{2} + 10 \cdot 5 \cdot \underbrace{5^{-1}_{\bmod 13}}_{-5} \equiv 104 - 250 \equiv 49 \pmod{65}$

Check. Since it is easy to do so, we should quickly check our answer: $49 \equiv 4 \pmod 5$, $49 \equiv 10 \pmod{13}$

Example 73. Let $p, q > 3$ be distinct primes.

(a) Show that $x^2 \equiv 9 \pmod p$ has exactly two solutions (i.e. ± 3).

(b) Show that $x^2 \equiv 9 \pmod{pq}$ has exactly four solutions (± 3 and two more solutions $\pm a$).

Solution.

(a) If $x^2 \equiv 9 \pmod p$, then $0 \equiv x^2 - 9 = (x-3)(x+3) \pmod p$. Since p is a prime it follows that $x - 3 \equiv 0 \pmod p$ or $x + 3 \equiv 0 \pmod p$. That is, $x \equiv \pm 3 \pmod p$.

(b) By the CRT, we have $x^2 \equiv 9 \pmod{pq}$ if and only if $x^2 \equiv 9 \pmod p$ and $x^2 \equiv 9 \pmod q$. Hence, $x \equiv \pm 3 \pmod p$ and $x \equiv \pm 3 \pmod q$. These combine in four different ways.
For instance, $x \equiv 3 \pmod p$ and $x \equiv 3 \pmod q$ combine to $x \equiv 3 \pmod{pq}$. However, $x \equiv 3 \pmod p$ and $x \equiv -3 \pmod q$ combine to something modulo pq which is different from 3 or -3.

Why primes >3? Why did we exclude the primes 2 and 3 in this discussion?

Comment. There is nothing special about 9. The same is true for $x^2 \equiv a^2 \pmod{pq}$ for each integer a.

Example 74. Determine all solutions to $x^2 \equiv 9 \pmod{35}$.

Solution. By the CRT:

$$\begin{aligned} & x^2 \equiv 9 \pmod{35} \\ \iff\ & x^2 \equiv 9 \pmod 5 \text{ and } x^2 \equiv 9 \pmod 7 \\ \iff\ & x \equiv \pm 3 \pmod 5 \text{ and } x \equiv \pm 3 \pmod 7 \end{aligned}$$

The two obvious solutions modulo 35 are ± 3. To get one of the two additional solutions, we solve $x \equiv 3 \pmod 5$, $x \equiv -3 \pmod 7$. [Then the other additional solution is the negative of that.]

$x \equiv 3 \cdot 7 \cdot \underbrace{7^{-1}_{\bmod 5}}_{3} - 3 \cdot 5 \cdot \underbrace{5^{-1}_{\bmod 7}}_{3} \equiv 63 - 45 \equiv 18 \pmod{35}$

Hence, the solutions are $x \equiv \pm 3 \pmod{35}$ and $x \equiv \pm 17 \pmod{35}$. [$\pm 18 \equiv \pm 17 \pmod{35}$]

Silicon slave labor. We can let Sage (more next page!) do the work for us:

```
Sage] solve_mod(x^2 == 9, 35)
```

$[(17), (32), (3), (18)]$

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.

[For basic computations, you can also simply use the textbox on our course website.]

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

Example 75. 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 76. 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

Example 77. (review) The solutions to $x^2 \equiv 9 \pmod{35}$ are ± 3 and $\pm 17 \pmod{35}$.

Example 78. Determine all solutions to $x^2 \equiv 4 \pmod{105}$.

Solution. By the CRT:

$$\begin{aligned} & x^2 \equiv 4 \pmod{105} \\ \iff\ & x^2 \equiv 4 \pmod 3 \text{ and } x^2 \equiv 4 \pmod 5 \text{ and } x^2 \equiv 4 \pmod 7 \\ \iff\ & x \equiv \pm 2 \pmod 3 \text{ and } x \equiv \pm 2 \pmod 5 \text{ and } x \equiv \pm 2 \pmod 7 \end{aligned}$$

At this point, we see that there are $2^3 = 8$ solutions.

For instance, let us find the solution corresponding to $x \equiv 2 \pmod 3$, $x \equiv 2 \pmod 5$, $x \equiv -2 \pmod 7$:

$$x \equiv 2 \cdot 5 \cdot 7 \cdot \underbrace{[(5 \cdot 7)^{-1}_{\bmod 3}]}_{-1} + 2 \cdot 3 \cdot 7 \cdot \underbrace{[(3 \cdot 7)^{-1}_{\bmod 5}]}_{1} - 2 \cdot 3 \cdot 5 \cdot \underbrace{[(3 \cdot 5)^{-1}_{\bmod 7}]}_{1} \equiv -70 + 42 - 30 = -58 \equiv 47$$

Similarly, we find all eight solutions (note how the solutions pair up):

$\pmod 3$	$\pmod 5$	$\pmod 7$	$\pmod{105}$
2	2	2	2
-2	-2	-2	-2
2	2	-2	47
-2	-2	2	-47
2	-2	2	23
-2	2	-2	-23
-2	2	2	37
2	-2	-2	-37

The complete list of solutions is: $\pm 2, \pm 23, \pm 37, \pm 47$

Silicon slave labor. Once we are comfortable doing it by hand, we can easily let Sage do the work for us:

```
Sage] crt([2,2,-2], [3,5,7])
```

47

```
Sage] solve_mod(x^2 == 4, 105)
```

$[(37), (82), (58), (103), (2), (47), (23), (68)]$

Review: quadratic residues

Definition 79. An integer a is a **quadratic residue** modulo n if $a \equiv x^2 \pmod n$ for some x.

Important note. Products of quadratic residues are quadratic residues.

Example 80. List all quadratic residues modulo 11.

Solution. We compute all squares: $0^2 = 0$, $(\pm 1)^2 = 1$, $(\pm 2)^2 = 4$, $(\pm 3)^2 = 9$, $(\pm 4)^2 \equiv 5$, $(\pm 5)^2 \equiv 3$. Hence, the quadratic residues modulo 11 are $0, 1, 3, 4, 5, 9$.

Important comment. Exactly half of the 10 nonzero residues are quadratic. Can you explain why?

[*Hint.* $x^2 \equiv y^2 \pmod p \iff (x-y)(x+y) \equiv 0 \pmod p \iff x \equiv y$ or $x \equiv -y \pmod p$]

Example 81. List all quadratic residues modulo 15.

Solution. We compute all squares modulo 15: $0^2 = 0$, $(\pm 1)^2 = 1$, $(\pm 2)^2 = 4$, $(\pm 3)^2 = 9$, $(\pm 4)^2 \equiv 1$, $(\pm 5)^2 \equiv 10$, $(\pm 6)^2 \equiv 6$, $(\pm 7)^2 \equiv 4$. Hence, the quadratic residues modulo 15 are $0, 1, 4, 6, 9, 10$.

Important comment. Among the $\phi(15) = 8$ invertible residues, the quadratic ones are $1, 4$ (exactly a quarter). Note that 15 is of the form $n = pq$ with p, q distinct primes.

Armin Straub
straub@southalabama.edu

Theorem 82. Let p, q, r be distinct odd primes.

- The number of invertible residues modulo n is $\phi(n)$.
- The number of invertible quadratic residues modulo p is $\frac{\phi(p)}{2} = \frac{p-1}{2}$.
- The number of invertible quadratic residues modulo pq is $\frac{\phi(pq)}{4} = \frac{p-1}{2}\frac{q-1}{2}$.
- The number of invertible quadratic residues modulo pqr is $\frac{\phi(pqr)}{8} = \frac{p-1}{2}\frac{q-1}{2}\frac{r-1}{2}$.
- ...

Proof.

- We already knew that the number of invertible residues modulo n is $\phi(n)$.
- Think about squaring all residues modulo p to make a complete list of all quadratic residues. Let a^2 be one of the nonzero quadratic residues. As we observed earlier, $x^2 \equiv a^2 \pmod{p}$ has exactly 2 solutions, meaning that exactly two residues (namely $\pm a$) square to a^2. Hence, the number of invertible quadratic residues modulo p is half the number of invertible residues modulo p.
- Again, think about squaring all residues modulo pq to make a complete list of all quadratic residues. Let a^2 be one of the invertible quadratic residues. By the CRT, $x^2 \equiv a^2 \pmod{pq}$ has exactly 4 solutions (why is it important that a is invertible here?!), meaning that exactly four residues square to a^2. Hence, the number of invertible quadratic residues modulo pq is a quarter of the number of invertible residues modulo pq.
- Spell out the situation modulo pqr! □

Comment. Make similar statements when one of the primes is equal to 2.

Example 83. (bonus!) What is the total number of quadratic residues modulo pqr if p, q, r are distinct odd primes? (To collect a bonus point, send me the answer and a short explanation by next week.)

Example 84. (bonus!) The LFSR $x_{n+31} \equiv x_{n+28} + x_n \pmod 2$ from Example 58, which is used in glibc, is entirely predictable because observing $x_1, x_2, \ldots, x_{31}$ we know what $x_{32}, x_{33}, \ldots$ are going to be. Alice tries to reduce this predictability by using only $x_3, x_6, x_9, \ldots$ as the output of the LFSR. Demonstrate that this PRG is still perfectly predictable by showing the following:

Challenge. Find a simple LFSR which produces $x_3, x_6, x_9, \ldots$

Send me the LFSR, and an explanation how you found it, by next week for a bonus point!

Comment. There is nothing special about this LFSR. Moreover, a generalization of this argument shows that only outputting every Nth bit of an LFSR is always going to result in an entirely predictable PRG.

The Blum-Blum-Shub PRG

The Blum-Blum-Shub PRG is an example of a PRG, which is believed to be unpredictable.

More precisely, it has been shown that the ability to predict its values is equivalent to being able to efficiently solve the quadratic residuosity problem (which is believed to be hard). Currently, the best way to "solve" the quadratic residuosity problem mod M relies on factoring M. However, factoring large numbers is considered to be hard (and lots of crypto relies on that).

Quadratic residuosity problem. Given big $M = pq$ and a residue x modulo M, decide whether x is a quadratic residue. (About $M/4$ are quadratic residues (the exact number is $\phi(M)/4 = (p-1)(q-1)/4$); $M/2$ are easily determined to be nonsquare using the Jacobi symbol [don't worry if you haven't heard about that].)

(Blum-Blum-Shub PRG) Let $M = pq$ where p, q are large primes $\equiv 3 \pmod 4$.
From the seed y_0, we generate $y_{n+1} \equiv y_n^2 \pmod M$.
The random bits x_n we produce are $y_n \pmod 2$ (i.e. $x_n = \text{least bit of}(y_n)$).

B-B-S is very slow, and mostly of theoretical value. However, as indicated above, it is interesting because it is indeed unpredictable (to anyone not knowing the factorization of M) if an important number theory problem (the quadratic residuosity problem) is "hard" (this can be made precise), as is believed to be the case.

Why the conditions on p and q? Recall from the CRT that an invertible quadratic residue x^2 modulo $M = pq$ has exactly four squareroots $\pm x$, $\pm y$. The condition $3 \pmod 4$ guarantees that, of these four, exactly one is itself a quadratic residue. As a consequence, the mapping $y \mapsto y^2 \pmod M$ is 1-1 when restricted to invertible quadratic residues (see below).

Comment. For obvious reasons, the seed $y_0 \equiv \pm 1 \pmod M$ should be excluded. Also, for the above considerations to apply, the seed needs to be coprime to M. However, we don't need to worry about that: running into a factor of M by accident is close to impossible (recall that nobody should be able to factor M even on purpose and with lots of time and resources).

Comment. To increase speed, at the expense of some security, we can also take several, say k, bits of y_n (as long as k is small, say, $k \leqslant \log_2 \log_2 M$).

Example 85. Generate random bits using the B-B-S PRG with $M = 77$ and seed 3.

Solution. With $y_0 = 3$, we have $y_1 \equiv y_0^2 = 9$, followed by $y_2 \equiv y_1^2 \equiv 4 \pmod{77}$, $y_3 \equiv 16$, $y_4 \equiv 25$, $y_5 \equiv 9$, so that the values y_n now start repeating.

These numbers are, however, not the output of the PRG. We only output the least bit of the numbers y_n, i.e. the value of $y_n \pmod 2$. For $y_1 \equiv 9$ we output 1, for $y_2 \equiv 4$ we output 0, for $y_3 \equiv 16$ we output 0, for $y_4 \equiv 25$ we output 1, and so on.

In other words, the seed 3 produces the sequence $1, 0, 0, 1, 1, 0, 0, 1, 1, 0, \ldots$ of period 4.

Comment. Note that it was completely to be expected that the numbers repeat. In fact, we immediately see that the number of possible y_n is at most the number of invertible quadratic residues, of which there are only $\phi(77)/4 = 15$.

Armin Straub
straub@southalabama.edu

Example 86.

(a) List all invertible quadratic residues modulo 21. Compute the square of all these residues.

(b) Repeat the first part modulo 33 and modulo 35. When computing the squares of these, do you notice a difference modulo 35?

[Note that $35 = 5 \cdot 7$ with $5 \equiv 1 \pmod 4$. This case is excluded in the B-B-S PRG.]

Solution. (final answers only)

(a) Among the $\phi(21) = 12$ many invertible elements, the squares are $1, 4, 16$ (exactly a quarter).
Computing their squares: $1^2 \equiv 1$, $4^2 \equiv 16$, $16^2 \equiv 4 \pmod{21}$. Note that the squares are all different!

(b) Modulo 33: among the $\phi(33) = 20$ many invertible elements, the squares are $1, 4, 16, 25, 31 \equiv 8^2$ (exactly a quarter). Computing their squares: $1^2 \equiv 1$, $4^2 \equiv 16$, $16^2 \equiv 25$, $25^2 \equiv 31$, $31^2 \equiv 4 \pmod{33}$. Again, all the squares are different!

Modulo 35: among the $\phi(35) = 24$ many invertible elements, the squares are $1, 4, 9, 11 \equiv 9^2, 16, 29 \equiv 8^2$ (exactly a quarter). Computing their squares: $1^2 \equiv 1$, $4^2 \equiv 16$, $9^2 \equiv 11$, $11^2 \equiv 16$, $16^2 \equiv 11$, $29^2 \equiv 1 \pmod{35}$. Observe that these are not all different: for instance, $9^2 \equiv 16^2 \pmod{35}$.

Advanced comment. The map $x \mapsto x^2 \pmod p$ restricted to invertible quadratic residues is 1-1 if and only if -1 is not a quadratic residue (which, by the next result, is equivalent to $p \equiv 3 \pmod 4$).

[*Sketch of proof.* The map is 1-1 if and only if, for each invertible quadratic residue x^2, exactly one of the two square roots $\pm x$ is itself a quadratic residue. This is equivalent to -1 not being a quadratic residue.

Indeed, if -1 is a quadratic residue, then x and $-x$ are either both quadratic residues or both not.

On the other hand, if not exactly one of $\pm x$ is a quadratic residue then, because exactly half of the invertible residues are quadratic, there would be some pair of residues $\pm z$ which are both quadratic. But then $-z$ and z^{-1} are quadratic residues which implies that their product $-zz^{-1} \equiv -1$ would be a quadratic residue as well.]

Theorem 87. -1 is a quadratic residue modulo (an odd prime) p if and only if $p \equiv 1 \pmod 4$.

In other words, the quadratic congruence $x^2 \equiv -1 \pmod p$ has a solution if and only if $p \equiv 1 \pmod 4$.

Solution. Let us first see that $p \equiv 1 \pmod 4$ is necessary. Assume $x^2 \equiv -1 \pmod p$. Then, by Fermat's little theorem, $x^{p-1} \equiv 1 \pmod p$. On the other hand, $x^{p-1} = (x^2)^{(p-1)/2} \equiv (-1)^{(p-1)/2} \pmod p$. We therefore need $(-1)^{(p-1)/2} = 1$, which is equivalent to $(p-1)/2$ being even. Which is equivalent to $p \equiv 1 \pmod 4$. (Make sure that's absolutely clear!)

On the other hand, assume that $p \equiv 1 \pmod 4$. We will show that $x = \left(\frac{p-1}{2}\right)!$ has the property that $x^2 \equiv -1 \pmod p$. Indeed,

$$\left[\left(\frac{p-1}{2}\right)!\right]^2 = (-1)^{(p-1)/2}\left(1 \cdot 2 \cdot \ldots \cdot \frac{p-1}{2}\right)^2 = (\pm 1) \cdot (\pm 2) \cdot \ldots \cdot \left(\pm\frac{p-1}{2}\right) \equiv -1 \pmod p.$$

[Here, $(\pm 1) \cdot (\pm 2) \cdots$ is short for $1 \cdot (-1) \cdot 2 \cdot (-2) \cdots$.] For the final congruence, observe that $\pm 1, \pm 2, \ldots, \pm\frac{p-1}{2}$ is a complete set of all nonzero residues. When multiplying all residues, each will cancel with its (modular) inverse, except the ones that are their own inverse. But $a \cdot a \equiv 1 \pmod p$ has only the solution $a \equiv \pm 1$, so that ± 1 are the only residues not canceling.

Comment. The final step of our argument is known as Wilson's congruence: $(p-1)! \equiv -1 \pmod p$.

Theorem 88. (advanced) Let $M=pq$ where p,q are primes $\equiv 3 \pmod 4$. Then the sequence generated by $y_{n+1} \equiv y_n^2 \pmod M$ repeats with period dividing $\operatorname{lcm}(\phi(p-1), \phi(q-1))$.

In particular, the period of the corresponding B-B-S PRG divides $\operatorname{lcm}(\phi(p-1), \phi(q-1))$.

Proof.

- Observe that the numbers are $y_n = y_{n-1}^2 = y_{n-2}^4 = \ldots = y_0^{2^n} \pmod M$. Hence, $y_n \equiv y_0^{2^n} \pmod M$.
- Instead of determining the period directly modulo $M=pq$, we determine the periods modulo p and q. [Why? By the CRT, $y_m \equiv y_n \pmod M$ if and only if $y_m \equiv y_n \pmod p$ and $y_m \equiv y_n \pmod q$.] The period modulo M then is the lcm of of the two periods modulo p and q.
- $y_m \equiv y_n \pmod p$
  $\iff \quad y_0^{2^m} \equiv y_0^{2^n} \pmod p$
  $\impliedby \quad 2^m \equiv 2^n \pmod{\phi(p)}$
  [it would be "$\iff$" with $2^m \equiv 2^n \pmod k$ where k is the order of $y_0 \pmod p$]
  $\iff \quad 2^m \equiv 2^n \pmod{p-1}$
  [note that 2 is not invertible $\pmod{p-1}$; but 2 is invertible $\left(\bmod \frac{p-1}{2}\right)$ because $p \equiv 3 \pmod 4$]
  $\iff \quad 2^{m-1} \equiv 2^{n-1} \left(\bmod \frac{p-1}{2}\right)$ [note that $m,n \geqslant 1$]
  $\impliedby \quad m \equiv n \left(\bmod \phi\left(\frac{p-1}{2}\right)\right)$
  [again, it would be "$\iff$" with $m \equiv n \pmod k$ where k is the order of $2 \left(\bmod \frac{p-1}{2}\right)$]
- Reading this backwards, we see that the sequence y_n modulo p repeats after $\phi\left(\frac{p-1}{2}\right)$ terms.
  In other words, the (minimal) period divides $\phi\left(\frac{p-1}{2}\right) = \phi(p-1)$.

  Comment. Here we used $p \equiv 3 \pmod 4$ to conclude $\phi\left(\frac{p-1}{2}\right) = \phi(p-1)$. Indeed, in that case, $p-1$ is divisible by 2 but not by 4. Hence, 2 and $\frac{p-1}{2}$ are coprime, so that $\phi(p-1) = \phi(2)\phi\left(\frac{p-1}{2}\right) = \phi\left(\frac{p-1}{2}\right)$.
- By the CRT, the period modulo $M=pq$ divides $\operatorname{lcm}(\phi(p-1), \phi(q-1))$. □

Example. In Example 85, we had $M = 7 \cdot 11$, so that the period of the PRG must divide $\operatorname{lcm}(\phi(6), \phi(10)) = \operatorname{lcm}(2,4) = 4$.

Comment. In practice, people therefore say that, for the cycle length of B-B-S to be large, $\gcd(\phi(p-1), \phi(q-1))$ should be small.

Example 89. We mentioned that the unpredictability of the B-B-S PRG relies on the difficulty of factoring large numbers. Here's an indication how difficult it seems to be. In 1991, RSA Laboratories challenged everyone to factor several numbers including:

```
13506641086599522334960321627880596993888147560566702752448514385152651\
06048595338339402871505719094417982072821644715513736804197039641917\
30464965892742562393410208643832021103729587257623585096431105640735\
50818751067659462920556368552947521350085287941637732853390610975054\
43499981115005697723689092756 3
```

Since then, nobody has been able to factor this 1024 bit number (309 decimal digits). Until 2007, cash prizes were offered up to 200,000 USD, with 100,000 USD for the number above.

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

Let us illustrate how to actually use this number in the B-B-S PRG.

```
Sage] rsa = Integer("135066410865995223349603216278805969938881475605667027524485143851\
                     526510604859533833940287150571909441798207282164471551373680419703\
                     964191743046496589274256239341020864383202110372958725762358509643\
                     110564073501508187510676594629205563685529475213500852879416377328\
                     533906109750544334999811150056977236890927563")

Sage] seed = randint(2,rsa-2)

Sage] y = seed; prg = []

Sage] for i in [1..25]:
          y = power_mod(y, 2, rsa)
          prg.append(y % 2)

Sage] prg
```

$$[0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1]$$

If you are able, even after gigabytes of pseudorandom bits, to predict the next bits with an accuracy better than 50% (which is just pure guessing), then you likely have a shot at factoring the big integer. You would be the first!

Of course, it is not impressive to see a few random bits in the example above. After all, the seed (which you don't know!) itself consists of 1024 random bits. The whole point is that we can, from these 1024 random bits, produce gigabytes of further pseudorandom bits. As of this day, no one would be able to distinguish these from truly random bits.

While all of this works nicely, B-B-S is considered to be too slow for most practical purposes.

Comment. Note that $M = 135...563 \equiv 3 \pmod 4$. Hence it cannot be a product of primes p, q which are both $3 \pmod 4$ (because $3 \cdot 3 \equiv 1 \pmod 4$).

Example 90. (extra) Generate random bits using the B-B-S PRG with $M = 209$ and seed 10. What is the period of the generated sequence? (Then repeat with seed 25.)

Solution. (final answer only) The seed 10 produces the sequence $0, 1, 0, 1, 1, 1, \ldots$ of period 6.

The seed 25 generates the sequence $1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, \ldots$ of period 12.

[By the way, it is an excellent idea to let Sage assist you.]

Primality testing

Recall that it is extremely difficult to factor large integers (this is the starting point for many cryptosystems). Surprisingly, it is much simpler to tell if a number is prime.

Example 91. The following is the number from Example 89, for which RSA Laboratories, until 2007, offered $100,000 to the first one to factorize it. Nobody has been able to do so to this day.

Has the thought crossed your mind that the challengers might be tricking everybody by choosing M to be a huge prime that cannot be factored further? Well, we'll talk more about primality testing soon. But we can actually quickly convince ourselves that M cannot be a prime. If M was prime then, by Fermat's little theorem, $2^{M-1} \equiv 1 \pmod{M}$. Below, we compute $2^{M-1} \pmod{M}$ and find that $2^{M-1} \not\equiv 1 \pmod{M}$. This proves that M is not a prime. It doesn't bring us any closer to factoring it though.

Comment. Ponder this for a while. We can tell that a number is composite without finding its factors. Both sides to this story (first, being able to efficiently tell whether a number is prime, and second, not being able to factor large numbers) are of vital importance to modern cryptography.

```
Sage] rsa = Integer("135066410865995223349603216278805969938881475605667027524485143851\
                     525510604859533833940287150571909441798207282164471551373680419703\
                     964191743046496589274256239341020864383202110372958725762358509643\
                     110564073501508187510676594629205563685529475213500852879416377328\
                     533906109750544334999811150056977236890927563")

Sage] power_mod(2, rsa-1, rsa)

    120939094432033615867650595352956996867540098463588951238902808367556733932202059338 53\
    348534147116662841968124107288512373904071077139405352848835710498409193003137847878 95\
    226029615123284879513798127406300472693925500331497519103479951096634123177725212482 97\
    950196643140069546889855131459759160570963857373851
```

Comment. Just for giggles, let us emphasize once more the need to compute $2^{N-1} \pmod{N}$ without actually computing 2^{N-1}. Take, for instance, the 1024 bit RSA challenge number $N = 135...563$. In Example 91, we computed $2^{N-1} \pmod{N}$, observed that it was $\not\equiv 1$ and concluded that N is not prime. The number 2^{N-1} itself has $N \approx 2^{1024} \approx 10^{308.3}$ binary digits. It is often quoted that the number of particles in the visible universe is estimated to be between 10^{80} and 10^{100}. Whatever these estimates are worth, our number has WAY more digits (!) than that. Good luck writing it out! [Of course, the binary digits are a single 1 followed by all zeros. However, we need to further compute with that!]

Comment. There is nothing special about 2. You could just as well use, say, 3.

Example 92. (bonus challenge) Find the factors of the following number $M = pq$:

89320280057437363393608386387469360495079915773073599087435569428108 27\
07615146116506918133536640188765047775335776026093439165454319252186 33\
75114106509563452970373049082933244013107347141654282924032714311

As indicated in Example 89, this is difficult. Through some sort of espionage, however, you have learned that $\phi(M)$ is:

89320280057437363393608386387469360495079915773073599087435569428108 27\
07615146116506918133536640188675726495278338662699830779066849891691 25\
75956375773572578614678768000225628866990840223520746283867797512

In general, if $M = pq$ is a product of two large primes p, q, given $\phi(M)$, how can we factor M?

Send me the factorization, and an explanation how you found it, by next week for a bonus point!

Comment. Even if we don't know the number of prime factors of M (in the above case we know that M is a product of two primes), we can "efficiently" factor M if we know the value of $\phi(M)$.

Armin Straub
straub@southalabama.edu

Example 93. Fermat's little theorem can be stated in the slightly stronger form:

n is a prime $\iff$ $a^{n-1} \equiv 1 \pmod{n}$ for all $a \in \{1, 2, ..., n-1\}$

Why? Fermat's little theorem covers the "$\Longrightarrow$" part. The "$\Longleftarrow$" part is a direct consequence of the fact that, if n is composite with divisor d, then $d^{n-1} \not\equiv 1 \pmod{n}$. (Why?!)

Fermat primality test

Input: number n and parameter k indicating the number of tests to run

Output: "not prime" or "likely prime"

Algorithm:

Repeat k times:
  Pick a random number a from $\{2, 3, ..., n-2\}$.
  If $a^{n-1} \not\equiv 1 \pmod{n}$, then stop and output "not prime".
Output "likely prime".

If $a^{n-1} \equiv 1 \pmod{n}$ although n is composite, then a is called a **Fermat liar** modulo n.

On the other hand, if $a^{n-1} \not\equiv 1 \pmod{n}$, then n is composite and a is called a **Fermat witness** modulo n.

Flaw. There exist certain composite numbers n (see Definition 97) for which every a is a Fermat liar (or reveals a factor of n). For this reason, the Fermat primality test should not be used as a general test for primality. That being said, for very large random numbers, it is exceedingly unlikely to meet one of these troublesome numbers, and so the Fermat test is indeed used for the purpose of randomly generating huge primes (for instance in PGP). In fact, in that case, we can even always choose $a = 2$ and $k = 1$ with virtual certainty of not messing up.

Next class, we will discuss an extension of the Fermat primality test which solves these issues (and is just mildly slower).

Advanced comment. If n is composite but not an absolute pseudoprime (see Definition 97), then at least half of the values for a satisfy $a^{n-1} \not\equiv 1 \pmod{n}$ and so reveal that n is not a prime. This is more of a theoretical result: for most large composite n, almost every a (not just half) will be a Fermat witness.

Example 94. Suppose we want to determine whether $n = 221$ is a prime. Simulate the Fermat primality test for the choices $a = 38$ and $a = 24$.

Solution.

- First, maybe we pick $a = 38$ randomly from $\{2, 3, ..., 219\}$.
  We then calculate that $38^{220} \equiv 1 \pmod{221}$. So far, 221 is behaving like a prime.
- Next, we might pick $a = 24$ randomly from $\{2, 3, ..., 219\}$.
  We then calculate that $24^{220} \equiv 81 \not\equiv 1 \pmod{221}$. We stop and conclude that 221 is not a prime.

Important comment. We have done so without finding a factor of n. (To wit, $221 = 13 \cdot 17$.)

Comment. Since 38 was giving us a false impression regarding the primality of n, it is called a **Fermat liar** modulo 221. Similarly, we say that 221 is a **pseudoprime** to the base 38.

On the other hand, we say that 24 was a **Fermat witness** modulo 221.

Comment. In this example, we were actually unlucky that our first "random" pick was a Fermat liar: only 14 of the 218 numbers (about 6.4%) are liars. As indicated above, for most large composite numbers, the proportion of liars will be exceedingly small.

Example 95. Which of $6, 7, 8, 9$ are Fermat liars modulo 25?

Solution. Recall that a is a Fermat liar modulo 25 if $a^{24} \equiv 1 \pmod{25}$. We compute $6^{24} \equiv 21$, $7^{24} \equiv 1$, $8^{24} \equiv 21$, $9^{24} \equiv 11$ (all modulo 25). It follows that, among those four, only 7 is a Fermat liar modulo 25.

Example 96. Which of $10, 15, 20, 25, 30$ are pseudoprimes to the base 7?

Solution. Recall that n is a pseudoprime to the base 7 if $7^{n-1} \equiv 1 \pmod{n}$. We compute $7^9 \equiv 7 \pmod{10}$, $7^{14} \equiv 4 \pmod{15}$, $7^{19} \equiv 3 \pmod{20}$, $7^{24} \equiv 1 \pmod{25}$, $7^{29} \equiv 7 \pmod{30}$. It follows that, among those five, only 25 is a pseudoprime to the base 7.

Absolute pseudoprimes

Somewhat suprisingly, there exist composite numbers n with the following disturbing property: every residue a is a Fermat liar or $\gcd(a, n) > 1$.

This means that the Fermat primality test is unable to distinguish n from a prime, unless the randomly picked number a happens to reveal a factor (namely, $\gcd(a, n)$) of n (which is exceedingly unlikely for large numbers).

[Recall that, for large numbers, we do not know how to find factors even if that was our primary goal.]

Such numbers are called absolute pseudoprimes:

Definition 97. A composite positive integer n is an **absolute pseudoprime** (or Carmichael number) if $a^{n-1} \equiv 1 \pmod{n}$ holds for each integer a with $\gcd(a, n) = 1$.

The first few are $561, 1105, 1729, 2465, \ldots$ (it was only shown in 1994 that there are infinitely many of them).

These are very rare, however: there are 43 absolute pseudoprimes less than 10^6. (Versus $78,498$ primes.)

Example 98. Show that 561 is an absolute pseudoprime.

Solution. We need to show that $a^{560} \equiv 1 \pmod{561}$ for all invertible residues a modulo 561.

Since $561 = 3 \cdot 11 \cdot 17$, $a^{560} \equiv 1 \pmod{561}$ is eqivalent to $a^{560} \equiv 1 \pmod{p}$ for each of $p = 3, 11, 17$.

By Fermat's little theorem, we have $a^2 \equiv 1 \pmod{3}$, $a^{10} \equiv 1 \pmod{11}$, $a^{16} \equiv 1 \pmod{17}$. Since $2, 10, 16$ each divide 560, it follows that indeed $a^{560} \equiv 1 \pmod{p}$ for $p = 3, 11, 17$.

Comment. Korselt's criterion (1899) states that what we just observed in fact characterizes absolute pseudoprimes. Namely, a composite number n is an absolute pseudoprime if and only if n is squarefree, and for all primes p dividing n, we also have $p - 1 | n - 1$.

Comment. Our argument above shows that, in fact, $a^{80} \equiv 1 \pmod{561}$ for all invertible residues a modulo 561.

Theorem 99. (Korselt's Criterion) A composite positive integer n is an absolute pseudoprime if and only if n is squarefree and $(p-1)|(n-1)$ for each prime divisor p of n.

Proof. Here, we will only consider the "if" part (the "only if" part is also not hard to show but the typical proof requires a little more insight into primitive roots than we currently have).

To that end, assume that n is **squarefree** and that $(p-1)|(n-1)$ for each prime divisor p of n. Let a be any integer with $\gcd(a, n) = 1$. We will show that $a^{n-1} \equiv 1 \pmod{n}$.

n being squarefree means that its prime factorization is of the form $n = p_1 \cdot p_2 \cdots p_d$ for distinct primes p_i (this is equivalent to saying that there is no integer $m > 1$ such that $m^2 | n$). By Fermat's little theorem $a^{p_i - 1} \equiv 1 \pmod{p_i}$ and, since $(p_i - 1)|(n-1)$, we have $a^{n-1} \equiv 1 \pmod{p_i}$ for all p_i. It therefore follows from the Chinese remainder theorem that $a^{n-1} \equiv 1 \pmod{n}$. □

Comment. Modulo a prime p, Fermat's little theorem implies that $a^p \equiv a \pmod{p}$ for each integer a. (Why?!) It therefore follows from the above argument that, for an absolute pseudoprime n, we have $a^n \equiv a \pmod{n}$ for each integer a (and this property characterizes absolute pseudoprimes).

Example 100. How can you check whether a huge randomly selected number N is prime?

Solution. Compute $2^{N-1} \pmod N$ using binary exponentiation. If this is $\not\equiv 1 \pmod N$, then N is not a prime.
Otherwise, N is a prime or 2 is a Fermat liar modulo N (but the latter is exceedingly unlikely for a huge randomly selected number N; the bonus challenge below indicates that this is almost as unlikely as randomly running into a factor of N).

Comment. There is nothing special about 2 here (you could also choose 3 or any other generic residue).

How many primes are there?

Theorem 101. (Euclid) There are infinitely many primes.

Proof. Assume (for contradiction) there are only finitely many primes: $p_1, p_2, ..., p_n$.
Consider the number $N = p_1 \cdot p_2 \cdot ... \cdot p_n + 1$.
None of the p_i divide N (because division of N by any p_i leaves remainder 1).
Thus any prime dividing N is not on our list. Contradiction.

Just being silly. Similarly, there are infinitely many composite numbers.

Indeed, assume (for contradiction) there are only finitely many composites: $m_1, m_2, ..., m_n$.
Consider the number $N = m_1 \cdot m_2 \cdot ... \cdot m_n$ (don't add 1).
N is not on our list. Contradiction.

Historical note. This is not necessarily a proof by contradiction, and Euclid (300BC) himself didn't state it as such. Instead, one can think of it as a constructive machinery of producing more primes, starting from any finite collection of primes. □

The following famous and deep result quantifies the infinitude of primes.

Theorem 102. (prime number theorem) Let $\pi(x)$ be the number of primes $\leqslant x$. Then

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln(x)} = 1.$$

In other words: Up to x, there are roughly $x/\ln(x)$ many primes.

Examples.

Proportion of primes up to 10^6: $\frac{78,498}{10^6} = 7.85\%$ vs the estimate $\frac{1}{\ln(10^6)} = \frac{1}{6\ln(10)} = 7.24\%$

Proportion of primes up to 10^{12}: $\frac{37,607,912,018}{10^{12}} = 3.76\%$ vs the estimate $\frac{1}{\ln(10^{12})} = \frac{1}{12\ln(10)} = 3.62\%$

An example of huge relevance for crypto.

By the PNT, the proportion of primes up to 2^{2048} is about $\frac{1}{\ln(2^{2048})} = \frac{1}{2048 \cdot \ln(2)} = 0.0704\%$.

That means, roughly, 1 in 1500 numbers of this magnitude are prime. That means we (i.e. our computer) can efficiently generate large random primes by just repeatedly generating large random numbers and discarding those that are not prime.

Comment. Here, $\ln(x)$ is the logarithm with base e. Isn't it wonderful how Euler's number $e \approx 2.71828$ is sneaking up on the primes?

Historical comment. Despite progress by Chebyshev (who succeeded in 1852 in showing that the quotient in the above limit is bounded, for large x, by constants close to 1), the PNT was not proved until 1896 by Hadamard and, independently, de la Vallée Poussin, who both used new ideas due to Riemann.

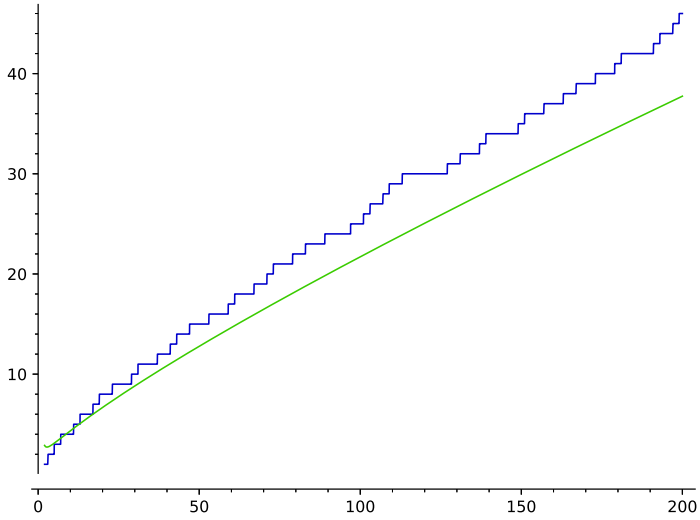
Armin Straub
straub@southalabama.edu

Example 103. Playing with the prime number theorem in Sage:

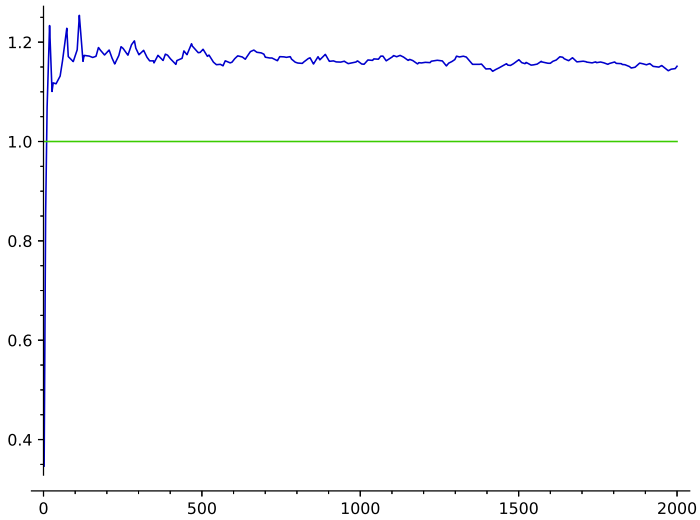
```
Sage] prime_pi(10)
```

```
4
```

```
Sage] plot([prime_pi(x), x/ln(x)], 2, 200)
```




```
Sage] plot([prime_pi(x)/(x/ln(x)), 1], 2, 2000)
```



Comment. As the final plot suggests, the quotient of $\pi(x)$ and $x/\ln(x)$ indeed approaches 1 from above. This is slightly stronger than the PNT, which only claims that the quotient approaches 1.

In particular, as the previous plot suggests, for large x, $x/\ln(x)$ is always an underestimate for $\pi(x)$ (though looking at a plot like this can be very misleading).

Review. If N is composite, then a residue a is a Fermat liar modulo N if $a^{N-1} \equiv 1 \pmod{N}$.

Example 104. Using Sage, determine all numbers n up to 5000, for which 2 is a Fermat liar.

```
Sage] def is_fermat_liar(x, n):
          return not is_prime(n) and power_mod(x, n-1, n) == 1

Sage] [ n for n in [1..5000] if is_fermat_liar(2, n) ]
```

$$[341, 561, 645, 1105, 1387, 1729, 1905, 2047, 2465, 2701, 2821, 3277, 4033, 4369, 4371, 4681]$$

Even if you have never written any code, you can surely figure out what's going on!

Heads-up! The improved primality test discussed today will reduce this list to just $2047, 3277, 4033, 4681$.

The Miller–Rabin primality test

Review. The congruence $x^2 \equiv 1 \pmod{p}$ has only the solutions $x \equiv \pm 1$.
By contrast, if n is composite (and odd), then $x^2 \equiv 1 \pmod{n}$ has additional solutions.

The Miller–Rabin primality test exploits this difference to fix the issues of the Fermat primality test.

The Fermat primality test picks a and checks whether $a^{n-1} \equiv 1 \pmod{n}$.

- If $a^{n-1} \not\equiv 1 \pmod{n}$, then we are done because n is definitely not a prime.
- If $a^{n-1} \equiv 1 \pmod{n}$, then either n is prime or a is a Fermat liar.
  But instead of leaving off here, we can dig a little deeper:
  Note that $a^{(n-1)/2}$ satisfies $x^2 \equiv 1 \pmod{n}$. If n is prime, then $x \equiv \pm 1$ so that $a^{(n-1)/2} \equiv \pm 1 \pmod{n}$.
  - Hence, if $a^{(n-1)/2} \not\equiv \pm 1 \pmod{n}$, then we again know for sure that n is not a prime.
    Advanced comment. In fact, we can now factor n! See bonus challenge below.
  - If $a^{(n-1)/2} \equiv 1 \pmod{n}$ and $\frac{n-1}{2}$ is divisible by 2, we continue and look at $a^{(n-1)/4} \pmod{n}$.
    - If $a^{(n-1)/4} \not\equiv \pm 1 \pmod{n}$, then n is not a prime.
    - If $a^{(n-1)/4} \equiv 1 \pmod{n}$ and $\frac{n-1}{4}$ is divisible by 2, we continue...

Write $n - 1 = 2^s \cdot m$ with m odd. In conclusion, if n is a prime, then

$$a^m \equiv 1 \quad \text{or, for some } r = 0, 1, \ldots, s-1, \quad a^{2^r m} \equiv -1 \pmod{n}.$$

In other words, if n is a prime, then the values $a^m, a^{2m}, \ldots, a^{2^s m}$ must be of the form $1, 1, \ldots, 1$ or $\ldots, -1, 1, 1, \ldots, 1$. If the values are of this form even though n is composite, then a is a **strong liar** modulo n.

This gives rise to the following improved primality test:

Miller–Rabin primality test

Input: number n and parameter k indicating the number of tests to run
Output: "not prime" or "likely prime"
Algorithm:

Write $n - 1 = 2^s \cdot m$ with m odd.
Repeat k times:
  Pick a random number a from $\{2, 3, \ldots, n-2\}$.
  If $a^m \not\equiv 1 \pmod{n}$ and $a^{2^r m} \not\equiv -1 \pmod{n}$ for all $r = 0, 1, \ldots, s-1$, then
    stop and output "not prime".
Output "likely prime".

Comment. If n is composite, then fewer than a quarter of the values for a can possibly be strong liars. In other words, for all composite numbers, the odds that the Miller–Rabin test returns "likely prime" are less than 4^{-k}.

Comment. Note that, though it looks more involved, the Miller–Rabin test is essentially as fast as the Fermat primality test (recall that, to compute a^{n-1}, we proceed using binary exponentiation).

Advanced comments. This is usually implemented as a probabilistic test. However, assuming GRH (the generalized Riemann hypothesis), it becomes a deterministic algorithm if we check $a = 2, 3, ..., \lfloor 2(\log n)^2 \rfloor$. This is mostly of interest for theoretical applications. For instance, this then becomes a polynomial time algorithm for checking whether a number is prime.

More recently, in 2002, the AKS primality test was devised. This test is polynomial time (without relying on outstanding conjectures like GRH).

Example 105. Suppose we want to determine whether $n = 221$ is a prime. Simulate the Miller–Rabin primality test for the choices $a = 24$, $a = 38$ and $a = 47$.

Solution. $n - 1 = 4 \cdot 55 = 2^s \cdot m$ with $s = 2$ and $m = 55$.

- For $a = 24$, we compute $a^m = 24^{55} \equiv 80 \not\equiv \pm 1 \pmod{221}$. We continue with $a^{2m} \equiv 80^2 \equiv 212 \not\equiv -1$, and conclude that n is not a prime.

  Note. We do not actually need to compute that $a^{n-1} = a^{4m} \equiv 81$, which features in the Fermat test and which would also lead us to conclude that n is not prime.

- For $a = 38$, we compute $a^m = 38^{55} \equiv 64 \not\equiv \pm 1 \pmod{221}$. We continue with $a^{2m} \equiv 64^2 \equiv 118 \not\equiv -1$ and conclude that n is not a prime.

  Note. This case is somewhat different from the previous in that 38 is a Fermat liar. Indeed, $a^{4m} \equiv 118^2 \equiv 1 \pmod{221}$. This means that we have found a nontrivial sqareroot of 1. In this case, the Fermat test would have failed us while the Miller–Rabin test succeeds.

- For $a = 47$, we compute $a^m = 47^{55} \equiv 174 \not\equiv \pm 1 \pmod{221}$. We continue with $a^{2m} \equiv 174^2 \equiv -1$. We conclude that n is a prime or a is a strong liar. In other words, we are not sure but are (incorrectly) leaning towards thinking that 221 was likely a prime.

Comment. In this example, only 4 of the 218 residues $2, 3, ..., 219$ are strong liars (namely $21, 47, 174, 200$). For comparison, there are 14 Fermat liars (namely $18, 21, 38, 47, 64, 86, 103, 118, 135, 157, 174, 183, 200, 203$). [Note that ± 1 are Fermat as well as strong liars, too. However, these are usually excluded when testing.]

Example 106. In Example 98, we saw that all $\phi(561) = 320$ invertible residues a modulo 561 are Fermat liars (that is, they all satisfy $a^{560} \equiv 1 \pmod{561}$). How many strong liars are there?

Solution. There are 10 strong liars in total: $\pm 1, \pm 50, \pm 101, \pm 103, \pm 256$.

In particular, only 8 of the 558 residues $2, 3, ..., 559$ are strong liars. That's about 1.43% (much less than the theoretic bound of 25%).

(bonus challenge) For which $N < 1000$ is the proportion of strong liars the highest?

Here (as illustrated in the case of 561 above) we define the proportion of strong liars to be the proportion of residues among $2, 3, ..., N - 2$, which are strong liars.

[That proportion is almost 23%, just shy of the theoretical bound of 25%.]

Send in a solution by next week for a bonus point!

Extra excursion on Mersenne primes

Example 107. In 12/2018, a new largest (proven) prime was found: $2^{82,589,933}-1$.

https://www.mersenne.org/primes/?press=M82589933

This is a **Mersenne prime** (like the last 17 record primes). It has a bit over 24.8 million (decimal) digits (versus 23.2 for the previous record). The prime was found as part of GIMPS (Great Internet Mersenne Prime Search), which offers a $3,000 award for each new Mersenne prime discovered.

The EFF (Electronic Frontier Foundation) is offering $150,000 (donated anonymously for that specific purpose) for the discovery of the first prime with at least 100 million decimal digits.

https://www.eff.org/awards/coop

[Prizes of $50,000 and $100,000 for primes with 1 and 10 million digits have been claimed in 2000 and 2009.]

Definition 108. A **Mersenne prime** is a prime of the form 2^n-1.

For instance. The first few Mersenne primes have exponents $2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, \ldots$ All of these exponents are primes (but not all primes work: for instance, $2^{11}-1=23\cdot 89$). See below.

Anecdote. Euler proved in 1772 that $2^{31}-1$ is prime (then, and until 1867, the largest known prime).

"$2^{31}-1$ is probably the greatest [Mersenne prime] that ever will be discovered; for as they are merely curious, without being useful, it is not likely that any person will attempt to find one beyond it." — P. Barlow, 1811

https://en.wikipedia.org/wiki/2,147,483,647

Mersenne primes give rise precisely to all even perfect numbers (numbers whose proper divisors sum to the number itself; for instance, 6 is perfect because $6=1+2+3$). Indeed, Euclid showed that, if 2^p-1 is prime, then $2^{p-1}(2^p-1)$ is perfect [$p=2$: $2\cdot 3=6$, $p=3$: $4\cdot 7=28=1+2+4+7+14$, $p=5$: $16\cdot 33=528$, ...]. It is not known whether odd perfect numbers exist.

Example 109. (geometric sum) Evaluate $1+x+x^2+\ldots+x^n$.

Solution. $(1+x+x^2+\ldots+x^n)(x-1)=x^{n+1}-1$, so that $1+x+x^2+\ldots+x^n=\frac{x^{n+1}-1}{x-1}$.

Geometric series. In particular, $\sum_{k=1}^{\infty} x^k=\lim_{n\to\infty}\frac{x^{n+1}-1}{x-1}=\frac{1}{1-x}$, provided that $|x|<1$.

Lemma 110. If $r \mid n$, then $x^r-1 \mid x^n-1$.

Proof. Indeed, we have $x^n-1=(x^r-1)(1+x^r+x^{2r}+\ldots+x^{n-r})$.

Comment. For a tiny bit more detail, write $n=rs$. It follows from $x^s-1=(x-1)(1+x+x^2+\ldots+x^{s-1})$ that $x^{rs}-1=(x^r-1)(1+x^r+x^{2r}+\ldots+x^{r(s-1)})$. □

Corollary 111. 2^n-1 can only be prime if n is prime.

Proof. It follows from the previous lemma that, if $n=rs$ is composite, then 2^n-1 is divisible by 2^r-1 (as well as 2^s-1). □

For instance. $2^6-1=63$ is divisible by both $2^2-1=3$ and $2^3-1=7$.

Example 112. (bonus challenge) If $a^{n-1} \equiv 1 \pmod{n}$ but $a^{(n-1)/2} \not\equiv \pm 1 \pmod{n}$, then we can find a factor of n! How?!

For instance. $a = 38$ and $n = 221$ in Example 105.

Comment. However, note that this only happens if a is a Fermat liar modulo n, and these are typically very rare. So, unfortunately, we have not discovered an efficient factorization algorithm. [But we have run into an idea which is used for some of the best known factorization algorithms. If time permits, more on that later...]

Send in a solution by next week for a bonus point!

Block ciphers (and DES in particular)

We now introduce block ciphers at the example of **DES** (short for data encryption standard).

This sketch only provides an overview but does not include all details. See Chapter 4 in our book for these internals and detailed diagrams.

DES was the first public cryptosystem. While a public standard, the design decisions have been kept secret.

1974: proposed by IBM (lead by Horst Feistel; Lucifer) with input from NSA (key size reduced from 128 to 56 bits)

1976–2000: US national standard (broken by exhaustive search in 1997)

2000: replaced with AES (Rijndael) by NIST; however, **3DES** still considered secure (more later)

Why was the design secret? For many years, a particular mystery about DES was the choice of the S-boxes. Much later, in 1990, Biham and Shamir discovered **differential cryptanalysis**, a general method for breaking block ciphers. Surprisingly, it turned out that the particular choice of S-boxes made DES rather resistant against that attack. Indeed, as confirmed later, the IBM researchers had already discovered and anticipated that attack, but were asked by the NSA to keep it secret (it was a powerful weapon against other cryptosystems).

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

Comment. As our discussion will show, DES was designed to be implemented in hardware.

General principles of block cipher design

A block cipher takes a plaintext block of, say, B bits and encrypts it into a ciphertext block of B bits.

For instance, for DES, $B = 64$: 64 bit blocks are encrypted to 64 bit blocks.

For now, we will just focus on encrypting a single block.

However, we will need to talk about how to use a block cipher to encrypt longer plaintexts that need to be broken into many blocks (it is generally a bad idea to individually and independently encrypt each block).

The design of a block cipher is almost an art, but there are two guiding principles due to Claude Shannon, the father of information theory:

- confusion

refers to making the relationship between the ciphertext and the key as complex and involved as possible

(for instance, changing one bit of the key should change the ciphertext completely)

For instance. In DES, confusion is increased by the S-box substitutions. These are the only nonlinear part of DES. Without them, DES would be easily broken with linear algebra.

- diffusion

refers to dissipating the statistical structure of plaintext over the bulk of ciphertext

(for instance, changing one bit of the plaintext should change the ciphertext completely; likewise, changing one bit of the ciphertext should change the plaintext completely)

For instance. In DES, diffusion is increased by the E-box and P-box permutations.

Example 113. The classical substitution cipher provides only confusion.

Diffusion is completely missing. Changing bits of the plaintext only changes corresponding parts of the ciphertext. That's why frequency analysis can break these ciphers so easily.

Typical block ciphers are built by iteration, and consist of several **rounds**. Each round should have steps to increase both confusion and diffusion.

- **(key expansion)** First, we need to expand key k into several **round keys** $k_1, k_2, ..., k_n$ (n rounds).
  For instance. For DES, each round key k_i has 48 bits, which are drawn from the 56 bit DES key k in such a way that each bit of k shows up in about 14 of the 16 rounds.
- **(round functions)** Then, the message m is encrypted successively with R_{k_1}, R_{k_2}, ..., R_{k_n} to obtain c in the end.

$$m \to \boxed{R_{k_1}} \to \boxed{R_{k_2}} \to \ldots \to \boxed{R_{k_n}} \to c$$

  Each R_k is called a **round function**.
  For instance. For DES, there are $n = 16$ rounds; for AES-128, there are $n = 10$ rounds

A specific block cipher now needs specific algorithms for key expansion and round functions.

For DES, 16 rounds are used, which are identical in functionality but use different round keys k_i.

There is one additional, cryptographically irrelevant, step for DES: namely, there is a (fixed) **initial permutation** IP, which shuffles the bits of m before being sent to R_{k_1}. Similarly, the output of $R_{k_{16}}$ is shuffled with IP^{-1}, the inverse permutation, to produce c. (For the exact permutation see Chapter 4.4 in our book.)

Why? When implemented in hardware, this permutation does not cost any work, since it is just a wiring of the bits. In fact, the permutation somehow simplified the electrical engineering in the chips of the 70s.

A block cipher design: Feistel ciphers

Many ciphers, including DES (but not AES) are Feistel ciphers. This means that the encryption functions R_{k_i} are of a special format. The crucial ingredient is a **round function** $f_{k_i}(x)$.

This round function can be **any** function, such that x and $f_{k_i}(x)$ have the same size in bits (though only good choices will provide security). Also, several different round functions can be used for the different rounds.

To encrypt m using R_{k_i} (for DES, m is 64 bits and the round key k_i is 48 bits):

- Split the plaintext m into two halves (L_0, R_0) (for DES, each half is 32 bits).
- $L_1 = R_0$
  $R_1 = L_0 \oplus f_{k_i}(R_0)$
- Then, $R_{k_i}(m)$ is (L_1, R_1).

Example 114. How to decrypt one round? That is, how to obtain (L_0, R_0) from (L_1, R_1)?

Solution. First, $R_0 = L_1$. Then, $L_0 = R_1 \oplus f_{k_i}(R_0)$.

Important comment. In particular, we can take any round function f in the sense that we obtain some cipher, which can actually be decrypted (however, most choices for f will be insecure; see example below).

Comment. In hardware, the circuit for decryption is the same as for encryption, just reversed.

Example 115. What happens if we choose $f_{k_i}(R) = 0$ as the round function?

Solution. In that case, we are just swapping left and right half. No security whatsoever.

To finish the description of DES, we need to specify $f_{k_i}(R)$, where R is 32 bits and k_i is 48 bits.

We did that in class, but do not reproduce the description and diagrams here. See Chapter 4.4 of our book.
The crucial ingredients are:

- an E-box (expansion; expands 32 input bits into 48 output bits by repeating some),
- eight S-boxes (substitution; lookup tables that for each 6 bit input specify a 4 bit output),
- and a P-box (permutation; permutes 32 input bits to produce 32 output bits).

Further comments on DES

The S-boxes $S_1, S_2, ..., S_8$ are lookup tables (for each 6 bit input, they specify a 4 bit output).

- They have been carefully designed.
  For instance, their design already anticipated and protected against differential cryptanalysis (which wasn't publicly known at the time).
- On the other hand, they do not follow any simple rule. In particular, they must not be linear (or close to it). If they were, DES would be entirely insecure.
  [Slightly more specifically, if the S-boxes were linear, then the encryption map $m \mapsto c$ would be linear. In the usual spirit of linear algebra, a few (m, c) pairs would then suffice to recover the key.]
- They are also designed so that if one bit is changed in the input, then at least 2 bits of the output change.
  Important consequence. Go through one application of the round function $f_{k_i}(R)$, and convince yourself that flipping one bit of R has the effect of flipping at least two bits of $f_{k_i}(R)$. Repeating this for 16 rounds, you can see how the goal of diffusion seems to be achieved: changing one bit of the plaintext should change the ciphertext completely.

Example 116. Sometimes it is stated that DES works with a 64 bit key size. In that case, every 8th bit is a parity bit, but the algorithm really operates with 56 bit keys.

Comment. Apparently, the NSA was interested in strengthening DES against any attack (recall that developments like differential cryptanalysis were foreseen) except brute-force. Indeed, the NSA seems to have pushed for a key size of 48 bits versus proposed 64 bits, and the result was a compromise for 56 bits.

Example 117. If DES is insecure because of its 56 bit key size, why not just increase that?

Solution. DES was designed specifically for that key size. Increasing it necessitates a completely new analysis on how to choose the S-boxes and so on.

On the other hand. See the upcoming discussion of 3DES for how to leverage the original DES to increase the key size.

However. With the advent of powerful successors like AES there are very few reasons to use 3DES for new cryptosystems. (One slight advantage of 3DES is its particularly small footprint in hardware implementations.)

Example 118. Can we (easily) break DES if we know one of the round keys?

Solution. Absolutely! Recall that each round key consists of 48 bits taken from the overall 56 bit DES key. Hence, we know all but 8 bits of the key. We just need to brute-force these $2^8 = 256$ many possibilities.

Example 119. To (naively) brute-force DES, how much data must we encrypt?

Solution. By brute-forcing, we mean that, given a pair of 64-bit blocks m, c, we go through all 2^{56} possibilities (DES uses 56-bit keys) for k and look for k such that $E_k(m) = c$? We need to encrypt 2^{56} times 64 bits.

This is $2^{56} \cdot 8 = 2^{59}$ byte, or 512 pebibyte (binary analog of petabyte) or 576 petabyte (since $2^{59} \approx 5.76 \cdot 10^{17}$).

How long will this take? Of course, this depends on your machine. Assume we are able to encrypt 1 GB/sec. Then, this will take us about $5.76 \cdot 10^8$ sec, or about 18.3 years.

Of course, such a brute-force attack can be fully parallelized to quickly bring this number down to less than an hour for a powerful attacker. Also, the attack can be sped up considerably by careful design (like early aborts).

For comparison. Though mostly of theoretical value, for DES, some possibilities for attacks better than brute-force are known: for instance, as of 2008, linear cryptanalysis can mount an attack with 2^{43} known plaintexts in about 2^{40} (instead of 2^{56}) steps.

Example 120. (bonus challenge) Using DES, are there blocks m, c such that $E_k(m) = c$ for more than one key k?

This would mean that, using just a single plaintext-ciphertext pair, the above brute-forcing might uncover more than one possible key.

[I don't know the answer (but expect that it is "yes") and couldn't find it easily. Maybe you are more skilled?]

Example 121. (3DES) A simple approach to increasing the key size of DES, without the need to design and analyze a new block cipher, is **3DES**. It consists of three applications of DES to each block and is still considered secure.

$$c = E_{k_3}(D_{k_2}(E_{k_1}(m)))$$

The 3DES standard allows three keying options:

- k_1, k_2, k_3 independent keys: $3 \times 56 = 168$ key size, but effective key size is 112
- $k_1 = k_3$: $2 \times 56 = 112$ key size, effective key size is stated as 80 by NIST
- $k_1 = k_2 = k_3$: this is just the usual DES, and provides backwards compatibility (which is a major reason for making the middle step a decryption instead of another encryption).

Comment. The reason for the reduced effective key sizes is the meet-in-the-middle attack. It is also the reason why something like 2DES is not used. See next example!

Comment. NIST approved 3DES until 2030 for sensitive government data.

Example 122. (no 2DES) Explain why "2DES" does not really provide extra security over DES.

Solution. Let's denote DES encryption with E_k and decryption with D_k. The keys k are 56 bits.

Then, 2DES encrypts according to $c = E_{k_2}(E_{k_1}(m))$. The key size of 2DES is $56 + 56 = 112$ bits.

- A brute-force attack would go through all possibilities for pairs (k_1, k_2), of which there are $2^{56} \cdot 2^{56} = 2^{112}$, to check whether $c = E_{k_2}(E_{k_1}(m))$. That requires 2^{112} 2DES computations.
- On the other hand, note that $c = E_{k_2}(E_{k_1}(m))$ is equivalent to $D_{k_2}(c) = E_{k_1}(m)$.

  Assuming sufficient memory, we first go through all 2^{56} keys k_2 and store the values $D_{k_2}(c)$ in a lookup table.

  We then go through all 2^{56} keys k_1, compute $E_{k_1}(m)$ and see if we have stored that value before. (Even though this is a huge table, the cost for checking whether an element is in the table can be disregarded; thanks to the magic of hash tables!)

  Comment. In this second step, we see that m and c should be more than one block (otherwise we get too many candidate keys $k = (k_1, k_2)$).

  The total number of DES computations to break 2DES therefore is $2^{56} + 2^{56} = 2^{57}$, which is hardly more than for breaking DES!

Armin Straub
straub@southalabama.edu

This is known as a meet-in-the-middle attack.

https://en.wikipedia.org/wiki/Meet-in-the-middle_attack

Comment. The price to pay is that this attack also requires memory for storing This sort of approach is referred to as a time-memory trade-off. Instead of brute-forcing 2DES in 2^{112} steps, we can attack it in 2^{57} steps while storing 2^{56} values of the size of m.

Comment. This applies to any block cipher, not just DES!

Comment. For some block ciphers it is the case that for all pairs of keys k_1, k_2, there is a third key k_3 such that $E_{k_2}(E_{k_1}(m)) = E_{k_3}(m)$. In that case, we say that the cipher is a group, and double (or triple, or quadruple) encryption does not add any additional security! DES, however, is not a group.

Example 123. Explain why 3DES, used with three different keys, only has effective key size 112.

Solution. (fill in the details!) Instead of going through all k_1, k_2, k_3 to check whether

$$c = E_{k_3}(D_{k_2}(E_{k_1}(m)))$$

(which would take $2^{56} \cdot 2^{56} \cdot 2^{56} = 2^{168}$ DES computations), we can use that the latter is equivalent to

$$D_{k_3}(c) = D_{k_2}(E_{k_1}(m)).$$

Now proceed as in the previous example ... to see that we can break 3DES with $\sim 2^{112}$ DES computations. How much memory do we need?

Example 124. (extra; use as PRG) ANSI X9.17 is a U.S. federal standard for a PRG based on 3DES.

Input: random, secret 64 bit seed s, key k for 3DES (keying option 2)

Produce a random number as follows:

- obtain current time D, compute $t = \mathrm{3DES}_k(D)$
- output $x = \mathrm{3DES}_k(s \oplus t)$ (that's the pseudo-random output)
- update the seed to $s = \mathrm{3DES}_k(x \oplus t)$ for future use

Comment. ANSI (American National Standards Institute) X9 are standards for the financial industry.

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

Comment. The same approach can be applied to any block cipher.

Comment. It is common practice to add in time for PRGs that are used to generate enormous amounts of data. If nothing else, it slightly increases the entropy and reduces the likelihood of "short" periods.

Example 125. (extra; DES-X) To increase the key size of DES, the following variation, known as DES-X, was proposed by Ron Rivest in 1984:

$$c = k_3 \oplus \mathrm{DES}_{k_2}(m \oplus k_1)$$

What is the key size of DES-X? What about the effective key size?

Solution. k_1 and k_3 are 64 bit, while k_2 is 56 bits. That's a total key size of 184 bits for DES-X.

However, just like for 3DES, proceeding as in a meet-in-the-middle-attack (without the need of much storage) reduces the effective key size to at most $184 - 64 = 120$ bits.

Comment. This approach of xoring with a subkey before and after everything else is known as **key whitening**. This features in many modern ciphers, including AES.

https://en.wikipedia.org/wiki/DES-X

Block cipher modes

Block ciphers encrypt blocks of a specified size (64 bit for DES, or 128 bit for AES). **Block cipher modes** specify how to encrypt larger plaintexts.

Let E_k be the encryption routine of a block cipher with block size n bit. As a first step, we split a plaintext m into blocks $m = m_1 m_2 m_3 \ldots$ such that each m_i is n bits (we may have to pad).

Example 126. (ECB, shouldn't be used) In the simplest mode, known as **electronic codebook**, we just encrypt each plaintext block individually:

$$c_j = E_k(m_j)$$

The ciphertext is $c = c_1 c_2 c_3 \ldots$. Decryption simply computes $D_k(c_j) = m_j$.

Though natural, ECB has several severe weaknesses. Can you think of some?

Solution. Using ECB is nothing else but a classical substitution cipher, except that ECB operates on larger blocks. Just like a classical substitution cipher is vulnerable to frequency attacks, ECB leaves patterns in the ciphertext. For a striking visual example when encrypting a picture, see:

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

If a block repeats later in the message (or in a later message), it will be encrypted the same way. Hence, Eve can notice such repetitions. This is problematic in practice, for instance, because certain files always begin with the same blocks, so that Eve has a good chance of detecting the file type.

Also, knowing the filetype, Eve might be able to rearrange the ciphertext blocks to adjust the message. She can also attempt to delete certain ciphertext blocks.

Conclusion. Unless you know exactly why (e.g. sending already randomized messages), you should not use ECB.

Example 127. (CBC) In **cipherblock chaining** mode, we encrypt each plaintext block after chaining it with the previous cipherblock; that is:

$$c_j = E_k(m_j \oplus c_{j-1})$$

In order to do that for $j = 1$, we need a value for c_0, known as an **initialization vector** IV.

The ciphertext is $c = c_0 c_1 c_2 c_3 \ldots$ (that's one more block than for the plaintext $m = m_1 m_2 m_3 \ldots$).

(a) How does decryption work?

(b) Why should the value IV be unpredictable (e.g. be chosen randomly)?

Solution.

(a) Since $c_j = E_k(m_j \oplus c_{j-1})$, we have $D_k(c_j) = m_j \oplus c_{j-1}$ or $m_j = D_k(c_j) \oplus c_{j-1}$.
For instance. $m_1 = D_k(c_1) \oplus c_0$

(b) The value IV should be unique, so that messages starting with the same plaintext block have different ciphertext blocks. More generally, it should be unpredictable so that Eve cannot mount a chosen-plaintext attack to test if an earlier plaintext equals her guess. See Example 129.

Just checking. What would happen if we set $c_j = E_k(m_j) \oplus c_{j-1}$ instead?

[In that case, we would gain nothing over ECB: since Eve knows all c_j, she can compute $c_j \oplus c_{j-1} = E_k(m_j)$.]

Comment. CBC makes random access possible during decryption (but not encryption). That means, we don't need to decrypt $c = c_0 c_1 c_2 c_3 \ldots$ sequentially but can directly decrypt $c_N c_{N+1} \ldots$ for some random N.

Example 128. Consider the (silly) block cipher with 4 bit block size and 4 bit key size such that

$$E_k(b_1b_2b_3b_4) = (b_2b_3b_4b_1) \oplus k.$$

(a) Encrypt $m = (0000\ 1011\ 0000\ ...)_2$ using $k = (1111)_2$ and ECB mode.

(b) Encrypt $m = (0000\ 1011\ 0000\ ...)_2$ using $k = (1111)_2$ and CBC mode ($\mathrm{IV} = (0011)_2$).

Solution. $m = m_1m_2m_3...$ with $m_1 = 0000$, $m_2 = 1011$ and $m_3 = 0000$.

(a) $c_1 = E_k(m_1) = 0000 \oplus 1111 = 1111$
$c_2 = E_k(m_2) = 0111 \oplus 1111 = 1000$
Since $m_3 = m_1$, we have $c_3 = c_1$. Hence, the ciphertext is $c = c_1c_2c_3... = (1111\ 1000\ 1111\ ...)$.

(b) $c_0 = 0011$
$c_1 = E_k(m_1 \oplus c_0) = E_k(0000 \oplus 0011) = E_k(0011) = 0110 \oplus 1111 = 1001$
$c_2 = E_k(m_2 \oplus c_1) = E_k(1011 \oplus 1001) = E_k(0010) = 0100 \oplus 1111 = 1011$
$c_3 = E_k(m_3 \oplus c_2) = E_k(0000 \oplus 1011) = E_k(1011) = 0111 \oplus 1111 = 1000$
Hence, the ciphertext is $c = c_0c_1c_2c_3... = (0011\ 1001\ 1011\ 1000\ ...)$.

Comment. Clearly, our cipher is not meant to be secure. One damning issue (besides the short key and block size) is that it is linear (in both the plaintext and the key).

Extra. In each case, can you decrypt c to get back the original m?

Example 129. (BEAST attack) BEAST is short for Browser Exploit Against SSL/TLS and was brought to public attention in 2011. The attack is based on the fact that the IV used by SSL was obtained from a previous ciphertext block (instead of randomly!):

Scenario. Imagine that plaintext blocks $m_1m_2...$ are continuously being encrypted using CBC to cipherblocks. However, the plaintexts are from different parties and Eve can ask for her own plaintexts to be encrypted along the way.

In such a scenario, different plaintexts should be separately encrypted using CBC, meaning that a new random IV should be chosen each time.

Eve's goal. Suppose Eve has observed the ciphertext blocks c_{j-1}, c_j and her goal is to find out whether $m_j = x$ where x is her educated guess. Obviously, this is something that Eve should not be able to do!

The exploit. Because the IV for the next encryption is c_j, and because Eve can interject plaintext blocks to be encrypted for her, she can ask for $m_{j+1} = x \oplus c_{j-1} \oplus c_j$ (these are all known to Eve!) to be encrypted next.

Because CBC with IV c_j is used, this results in $c_{j+1} = E_k(m_{j+1} \oplus c_j) = E_k(x \oplus c_{j-1})$.

Eve can now compare this with $E_k(m_j \oplus c_{j-1}) = c_j$ (which she knows!) to find out whether $m_j = x$.

https://en.wikipedia.org/wiki/Transport_Layer_Security#BEAST_attack

There exist many other modes, including modes which already include features like authentication. Other common basic modes such as OFB (output feedback) or CTR (counter) turn the block cipher into a stream cipher (one advantage of that is that we don't need to encrypt full blocks at a time).

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

Comment. One issue of ECB and CBC is the need for padding. If not handled properly, this can be exploited by a **padding oracle attack**:

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

Example 130. (bonus challenge!) Find the smallest (pseudo)prime with 100 decimal digits, all of which are 3 or 7.

(Send me an email by next week with the prime, and how you found it, to collect a bonus point. Earn an extra bonus point if you can find it using a single line of Sage code [artificial concatenations not allowed].)

AES

Finite fields

Example 131. We have already seen xor in several cryptosystems. Note that a single xor operation as in the one-time pad or stream ciphers provides no diffusion.

When designing a cipher it may be nice to replace xor of N bit blocks with an operation that does provide some diffusion.

- A tiny amount of diffusion is provided by instead using addition modulo 2^N.
  Due to carries, one bit flip in the input can propagate to more than one bit flipped in the output.
- More diffusion can be achieved using operations (multiplication/inversion) in finite fields like $\mathrm{GF}(2^N)$.
  [We only need to make sure in our design that we don't multiply with zero.]

A **field** is a set of elements which can be added/subtracted as well as multiplied/divided by according to the usual rules.

In particular, a field always has distinguished elements 0 and 1, which are the neutral elements with respect to addition and multiplication, respectively.

Example 132.

- The rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$, and the complex numbers $\mathbb{C}$ all are fields, which you have seen before. They contain infinitely many elements.
- The integers $\mathbb{Z}$ are not a field because, for instance, 3 is not invertible (since $\frac{1}{3}$ is not an integer itself). Quotients of integers (rational numbers!) are a field.
  Since addition/subtraction and multiplication work as they should, $\mathbb{Z}$ is what is called a **ring**.
- Polynomials are not a field (they are a ring like $\mathbb{Z}$). Quotients of polynomials (rational functions!) are a field.

Cryptographic applications require finite structures. Correspondingly, our focus will be on **finite fields**, that is, fields consisting of only a finite number of elements.

Example 133. Let p be a prime. The residues modulo p form a field, often denoted as $\mathrm{GF}(p)$.

GF is short for **Galois field**, which is another word for finite field.

Note that we can divide by any element! (Except the zero residue but, of course, we can never divide by 0.)

Example 134. The residues modulo 21 (or any other composite number) are not a field.

We can add/subtract and multiply these numbers, but we cannot always divide. Specifically, we cannot divide by elements like $3, 6, 7, \ldots$ even though these are nonzero (we can, of course, never divide by zero).

Note. We have already seen that this seemingly slight deficiency has "terrible" consequences. For instance, the quadratic equation $x^2 = 1$ has more than the two solutions $x = \pm 1$ modulo 21 (namely, ± 8 as well).

AES is built upon byte operations (in contrast to DES, which is built on bit operations). Each of the 2^8 bytes represents one of the 2^8 elements of the finite field $\mathrm{GF}(2^8)$.

Note. We do not yet know what $\mathrm{GF}(2^8)$ is. It cannot be the residues modulo 2^8, because we just observed that the residues modulo n are a field only if n is prime.

To construct the finite field $\mathrm{GF}(p^n)$ of p^n elements, we can do the following:

- Fix a polynomial $m(x)$ of degree n, which is irreducible modulo p (i.e. cannot be factored modulo p).
- The elements of $\mathrm{GF}(p^n)$ are polynomials modulo $m(x)$ modulo p.

We will discuss the irreducibility condition on $m(x)$ next time. For now, see Example 137.

Comment. Actually, all finite fields can be constructed in this fashion. Moreover, choosing different $m(x)$ to construct $\mathrm{GF}(p^n)$ does not really matter: the resulting fields are always isomorphic (i.e. work in the same way, although the elements are represented differently). That justifies writing down $\mathrm{GF}(p^n)$, since there is exactly one such field.

Example 135. AES is based on representing bytes as elements of the field $\mathrm{GF}(2^8)$. It is constructed using the polynomial $x^8+x^4+x^3+x+1$ (which is indeed irreducible mod 2).

From bits to polynomials. For instance, the polynomial x^7+x^4+x corresponds to the bits 10010010 while x^6+1 corresponds to 01000001.

Example 136. The polynomial x^2+x+1 is irreducible modulo 2, so we can use it to construct the finite field $\mathrm{GF}(2^2)$ with 4 elements.

(a) List all 4 elements, and make an addition table. Then realize that this is just xor.

(b) Make a multiplication table.

(c) What is the inverse of $x+1$?

Solution.

(a) The four elements are $0, 1, x, x+1$.
For instance, $(x+1)+x=2x+1=1$ (in $\mathrm{GF}(2^2)$, since we are working modulo 2). The full table is below.
Each of the four elements is of the form $ax+b$, which can be represented using the two bits ab (for instance, $(10)_2$ represents x and $(11)_2$ represents $x+1$).
Then, addition of elements $ax+b$ in $\mathrm{GF}(2^2)$ works in the same way as xoring bits ab.

(b) For instance, $(x+1)^2=x^2+2x+1\equiv x^2+1\equiv(x+1)+1\equiv x$.
Here, the key is to realize that reducing modulo x^2+x+1 is the same as saying that $x^2=-x-1$, i.e. $x^2=x+1$ in $\mathrm{GF}(2^2)$. That means all polynomials of degree 2 and higher can be reduced to polynomials of degree less than 2.

$+$	0	1	x	$x+1$
0	0	1	x	$x+1$
1	1	0	$x+1$	x
x	x	$x+1$	0	1
$x+1$	$x+1$	x	1	0

$\times$	0	1	x	$x+1$
0	0	0	0	0
1	0	1	x	$x+1$
x	0	x	$x+1$	1
$x+1$	0	$x+1$	1	x

(c) We are looking for an element y such that $y(x+1)=1$ in $\mathrm{GF}(2^2)$. Looking at the table, we see that $y=x$ has that property. Hence, $(x+1)^{-1}=x$ in $\mathrm{GF}(2^2)$.

Example 137. What if we proceed as in the previous example but used $m(x)=x^2+1$ instead?

Solution. The addition table would be the same. The multiplication table would be different and a crucial difference would be that $(x+1)\cdot(x+1)=x^2+2x+1\equiv x^2+1\equiv 0$, which implies that $x+1$ cannot be invertible. That means our construction is not a field.

Comment. Note how, here, $m(x)$ factors modulo 2 as $x^2+1\equiv(x+1)(x+1)$. Hence the condition of irreducibility in the construction of $\mathrm{GF}(p^n)$ is violated.

Review. $\mathrm{GF}(p^n)$ is "the" finite field with p^n elements.

Recall that, in the construction of $\mathrm{GF}(p^n)$, the polynomial $m(x)$ has to be such that it cannot be factored modulo p. We also require that $m(x)$ needs to be **irreducible** mod p.

For instance. The polynomial x^2+2x+1 can always be factored as $(x+1)^2$.

On the other hand. For the polynomials $m(x)=x^2+x+1$ things are more interesting:

- x^2+x+1 cannot be factored over $\mathbb{Q}$ because the roots $\frac{-1\pm\sqrt{-3}}{2}$ are not rational.
- However, $x^2+x+1\equiv(x+2)^2$ modulo 3, so it can be factored modulo 3.
- On the other hand, x^2+x+1 is irreducible modulo 2 (that is, it cannot be factored: the only linear factors are x and $x+1$, but x^2, $x(x+1)$ and $(x+1)^2$ are all different from x^2+x+1 modulo 2).

In general, it follows from the formula $\frac{-1\pm\sqrt{-3}}{2}$ for the roots that x^2+x+1 can be factored modulo a prime $p>2$ if and only if $\sqrt{-3}$ exists as a residue modulo p. In other words, if and only if -3 is a quadratic residue modulo p.

For instance. Modulo $p=7$, we have $-3\equiv 2^2$ and $\frac{1}{2}\equiv 4$, so that $\frac{-1\pm\sqrt{-3}}{2}\equiv 4\cdot(-1\pm 2)\equiv 2,4$. Indeed, we have the factorization $(x-2)(x-4)=x^2-6x+8\equiv x^2+x+1$ modulo 7.

Example 138. The polynomial x^3+x+1 is irreducible modulo 2, so we can use it to construct the finite field $\mathrm{GF}(2^3)$ with 8 elements.

(a) List all 8 elements.

(b) Reduce x^5+1 in $\mathrm{GF}(2^3)$.

(c) Multiply each element of $\mathrm{GF}(2^3)$ with x^2+x.

(d) What is the inverse of x^2+x in $\mathrm{GF}(2^3)$?

Solution.

(a) The elements are $0,1,x,x+1,x^2,x^2+1,x^2+x,x^2+x+1$.
[Note that $x^3=-x-1=x+1$ in $\mathrm{GF}(2^3)$. That means all polynomials of degree 3 and higher can be reduced to polynomials of degree less than 3. See next part.]

(b) We divide x^5+1 by x^3+x+1 (long division!) to find $x^5+1=(x^2-1)(x^3+x+1)+(-x^2+x+2)$.
It follows that x^5+1 reduces to $-x^2+x+2\equiv x^2+x$ in $\mathrm{GF}(2^3)$.
Important. We can simplify things by performing the long division modulo 2. We then find $x^5+1\equiv(x^2+1)(x^3+x+1)+(x^2+x)$.

(c) We multiply the polynomials as usual, then reduce as in the previous part.
For instance, $(x^2+x)(x^2+x+1)\equiv x^4+x$ and, by long division, $x^4+x\equiv x(x^3+x+1)+x^2$, which reduces to just x^2 in $\mathrm{GF}(2^3)$.

$\times$	0	1	x	$x+1$	x^2	x^2+1	x^2+x	x^2+x+1
x^2+x	0	x^2+x	x^2+x+1	1	x^2+1	$x+1$	x	x^2

(d) We are looking for an element y such that $y(x^2+x)=1$ in $\mathrm{GF}(2^3)$. Looking at the table, we see that $y=x+1$ has that property. Hence, $(x^2+x)^{-1}=x+1$ in $\mathrm{GF}(2^3)$.
Important. To find the inverse, we essentially tried all possibilities. That's not sustainable. Instead, we can (and should!) proceed as we did for computing the inverse of residues modulo n. That is, we should use the Euclidean algorithm as indicated in the next examples. Here, this is just one step: modulo 2, we have $\boxed{x^3+x+1}\equiv(x+1)\cdot\boxed{x^2+x}+1$, so that $(x^2+x)^{-1}=x+1$ in $\mathrm{GF}(2^3)$.

Armin Straub
straub@southalabama.edu

Example 139.

(a) Apply the extended Euclidean algorithm to find the gcd of x^2+1 and x^4+x+1, and spell out Bezout's identity.

(b) Repeat the previous computation but always reduce all coefficients modulo 2.

(c) What is the inverse of x^2+1 in $\mathrm{GF}(2^4)$? Here, $\mathrm{GF}(2^4)$ is constructed using x^4+x+1.

Solution.

(a) We use the extended Euclidean algorithm:

$$\begin{aligned} &\gcd(x^2+1, x^4+x+1) && \boxed{x^4+x+1} = (x^2-1)\cdot\boxed{x^2+1} + (x+2) \\ &= \gcd(x+2, x^2+1) && \boxed{x^2+1} = (x-2)\cdot\boxed{x+2} + 5 \end{aligned}$$

Backtracking through this, we find that Bézout's identity takes the form

$$\begin{aligned} 5 \;=\; 1\cdot\boxed{x^2+1} - (x-2)\cdot\boxed{x+2} \;&=\; 1\cdot\boxed{x^2+1} - (x-2)\cdot\left(\boxed{x^4+x+1} - (x^2-1)\cdot\boxed{x^2+1}\right) \\ &=\; (x^3-2x^2-x+3)\cdot\boxed{x^2+1} - (x-2)\cdot\boxed{x^4+x+1} \end{aligned}$$

If we wanted to, we could divide both sides by 5.

(b) We repeat the exact same computation but reduce modulo 2 at each step:

$$\begin{aligned} \boxed{x^4+x+1} &\equiv (x^2+1)\cdot\boxed{x^2+1} + x \\ \boxed{x^2+1} &\equiv x\cdot\boxed{x} + 1 \end{aligned}$$

Backtracking through this, we find that Bézout's identity takes the form

$$\begin{aligned} 1 \;=\; 1\cdot\boxed{x^2+1} + x\cdot\boxed{x} \;&=\; 1\cdot\boxed{x^2+1} + x\cdot\left(\boxed{x^4+x+1} + (x^2+1)\cdot\boxed{x^2+1}\right) \\ &=\; (x^3+x+1)\cdot\boxed{x^2+1} + x\cdot\boxed{x^4+x+1} \end{aligned}$$

(c) We can now read off that $(x^2+1)^{-1} = x^3+x+1$ in $\mathrm{GF}(2^4)$.

Example 140. (HW) Find the inverses of x^2+1 and x^3+1 in $\mathrm{GF}(2^8)$, constructed as in AES.

Solution. Recall that for AES, $\mathrm{GF}(2^8)$ is constructed using $x^8+x^4+x^3+x+1$.

(a) We use the extended Euclidean algorithm for polynomials, and reduce all coefficients modulo 2:

$$\boxed{x^8+x^4+x^3+x+1} \equiv (x^6+x^4+x)\cdot\boxed{x^2+1} + 1$$

Hence, $(x^2+1)^{-1} = x^6+x^4+x$ in $\mathrm{GF}(2^8)$.

(b) We use the extended Euclidean algorithm, and always reduce modulo 2:

$$\begin{aligned} \boxed{x^8+x^4+x^3+x+1} &\equiv (x^5+x^2+x+1)\cdot\boxed{x^3+1} + x^2 \\ \boxed{x^3+1} &\equiv x\cdot\boxed{x^2} + 1 \end{aligned}$$

Backtracking through this, we find that Bézout's identity takes the form

$$\begin{aligned} 1 \;\equiv\; 1\cdot\boxed{x^3+1} - x\cdot\boxed{x^2} \;&\equiv\; 1\cdot\boxed{x^3+1} - x\cdot\left(\boxed{x^8+x^4+x^3+x+1} - (x^5+x^2+x+1)\cdot\boxed{x^3+1}\right) \\ &\equiv\; (x^6+x^3+x^2+x+1)\cdot\boxed{x^3+1} + x\cdot\boxed{x^8+x^4+x^3+x+1}. \end{aligned}$$

Hence, $(x^3+1)^{-1} = x^6+x^3+x^2+x+1$ in $\mathrm{GF}(2^8)$.

Armin Straub
straub@southalabama.edu

Basics of AES

The block cipher AES (short for **advanced encryption standard**) replaced DES. By now, it is the most important symmetric block cipher.

1997: NIST requests proposals for AES (receives 15 submissions) [very different from how DES was selected!]

2000: Rijndael (by Joan Daemen and Vincent Rijmen) selected (from 5 finalists)

https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf

- 128 bit block size (as per NIST request)
- The key size of AES can be 128, 192 or 256 bit. The corresponding choices are referred to as AES-128, AES-192 and AES-256, and have 10, 12 and 14 rounds, respectively.
- AES-192/256 is first (and only) public cipher allowed by NSA for top secret information.
- No known attacks on AES which are substantially better than brute-force.

Attacks better than brute-force known if the number of rounds was 6 (instead of 10) for AES-128.

- Unlike DES, AES is not a Feistel network.

While for a Feistel network, each round only encrypts half of the bits, all bits are being encrypted during each round. That's one indication why AES requires fewer rounds than DES.

Internals of AES

Each round consists of 4 **layers**. Each layer takes 128 bits input and outputs 128 bits in a reversible way (so that we can decrypt as long as we know the key).The 128 bit state consists of 16 bytes. These 16 bytes $c_{0,0}, c_{1,0}, c_{2,0}, c_{3,0}, c_{0,1}, \ldots, c_{3,3}$ are often arranged in a 4x4 matrix as

$$\begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix}.$$

Each byte is identified with an element of $\mathrm{GF}(2^8)$.

Example 141. $(0000\ 0101)_2$ represents the element x^2+1 in $\mathrm{GF}(2^8)$.

The 4 layers are:

- ByteSub

  each byte gets substituted with another byte (like a single S-box in DES); provides confusion and guarantees non-linearity of AES

- ShiftRow

  the 16 bytes are permuted (like a P-box in DES but on bytes, not bits); provides diffusion

- MixCol

  the 4x4 matrix is linearly transformed; provides diffusion

- AddRoundKey

  the state is xored with a 128 bit round key

Slight deviations. Before the first round, AddRoundKey is applied with the 0th round key (which equals the AES key). Otherwise, our first step would be ByteSub, which wouldn't have any cryptographic effect since the plaintext bytes would just be changed in a fixed manner (no key involved yet).

Also, the last round has no MixCol layer. This has the effect that decryption can be made to look very much like encryption (see Section 5.3 in our book for the details).

ShiftRow

The layer **ShiftRow** permutes the 16 bytes $c_{0,0}, c_{1,0}, c_{2,0}, c_{3,0}, c_{0,1}, \ldots, c_{3,3}$ as follows:

$$\begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \mapsto \begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,1} & c_{1,2} & c_{1,3} & c_{1,0} \\ c_{2,2} & c_{2,3} & c_{2,0} & c_{2,1} \\ c_{3,3} & c_{3,0} & c_{3,1} & c_{3,2} \end{bmatrix}$$

MixCol

Again, arrange the 16 bytes as a 4×4 matrix with entries in $\mathrm{GF}(2^8)$. The **MixCol** layer transform this 4×4 matrix by multiplying it with another, fixed, 4×4 matrix:

$$\begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \mapsto \begin{bmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{bmatrix} \begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix}$$

Example 142. For instance, the new byte at the position of $c_{2,1}$ (third row, second col) is

$$\begin{bmatrix} 1 & 1 & x & x+1 \end{bmatrix} \begin{bmatrix} c_{0,1} \\ c_{1,1} \\ c_{2,1} \\ c_{3,1} \end{bmatrix} = c_{0,1} + c_{1,1} + x c_{2,1} + (x+1) c_{3,1},$$

where all computations are to be done in $\mathrm{GF}(2^8)$.

AddRoundKey

The **AddRoundKey** layer simply xors the current 128 bit state with a 128 bit round key.

The **key schedule** for AES-128 is as follows. Like for the states, arrange the original 16 byte AES key k in a 4×4 matrix with columns $W(0), W(1), W(2), W(3)$.

The ith round key is then obtained from the matrix with columns $W(4i), W(4i+1), W(4i+2), W(4i+3)$, where $W(4), W(5), \ldots$ are recursively constructed:

$$W(i) = \begin{cases} W(i-4)+W(i-1), & \text{if } 4\nmid i, \\ W(i-4)+\tilde{W}(i-1), & \text{if } 4 \mid i. \end{cases}$$

Here, $\tilde{W}(i-1)$ is obtained from $W(i-1)$ as follows:

$$W(i-1) = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} \quad \Longrightarrow \quad \tilde{W}(i-1) = \begin{bmatrix} S(w_1)+x^{(i-4)/4} \\ S(w_2) \\ S(w_3) \\ S(w_4) \end{bmatrix}.$$

Note that w_1, w_2, w_3, w_4 each are bytes. The function S is the ByteSub substitution. Since that substitution is nonlinear, the round keys are constructed from k in a nonlinear manner (unlike in DES).

As usual, the computation $S(w_1)+x^{(i-4)/4}$ happens in $\mathrm{GF}(2^8)$.

ByteSub

The **ByteSub** layer takes each of the 16 bytes y and replaces it with the byte $S(y)$. As in DES, we could simply describe the (invertible) map by a lookup table. However, like the other steps of AES, it has a very simple mathematical description which we'll discuss next time.

Comment. As for the S-boxes in DES, ByteSub can be implemented in hardware as a lookup table. Since we have $2^8 = 256$ inputs, with 1 byte of output each, this table is 256 bytes large. (See Table 5.1 in our book.)

For comparison, each of the eight S-boxes in DES occupies 2^6 times 4 bits, which is 32 bytes. In total, these are also 256 bytes.

[In contrast to DES it is the case (and necessary for decryption!) that different inputs have different outputs.]

Example 143. (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 1 as a digit.

Send me the prime, and an explanation how you found it, by next week 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 every base, there always are just 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

- Recall that, in contrast to DES, the operations of AES have very simple (though somewhat advanced) mathematical descriptions.

  No mysteriously constructed S-boxes and P-boxes as in DES.

ByteSub (continued)

Each of the 16 bytes gets substituted as follows.

Note. The mathematical description below can be implemented in a **lookup table**: you can find this table in Table 5.1 of our book or, for instance, on wikipedia: https://en.wikipedia.org/wiki/Rijndael_S-box

- Interpret the input byte $(b_7b_6...b_0)_2$ as the element $b_7x^7+...+b_1x+b_0$ of $\mathrm{GF}(2^8)$.
- Compute $s^{-1}=c_0+c_1x+...+c_7x^7$ (with 0^{-1} interpreted as 0).

  Important comment. This inversion is what makes AES highly nonlinear.

  If the ByteSub substitution was linear, then all of AES would be linear (because all other layers are linear; assuming we adjust the key schedule accordingly).

- Then the output bits $(d_7d_6...d_1d_0)_2$ are

$$\begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Comment. The particular choice of matrix and vector has the effect that no ByteSub output equals the ByteSub input (or its complement).

Example 144. Invert x^3+1 in $\mathrm{GF}(2^8)$, constructed as in AES. [Example 140, again]

Solution. We use the extended Euclidean algorithm, and always reduce modulo 2:

$$\begin{aligned} \boxed{x^8+x^4+x^3+x+1} &\equiv (x^5+x^2+x+1)\cdot\boxed{x^3+1}+x^2 \\ \boxed{x^3+1} &\equiv x\cdot\boxed{x^2}+1 \end{aligned}$$

Backtracking through this, we find that Bézout's identity takes the form

$$\begin{aligned} 1 \equiv 1\cdot\boxed{x^3+1}-x\cdot\boxed{x^2} &\equiv 1\cdot\boxed{x^3+1}-x\cdot\left(\boxed{x^8+x^4+x^3+x+1}-(x^5+x^2+x+1)\cdot\boxed{x^3+1}\right) \\ &\equiv (x^6+x^3+x^2+x+1)\cdot\boxed{x^3+1}+x\cdot\boxed{x^8+x^4+x^3+x+1}. \end{aligned}$$

Hence, $(x^3+1)^{-1}=x^6+x^3+x^2+x+1$ in $\mathrm{GF}(2^8)$.

Example 145. (homework)

(a) What happens to the byte $(0000\ 0101)_2$ during ByteSub?

(b) What happens to the byte $(0000\ 1001)_2$ during ByteSub?

Armin Straub
straub@southalabama.edu

Solution.

(a) $(0000\ 0101)_2$ represents the polynomial x^2+1.

By Example 140, its inverse is $(x^2+1)^{-1} = x^6+x^4+x$ in $\mathrm{GF}(2^8)$, which is $\boldsymbol{c} = (0101\ 0010)_2$.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \underbrace{\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}}_{\boldsymbol{c}} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

[This is just the usual matrix-vector product modulo 2. The highlighted columns are the ones which get added up during this matrix-vector product.]

Hence, the output of ByteSub is the byte $(0110\ 1011)_2$.

Check with lookup tables. Indeed, our computation matches $107 = (0110\ 1011)_2$ in the lookup table in our book (row 0, column $(0101)_2 = 5$) or $(6\mathrm{B})_{16} = (0110\ 1011)_2$ on wikipedia (row $(0000)_2 = (0)_{16}$, column $(0101)_2 = (5)_{16}$).

(b) $(0000\ 1001)_2$ represents the polynomial x^3+1.

By Example 140 or 144, $(x^3+1)^{-1} = x^6+x^3+x^2+x+1$ in $\mathrm{GF}(2^8)$, which is $\boldsymbol{c} = (0100\ 1111)_2$.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}}_{\boldsymbol{c}} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Hence, the output of ByteSub is the byte $(0000\ 0001)_2$.

Check with lookup tables. Indeed, our computation matches the value 1 in the lookup table in our book (row 0, column $(1001)_2 = 9$) or $(01)_{16}$ on wikipedia (row $(0000)_2 = (0)_{16}$, column $(1001)_2 = (9)_{16}$).

Review: multiplicative order and primitive roots

Definition 146. The **multiplicative order** of an invertible residue a modulo n is the smallest positive integer k such that $a^k \equiv 1 \pmod{n}$.

Important note. By Euler's theorem, the multiplicative order can be at most $\phi(n)$.

Example 147. What is the multiplicative order of $2 \pmod 7$?

Solution. $2^1 = 2$, $2^2 = 4$, $2^3 \equiv 1 \pmod 7$. Hence, the multiplicative order of $2 \pmod 7$ is 3.

Definition 148. If the multiplicative order of an residue a modulo n equals $\phi(n)$ [in other words, the order is as large as possible], then a is said to be **primitive root** modulo n.

A primitive root is also referred to as a **multiplicative generator** (because the products of a and itself, that is, $1, a, a^2, a^3, \ldots$, produce all invertible residues).

Example 149. What is the multiplicative order of $3 \pmod 7$?

Solution. $3^1 = 3$, $3^2 \equiv 2$, $3^3 \equiv 6$, $3^4 \equiv 4$, $3^5 \equiv 5$, $3^6 \equiv 1$. Hence, the multiplicative order of $3 \pmod 7$ is 6. This means that 3 is a primitive root modulo 7. Note how every (invertible) residue shows up as a power of 3.

Review. $x \pmod n$ is a primitive root.

$\iff$ The (multiplicative) order of $x \pmod n$ is $\phi(n)$. (That is, the order is as large as possible.)

$\iff$ $x, x^2, \ldots, x^{\phi(n)}$ is a list of all invertible residues modulo n.

Lemma 150. If $a^r \equiv 1 \pmod n$ and $a^s \equiv 1 \pmod n$, then $a^{\gcd(r,s)} \equiv 1 \pmod n$.

Proof. By Bezout's identity, there are integers x, y such that $xr + ys = \gcd(r,s)$.
Hence, $a^{\gcd(r,s)} = a^{xr+ys} = a^{xr}a^{ys} = (a^r)^x(a^s)^y \equiv 1 \pmod n$. □

Corollary 151. The multiplicative order of a modulo n divides $\phi(n)$.

Proof. Let k be the multiplicative order, so that $a^k \equiv 1 \pmod n$. By Euler's theorem $a^{\phi(n)} \equiv 1 \pmod n$. The previous lemma shows that $a^{\gcd(k,\phi(n))} \equiv 1 \pmod n$. But since the multiplicative order is the smallest exponent, it must be the case that $\gcd(k, \phi(n)) = k$. Equivalently, k divides $\phi(n)$. □

Comment. By the same argument, if $a^m \equiv 1 \pmod n$, then the order of $a \pmod n$ divides m.

Example 152. Compute the multiplicative order of 2 modulo $7, 11, 9, 15$. In each case, is 2 a primitive root?

Solution.

- $2 \pmod 7$: $2^2 \equiv 4, 2^3 \equiv 1$. Hence, the order of 2 modulo 7 is 3.
  Since the order is less than $\phi(7) = 6$, 2 is not a primitive root modulo 7.
- $2 \pmod{11}$: Since $\phi(11) = 10$, the only possible orders are $2, 5, 10$. Hence, checking that $2^2 \not\equiv 1$ and $2^5 \not\equiv 1$ is enough to conclude that the order must be 10.
  Since the order is equal to $\phi(11) = 10$, 2 is a primitive root modulo 11.
  Brute force approach (too much unnecessary work). Just for comparison, $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 \equiv 5, 2^5 \equiv 2 \cdot 5 = 10, 2^6 \equiv 2 \cdot 10 \equiv 9, 2^7 \equiv 2 \cdot 9 \equiv 7, 2^8 \equiv 2 \cdot 7 \equiv 3, 2^9 \equiv 2 \cdot 3 = 6, 2^{10} \equiv 2 \cdot 6 \equiv 1$. Thus, the order of 2 mod 11 is 10.
- $2 \pmod 9$: Since $\phi(9) = 6$, the only possible orders are $2, 3, 6$. Hence, checking that $2^2 \not\equiv 1$ and $2^3 \not\equiv 1$ is enough to conclude that the order must be 6. (Indeed, $2^2 \equiv 4$, $2^3 \equiv 8$, $2^4 \equiv 7$, $2^5 \equiv 5$, $2^6 \equiv 1$.)
  Since the order is equal to $\phi(9) = 6$, 2 is a primitive root modulo 9.
- The order of $2 \pmod{15}$ is 4 (a divisor of $\phi(15) = 8$).
  2 is not a primitive root modulo 15. In fact, there is no primitive root modulo 15.

Comment. It is an open conjecture to show that 2 is a primitive root modulo infinitely many primes. (This is a special case of Artin's conjecture which predicts much more.)

Advanced comment. There exists a primitive root modulo n if and only if n is of one of $1, 2, 4, p^k, 2p^k$ for some odd prime p.

Example 153. Show that $x^4 \equiv 1 \pmod{15}$ for all invertible residues $x \pmod{15}$. In particular, there are no primitive roots modulo 15.

Solution. By the Chinese Remainder Theorem:

$x^4 \equiv 1 \pmod{15}$
$\iff$ $x^4 \equiv 1 \pmod 3$ and $x^4 \equiv 1 \pmod 5$

The congruences modulo 3 and 5 follow immediately from Fermat's little theorem.

Comment. The same argument shows that there are no primitive roots modulo pq, where p and q are distinct odd primes (because each element has order dividing $\phi(pq)/2$).

Armin Straub
straub@southalabama.edu

Lemma 154. Suppose $x \pmod n$ has (multiplicative) order k.

(a) $x^a \equiv 1 \pmod n$ if and only if $k|a$.

(b) x^a has order $\frac{k}{\gcd(k,a)}$.

Proof.

(a) "$\Longrightarrow$": By Lemma 150, $x^k \equiv 1$ and $x^a \equiv 1$ imply $x^{\gcd(k,a)} \equiv 1 \pmod n$. Since k is the smallest exponent, we have $k = \gcd(k,a)$ or, equivalently, $k|a$.
"$\Longleftarrow$": Obviously, if $k|a$ so that $a = kb$, then $x^a = (x^k)^b \equiv 1 \pmod n$.

(b) By the first part, $(x^a)^m \equiv 1 \pmod n$ if and only if $k|am$. The smallest such m is $m = \frac{k}{\gcd(k,a)}$. □

Example 155. Determine the orders of each (invertible) residue modulo 7. In particular, determine all primitive roots modulo 7.

Solution. First, observe that, since $\phi(7) = 6$, the orders can only be $1, 2, 3, 6$. Indeed:

residues	1	2	3	4	5	6
order	1	3	6	3	6	2

The primitive roots are 3 and 5.

Example 156. Redo Example 155, starting with the knowledge that 3 is a primitive root.

Solution.

residues	1	2	3	4	5	6
3^a	3^0	3^2	3^1	3^4	3^5	3^3
order$=\frac{6}{\gcd(a,6)}$	$\frac{6}{6}$	$\frac{6}{2}$	$\frac{6}{1}$	$\frac{6}{2}$	$\frac{6}{1}$	$\frac{6}{3}$

RSA and public key cryptography

- So far, our symmetric ciphers required a single **private key** k, a secret shared between the communicating parties.

  That leaves the difficult task of how to establish such private keys over a medium like the internet.

- In **public key cryptosystems**, there are two keys k_e, k_d, one for encryption and one for decryption. Bob keeps k_d secret (from anyone else!) and shares k_e with the world. Alice (or anyone else) can then send an encrypted message to Bob using k_e. However, Bob is the only who can decrypt it using k_d.

  It is crucial that the key k_d cannot be (easily) constructed from k_e.

RSA is one the first public key cryptosystems.

- It was described by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977. (Note the initials!)
- However, a similar system had already been developed in 1973 by Clifford Cocks for the UK intelligence agency GCHQ (classified until 1997). Even earlier, in 1970, his colleague James Ellis was likely the first to discover public key cryptography.

Example 157. Let us emphasize that it should be surprising that something like public key cryptography is even possible.

Imagine Alice, Bob and Eve sitting at a table. Everything that is being said is heard by all three of them. The three have never met before and share no secrets. Should it be possible in these circumstances that Alice and Bob can share information without Eve also learning about it?

Public key cryptography makes exactly that possible!

Comments on primitive roots

Example 158. Determine all primitive roots modulo 11.

Solution. Since $\phi(11)=10$, the possible orders of residues modulo 11 are $1,2,5,10$. Residues with order 10 are primitive roots. Our strategy is to find one primitive root and to use that to compute all primitive roots.
There is no good way of finding the first primitive root. We will just try the residues $2,3,5,\ldots$ (why not 4?!)
We compute the order of $2 \pmod{11}$:
Since $2^2=4\not\equiv 1$, $2^5\equiv -1\not\equiv 1 \pmod{11}$, we find that 2 has order 10. Hence, 2 is a primitive root.
All other invertible residues are of the form 2^x. Recall that the order of $2^x \pmod{11}$ is $\frac{10}{\gcd(10,x)}$.
Hence, 2^x is a primitive root if and only if $\gcd(10,x)=1$, which yields $x=1,3,7,9$.
In conclusion, the primitive roots modulo 11 are $2^1=2, 2^3=8, 2^7\equiv 7, 2^9\equiv 6$.

Example 159. (extra) Determine all primitive roots modulo 22.

Solution. We proceed as in the previous example:

- Since $\phi(22)=10$, the possible orders of residues modulo 22 are $1,2,5,10$.
- We find one primitive root by trying residues $3,5,\ldots$ (2 is out because it is not invertible modulo 22)
  Since $3^5\equiv 1 \pmod{22}$, 3 is not a primitive root modulo 22.
  Since $5^5\equiv 1 \pmod{22}$, 5 is not a primitive root modulo 22.
  Since $7^2\not\equiv 1$, $7^5\equiv -1\not\equiv 1 \pmod{22}$, 7 is a primitive root modulo 22.
- $7^x \pmod{22}$ has order $\frac{10}{\gcd(10,x)}$. We have $\gcd(10,x)=1$ for $x=1,3,7,9$.
- Hence, the primitive roots modulo 22 are $7^1=7, 7^3\equiv 13, 7^7\equiv 17, 7^9\equiv 19$.

Proceeding as in the previous example, we obtain the following result.

Theorem 160. (number of primitive roots) Suppose there is a primitive root modulo n. Then there are $\phi(\phi(n))$ primitive roots modulo n.

Proof. Let x be a primitive root. It has order $\phi(n)$. All other invertible residues are of the form x^a.
Recall that x^a has order $\frac{\phi(n)}{\gcd(\phi(n),a)}$. This is $\phi(n)$ if and only if $\gcd(\phi(n),a)=1$. There are $\phi(\phi(n))$ values a among $1,2,\ldots,\phi(n)$, which are coprime to $\phi(n)$.
In conclusion, there are $\phi(\phi(n))$ primitive roots modulo n. □

Comment. Recall that, for instance, there is no primitive root modulo 15. That's why we needed the assumption that there should be a primitive root modulo n (which is the case if and only if n is of the form $1,2,4,p^k,2p^k$ for some odd prime p).

In particular, since there are always primitive roots modulo primes, we have the following important case:

There are $\phi(\phi(p))=\phi(p-1)$ primitive roots modulo a prime p.

Example 161. (bonus challenge) For which prime $p<10^6$ is the proportion of primitive roots among invertible residues the smallest?

Send in a solution by next week for a bonus point!

Armin Straub
straub@southalabama.edu

Back to RSA

(RSA encryption)

- Bob chooses large random primes p, q.
- Bob chooses e, and then computes d such that $de \equiv 1 \pmod{(p-1)(q-1)}$.
- Bob makes $N = pq$ and e public. His (secret) private key is d.
- Alice encrypts $c = m^e \pmod{N}$.
- Bob decrypts $m = c^d \pmod{N}$.

Does decryption always work? What Bob computes is $c^d \equiv (m^e)^d = m^{de} \pmod{N}$. It follows from Euler's theorem and $de \equiv 1 \pmod{\phi(N)}$ that $m^{de} \equiv m \pmod{\phi(N)}$ for all invertible residues m. That this actually works for all residues can be seen from the Chinese Remainder Theorem (see Theorem 162 below).

Is that really secure? Well, if implemented correctly (we will discuss potential issues), RSA has a good track record of being secure. Next class, we will actually prove that finding the secret key d is as difficult as factoring N (which is believed, but has not been proven, to be hard). On the other hand, it remains an important open problem whether knowing d is actually necessary to decrypt a given message.

Comment. The $(p-1)(q-1)$ in the generation of d can be replaced with $\operatorname{lcm}(p-1, q-1)$. This will be illustrated in Example 166.

Theorem 162. Let $N = pq$ and d, e be as in RSA. Then, for any m, $m \equiv m^{de} \pmod{N}$.

Comment. Using Euler's theorem, this follows immediately for residues m which are invertible modulo N. However, it then becomes tricky to argue what happens if m is a multiple of p or q.

Proof. By the CRT, we have $m \equiv m^{de} \pmod{N}$ if and only if $m \equiv m^{de} \pmod{p}$ and $m \equiv m^{de} \pmod{q}$.
Since $de \equiv 1 \pmod{(p-1)(q-1)}$, we also have $de \equiv 1 \pmod{p-1}$. By little Fermat, it follows that $m^{de} \equiv m \pmod{p}$ for all $m \not\equiv 0 \pmod{p}$. On the other hand, if $m \equiv 0 \pmod{p}$, then this is obviously true. Thus, $m \equiv m^{de} \pmod{p}$ for all m. Likewise, modulo q. □

Example 163. Bob's public RSA key is $N = 33$, $e = 3$.

(a) Encrypt the message $m = 4$ and send it to Bob.

(b) Determine Bob's secret private key d.

(c) You intercept the message $c = 31$ from Alice to Bob. Decrypt it using the secret key.

Solution.

(a) The ciphertext is $c = m^e \pmod{N}$. Here, $c \equiv 4^3 = 64 \equiv 31 \pmod{33}$. Hence, $c = 31$.

(b) $N = 3 \cdot 11$, so that $\phi(N) = 2 \cdot 10 = 20$.
To find d, we need to compute $e^{-1} \pmod{20}$. Since the numbers are so simple we see $3^{-1} \equiv 7 \pmod{20}$.
Hence, $d = 7$.

(c) We need to compute $m = c^d \pmod{N}$, that is, $m = 31^7 \equiv (-2)^7 \equiv 4 \pmod{33}$.
That is, $m = 4$ (as we already knew from the first part).

Example 164. For his public RSA key, Bob needs to select p, q and e. Which of these must be chosen randomly?

Solution. The primes p and q must be chosen randomly. Anything that makes these primes more predictable, makes it easier for an attacker to get her hands on them [in which case, the secret key d is trivial to compute].
On the other hand, e does not need to be chosen at random. In fact, knowing any pair e, d such that $ed \equiv 1 \pmod{(p-1)(q-1)}$ would allow us to factor $N = pq$ (and thus break RSA). We'll prove that later.

Review. RSA

Example 165. If $N=77$, what is the smallest (positive) choice for e?

Solution. Technically, $e=1$ works but then we wouldn't be encrypting at all.
Note that e must be invertible modulo $\phi(N)=6\cdot 10=60$. Hence, $e=2,3,4,5,6$ are not allowed.
The smallest possible choice for e therefore is $e=7$.

Example 166. Bob's public RSA key is $N=33$, $e=13$. His private key is $d=17$.

(a) Explain how the decryption of, say, $c=26$ can be sped up using the CRT.

(b) Encrypt the message $m=4$ and send it to Bob. Compare with the example from last class where $N=33$, $e=3$.

(c) Bob's choice of $e=13$ is actually functionally equivalent to $e=3$ and, similarly, d can be obtained as $e^{-1} \pmod{10}$, resulting in $d=7$. Explain and generalize these claims!

(d) An RSA user is shocked by the previous part and exclaims "RSA is only half as secure as I thought...!" How shocked should we be?

Solution. Note that the private key is $d\equiv 13^{-1} \pmod{20}\equiv 17$.

(a) To decrypt, Bob needs to compute $m=c^d \pmod N$. Knowing that $N=pq=3\cdot 11$, we instead compute $c^d \pmod p$ and $c^d \pmod q$ [which is less work] and then use the CRT to recover $m \pmod N$.
Here, $26^{17}\equiv(-1)^{17}\equiv 2 \pmod 3$ and $26^{17}\equiv 4^{17}\equiv 4^7\equiv 4\cdot 4^2\cdot 4^4\equiv 4\cdot 5\cdot 3\equiv 5 \pmod{11}$.
Hence, $m=26^{17} \pmod{33}\equiv 2\cdot 11\cdot (11)^{-1}_{\bmod 3}+5\cdot 3\cdot (3)^{-1}_{\bmod 11}\equiv 22\cdot(-1)+15\cdot 4\equiv 5 \pmod{33}$.
Comment. Note that $(11)^{-1}_{\bmod 3}$ and $(3)^{-1}_{\bmod 11}$ can be precomputed and reused. In practice, using the CRT leads to about a 4-fold speed up.

(b) The ciphertext is $c=m^e \pmod N$. Here, $c\equiv 4^{13}\equiv\ldots\equiv 31 \pmod{33}$.
If $e=3$ instead, then $c\equiv 4^3=64\equiv 31 \pmod{33}$ so that we get the same ciphertext. See next item!

(c) If you look back at our proof of Theorem 162, you'll see that (again using the CRT) we only need $de\equiv 1 \pmod{(p-1)}$ and $de\equiv 1 \pmod{(q-1)}$ in order that $m^{de}\equiv m \pmod{pq}$.
So, instead of $d\equiv e^{-1} \pmod{(p-1)(q-1)}$, it is enough that $d\equiv e^{-1} \pmod{\operatorname{lcm}(p-1,q-1)}$.
Here, $\operatorname{lcm}(2,10)=10$, so that we only need $d=e^{-1} \pmod{10}$.

(d) It is definitely misleading that RSA is "half" as secure. It is indeed the case though that the key space for the secret key d is only half (or even less) as big as that RSA user initially thought.
However, that means that, for instance, if N is 2048 bit, then the secret key is one bit (possibly more) less than what the shocked RSA user expected. That hardly qualifies as "half as secure".
Comment. However, if $\operatorname{lcm}(p-1,q-1)$ is "too small", that is, $\gcd(p-1,q-1)$ is "too big" (so that we are loosing considerably more than 1 bit for the key size), then p, q should be discarded. If $\gcd(p-1,q-1)\approx 2^e$, then we are loosing about e bits for the key size.

Example 167. RSA is so cool! Why do we even care about, say, AES anymore?

Solution. RSA is certainly cool, but it is very slow (comparatively). As such, RSA is not practical for encrypting larger amounts of data. RSA is, however, perfect for sharing secret keys, which can then be used for encrypting data using, say, AES.

Armin Straub
straub@southalabama.edu

Example 168. Is it a problem that $m=1$ is always encrypted to $c=1$? (Likewise for $m=0$.)

Solution. Well, it would be a problem if we reply to questions using YES (say, 1) and NO (say, 0) and encrypt our reply. However, this would always be a terrible idea in any deterministic public key cryptosystem (that is, a system, in which a message gets encrypted in a single way)!

Why? That's because Eve can just encrypt both YES and NO (or any collection of expected messages) and see which matches the ciphertext she intercepted.

Important conclusion. We must not send messages taken from a small predictable set and encrypt them using a deterministic public key cryptosystem like RSA.

Once realized, this is easy to fix: for instance, Alice can just augment the plaintext with some random noise in such a way that Bob can discard that noise after decryption. This is done when RSA is used in practice.

Comment. This applies to any public key cryptosystem, in which a message gets encrypted in a single way. To avoid this issue, some randomness is typically introduced. For instance, for RSA, when used in practice, the plaintext would be padded with random noise before encryption. On the other hand, the ElGamal encryption we discuss next, has such randomness already built into it.

Comment. Note that this is not an issue with symmetric ciphers like DES or AES. In that case, even if the attacker knows that the plaintext must be one of "0" or "1", she still cannot draw any conclusions from intercepting the ciphertext.

Example 169. (extra) Bob's public RSA key is $N=55$, $e=7$.

(a) Encrypt the message $m=8$ and send it to Bob.

(b) Determine Bob's secret private key d.

(c) You intercept the message $c=2$ from Alice to Bob. Decrypt it using the secret key.

Solution.

(a) The ciphertext is $c=m^e \pmod N$. Here, $c\equiv 8^7 \pmod{55}$
$8^2\equiv 9$, $8^4\equiv 9^2\equiv 26$. Hence, $8^7=8^4\cdot 8^2\cdot 8\equiv 26\cdot 9\cdot 8\equiv 2 \pmod{55}$. Hence, $c=2$.

(b) $N=5\cdot 11$, so that $\phi(N)=4\cdot 10=40$.
To find d, we compute $e^{-1} \pmod{40}$ using the extended Euclidean algorithm:

$$\begin{aligned} & \gcd(7,40) && \boxed{40} &= 6\cdot\boxed{7}-2 \\ =\ & \gcd(2,7) && \boxed{7} &= 3\cdot\boxed{2}+1 \\ =\ & 1 \end{aligned}$$

Backtracking through this, we find that Bézout's identity takes the form

$$1=\boxed{7}-3\cdot\boxed{2}=\boxed{7}-3\cdot\left(6\cdot\boxed{7}-\boxed{40}\right)=-17\cdot\boxed{7}+3\cdot\boxed{40}.$$

Hence, $7^{-1}\equiv -17\equiv 23 \pmod{40}$ and, so, $d=23$.

Comment. Actually, as discussed in Example 166, $\phi(N)=(p-1)(q-1)=4\cdot 10$ can be replaced with $\operatorname{lcm}(p-1,q-1)=\operatorname{lcm}(4,10)=20$. It follows that the pair $(e,d)=(7,23)$ is equivalent to the pair $(e,d)=(7,3)$.

(c) We need to compute $m=c^d \pmod N$, that is, $m=2^{23} \pmod{55}$.
$2^2=4$, $2^4=16$, $2^8\equiv 36\equiv -19$, $2^{16}\equiv 19^2\equiv 31 \pmod{55}$. Hence, $2^{23}=2^{16}\cdot 2^4\cdot 2^2\cdot 2\equiv 31\cdot 16\cdot 4\cdot 2\equiv 8 \pmod{55}$.
That is, $m=8$ (as we already knew from the first part).
Comment. As noted above, $d=3$ is equivalent to $d=23$. Indeed, $m=2^3=8 \pmod{55}$.

The ElGamal public key cryptosystem and discrete logarithms

Whereas the security of RSA relies on the difficulty of factoring, the security of ElGamal and Diffie–Hellman relies on the difficulty of computing discrete logarithms.

Discrete logarithms

Suppose $b = a^x \pmod N$. Finding x is called the **discrete logarithm problem** mod N. If N is a large prime p, then this problem is believed to be difficult.

Note. If $b = a^x$, then $x = \log_a(b)$. Here, we are doing the same thing, but modulo N. That's why the problem is called the discrete logarithm problem.

Example 170. Find x such that $4 \equiv 3^x \pmod 7$.

Solution. We have seen in Example 155 that 3 is a primitive root modulo 7. Hence, there must be such an x. Going through the possibilities ($3^2 \equiv 2$, $3^3 \equiv 6$, $3^4 \equiv 4$), we find $x = 4$, because $3^4 \equiv 4 \pmod 7$.

Example 171. Find x such that $3 \equiv 2^x \pmod{101}$.

Solution. Let us check that the solution is $x = 69$. Indeed, a quick binary exponentiation confirms that $2^{69} \equiv 3 \pmod{101}$. (Do it!)

The point is that it is actually (believed to be) very difficult to compute these **discrete logarithms**. On the other hand, just like with factorization, it is super easy to verify the answer if somebody tells us the answer.

Comment. We can check that 2 is a primitive root modulo 101. That is, $2 \pmod{101}$ has (multiplicative) order 100. That means every equation $2^x \equiv a \pmod{101}$, where $a \not\equiv 0$, has a solution.

Diffie–Hellman key exchange

(Diffie–Hellman key exchange)

- Alice and Bob select a large prime p and a primitive root $g \pmod p$.
- Bob randomly selects a secret integer x and reveals $g^x \pmod p$ to everyone.
  Alice randomly selects a secret integer y and reveals $g^y \pmod p$ to everyone.
- Alice and Bob now share the secret $g^{xy} \pmod p$.
  Indeed, Alice can compute $g^{xy} = (g^x)^y$ using the public g^x and her secret y.
  Likewise, Bob can compute $g^{xy} = (g^y)^x$ using the public g^y and his secret x.

Why is this secure? We need to see why eavesdropping Eve cannot (simply) obtain the secret $g^{xy} \pmod p$. She knows g, g^x, $g^y \pmod p$ and needs to find $g^{xy} \pmod p$. This is the **computational Diffie–Hellman problem** (CDH), which is believed to be hard (it would be easy if we could compute discrete logarithms).

Example 172. You are Eve. Alice and Bob select $p = 53$ and $g = 5$ for a Diffie–Hellman key exchange. Alice sends 43 to Bob, and Bob sends 20 to Alice. What is their shared secret?

Solution. If Alice's secret is y and Bob's secret is x, then $5^y \equiv 43$ and $5^x \equiv 20 \pmod{53}$.
Since we haven't learned a better method, we just compute $5^2, 5^3, \ldots$ until we find 43 or 20:
$5^2 = 25$, $5^3 \equiv 19$, $5^4 \equiv 19 \cdot 5 \equiv -11$, $5^5 \equiv -11 \cdot 5 \equiv -2$, $5^6 \equiv -2 \cdot 5 \equiv -10 \equiv 43 \pmod{53}$.
Hence, Alice's secret is $y = 6$. The shared secret is $20^6 \equiv 9 \pmod{53}$.

Note. We don't need to find Bob's secret. [It is $x = 11$.]

Armin Straub
straub@southalabama.edu

ElGamal encryption

Proposed by Taher ElGamal in 1985

The original paper is actually very readable: https://dx.doi.org/10.1109/TIT.1985.1057074

(ElGamal encryption)

- Bob chooses a prime p and a primitive root $g \pmod p$.
  Bob also randomly selects a secret integer x and computes $h = g^x \pmod p$.
- Bob makes (p, g, h) public. His (secret) private key is x.
- To encrypt, Alice first randomly selects an integer y.
  Then, $c = (c_1, c_2)$ with $c_1 = g^y \pmod p$ and $c_2 = h^y m \pmod p$.
- Bob decrypts $m = c_2 c_1^{-x} \pmod p$.

Why does decryption work? $c_2 c_1^{-x} = (h^y m)(g^y)^{-x} = ((g^x)^y m)(g^y)^{-x} = m \pmod p$

More conceptually, the key idea (featured in Diffie–Hellman) that makes ElGamal encryption work is that Alice (her private secret is y) and Bob (his private secret is x) actually share a secret: g^{xy}

Note that encryption is just multiplying m with the shared secret $h^y = g^{xy}$. Likewise, decryption is division by the shared secret $c_1^x = g^{xy}$.

Comment. For ElGamal, the message space actually is $\{1, 2, ..., p-1\}$. $m = 0$ is not permitted.

That's, of course, no practical issue. For instance, we could simply identify $\{1, 2, ..., p-1\}$ with $\{0, 1, ..., p-2\}$ by adding/subtracting 1.

Comment. p and g don't have to be chosen randomly. They can be reused. In fact, it is common to choose p to be a "safe prime" (see next comment), with specific pre-selected choices listed, for instance, in RFC 3526.

Advanced comment. Note that in order to check whether g is a primitive root modulo p, we need to be able to factor $p-1$, which in general is hard (2 is an obvious factor, but other factors are typically large and, in fact, we need them to be large in order for the discrete logarithm problem to be difficult). It is therefore common to start with a prime n and then see if $2n+1$ is prime as well, in which case we select $p = 2n+1$. Such primes p [primes such that $(p-1)/2$ is prime, too] are called **safe primes** (more later).

On the other hand, g doesn't necessarily have to be a primitive root. However, we need the group generated by g (the elements $1, g, g^2, g^3, ...$) to be large. For more fancy cryptosystems, we can even replace these groups with other groups such as those generated by elliptic curves.

Example 173. Bob chooses the prime $p = 31$, $g = 11$, and $x = 5$. What is his public key?

Solution. Since $h = g^x \pmod p$ is $h \equiv 11^5 \equiv 6 \pmod{31}$, the public key is $(p, g, h) = (31, 11, 6)$.

Comment. Bob's secret key is $x = 5$. In principle, an attacker can compute x from $11^x \equiv 6 \pmod{31}$. However, this requires computing a discrete logarithm, which is believed to be difficult if p is large.

Example 174. Bob's public ElGamal key is $(p, g, h) = (31, 11, 6)$.

(a) Encrypt the message $m = 3$ ("randomly" choose $y = 4$) and send it to Bob.

(b) Determine Bob's private key from his public key.

(c) Using Bob's private key, decrypt $c = (9, 13)$.

Solution.

(a) The ciphertext is $c=(c_1,c_2)$ with $c_1=g^y \pmod p$ and $c_2=h^y m \pmod p$.
Here, $c_1=11^4\equiv 9 \pmod{31}$ and $c_2=6^4\cdot 3\equiv 13 \pmod{31}$. Hence, the ciphertext is $c=(9,13)$.

(b) To find Bob's secret key x, we need to solve $11^x\equiv 6 \pmod{31}$. This yields $x=5$.
(Since we haven't learned a better method, we just try $x=1,2,3,\dots$ until we find the right one.)
Comment. Alternatively, after having done the first part, we know that $m=c_2c_1^{-x} \pmod p$ takes the form $3=13\cdot 9^{-x} \pmod{31}$, which is equivalent to $9^x=13\cdot 3^{-1}\equiv 25 \pmod{31}$. While this also reveals $x=5$, there is an issue with this approach. Can you see it?
[The issue is that 9 (which is c_1 and could be anything) does not have to be a primitive root. In fact, 9 is not a primitive root modulo 31. Accordingly, $9^x\equiv 25 \pmod{31}$ does not have a unique solution: $x=20$ is another one (and does not correspond to Bob's private key).]

(c) We decrypt $m=c_2c_1^{-x} \pmod p$.
Here, $m=13\cdot 9^{-5}\equiv 3 \pmod{31}$.
Comment. One option is to compute $9^{-1}\equiv 7 \pmod{31}$, followed by $9^{-5}\equiv 7^5\equiv 5 \pmod{31}$ and, finally, $13\cdot 9^{-5}\equiv 13\cdot 5\equiv 3 \pmod{31}$. Another option is to begin with $9^{-5}\equiv 9^{25} \pmod{31}$ (by Fermat's little theorem).

Example 175. (extra) Bob's public ElGamal key is $(p,g,h)=(23,10,11)$.

(a) Encrypt the message $m=5$ ("randomly" choose $y=2$) and send it to Bob.

(b) Encrypt the message $m=5$ ("randomly" choose $y=4$) and send it to Bob.

(c) Break the cryptosystem and determine Bob's secret key.

(d) Use the secret key to decrypt $c=(8,7)$.

(e) Likewise, decrypt $c=(18,19)$.

Solution.

(a) The ciphertext is $c=(c_1,c_2)$ with $c_1=g^y \pmod p$ and $c_2=h^y m \pmod p$.
Here, $c_1=10^2\equiv 8 \pmod{23}$ and $c_2=11^2\cdot 5\equiv 6\cdot 5\equiv 7 \pmod{23}$. Hence, the ciphertext is $c=(8,7)$.

(b) Now, $c_1=10^4\equiv 18 \pmod{23}$ and $c_2=11^4\cdot 5\equiv 13\cdot 5\equiv 19 \pmod{23}$ so that $c=(18,19)$.

(c) To find Bob's secret key x, we need to solve $10^x\equiv 11 \pmod{23}$. This yields $x=3$.
(Since we haven't learned a better method, we just try $x=1,2,3,\dots$ until we find the right one.)

(d) We decrypt $m=c_2c_1^{-x} \pmod p$.
Here, $m=7\cdot 8^{-3}\equiv 7\cdot 4\equiv 5 \pmod{23}$, as we knew from the first part.
[$8^{-1}\equiv 3 \pmod{23}$, so that $8^{-3}\equiv 3^3\equiv 4 \pmod{23}$. Or, use Fermat: $8^{-3}\equiv 8^{19}\equiv 4 \pmod{23}$.]

(e) In this case, $m=19\cdot 18^{-3}\equiv 19\cdot 16\equiv 5 \pmod{23}$, as we knew from the second part.

Example 176. If Bob selects $p=23$ for ElGamal, how many possible choices does he have for g? Which are these?

Solution. g needs to be a primitive root modulo 23. Recall that, modulo a prime p, there are $\phi(\phi(p))=\phi(p-1)$ many primitive roots. Hence, Bob has $\phi(p-1)=\phi(22)=10$ choices for g.

Review. ElGamal encryption

- Like RSA, ElGamal is terribly slow compared with symmetric ciphers like AES.

  Encryption under ElGamal requires two exponentiations (slower than RSA); however, these exponentiations are independent of the message and can be computed ahead of time if need be (in that case, encryption is just a multiplication, which is much faster than RSA). Decryption only requires one exponentiation (like RSA).

- In contrast to RSA, ElGamal is randomized. That is, a single plaintext m can be encrypted to many different ciphertexts.

  A drawback is that the ciphertext is twice as large as the plaintext.

  On the positive side, an attacker who might be able to guess potential plaintexts cannot (as in the case of vanilla RSA) encrypt these herself and compare with the intercepted ciphertext.

Example 177. Does Alice have to choose a new y if she sends several messages to Bob using ElGamal encryption?

Solution. Yes, she absolutely has to randomly choose a new y every time! Here's why:

If she was using the same y to encrypt messages $m^{(1)}$ and $m^{(2)}$, Alice would be sending the ciphertexts $(c_1^{(1)}, c_2^{(1)}) = (g^y, g^{xy}m^{(1)})$ and $(c_1^{(2)}, c_2^{(2)}) = (g^y, g^{xy}m^{(2)})$.

That means, Eve can immediately figure out $c_2^{(1)} \,/\, c_2^{(2)} = m^{(1)} \,/\, m^{(2)}$ (the divison is a modular inverse and everything is modulo p). That's a combination of the plaintexts, and Eve should never be able to get her hands on such a thing.

(Note that Eve would know right away if Alice is doing the mistake of reusing y because $c_1^{(1)} = c_1^{(2)}$.)

Comment. The situation is just like for the one-time pad (in that case, reusing the key reveals $m^{(1)} \oplus m^{(2)}$).

The computational and decisional Diffie–Hellman problem

We indicated that the security of ElGamal depends on the difficulty of computing discrete logarithms. Here is a more precise statement.

Theorem 178. Obtaining m from c (without the private key) in ElGamal is exactly as difficult as the **computational Diffie–Hellman problem** (CDH).

The CDH problem is the following: given $g, g^x, g^y \pmod p$, find $g^{xy} \pmod p$. It is believed to be hard.

Proof. Recall that the public key is $(p, g, h) = (p, g, g^x)$. The ciphertext is $c = (g^y, h^y m) = (g^y, g^{xy}m)$.
Hence, determining m is equivalent to finding g^{xy}.
Since $g, g^x, g^y \pmod p$ are known, this is precisely the CDH problem. □

Armin Straub
straub@southalabama.edu

Example 179. In fact, even the **decisional Diffie–Hellman problem** (DDH) is believed to be difficult.

The DDH problem is the following: given g, g^x, g^y, $r \pmod p$, decide whether $r \equiv g^{xy} \pmod p$. Obviously, this is simpler than the CDH problem, where g^{xy} needs to be computed. Yet, it, too, is believed to be hard.

Comment. Well, at least it is hard (modulo p) if we always want to do better than guessing.

Here's how we can sometimes do better than guessing: if g^x or g^y is a quadratic residue (this is actually easy to check modulo primes p using Euler's criterion), then g^{xy} is a quadratic residue (why?!). Hence, if r is not a quadratic residue, we can conclude that $r \not\equiv g^{xy}$.

More on safe primes

Recall that p is a **safe prime** if both p and $(p-1)/2$ are prime. The next example illustrates why it is common to use safe primes for ElGamal.

In general, it is difficult to ensure that g is a primitive root, or almost a primitive root, modulo p.

Example 180. Suppose that p is a safe prime. Show that all residues $g \not\equiv 0, \pm 1 \pmod p$ have order $(p-1)/2$ or $p-1$.

In the latter case, g is a primitive root. In fact, if $p > 5$, then half of the residues $g \not\equiv 0, \pm 1$ are primitive roots.

Solution. Suppose $g \not\equiv 0, \pm 1 \pmod p$. Because p is a prime and $g \not\equiv 0$, g is invertible. Its multiplicative order N divides $\phi(p) = p-1$. But the prime factorization of $p-1$ is 2 times $(p-1)/2$. Hence, the only possible orders are $1, 2, (p-1)/2$ and $p-1$. The residues ± 1 are the only with order 1 and 2 (why?!). Thus, g must have order $(p-1)/2$ or $p-1$.

Finally, if $p > 5$ (with the property that $(p-1)/2$ is odd), note that the number of primitive roots is $\phi(p-1) = \phi(2)\phi((p-1)/2) = (p-3)/2$, which is exactly half of the residues g.

Advanced comment. Actually, it is easy to distinguish between the residues that have order $(p-1)/2$ and those that have order $p-1$. Recall that, if x has order $p-1$, then x^2 has order $\frac{p-1}{\gcd(p-1,2)} = \frac{p-1}{2}$. It follows that quadratic residues have order $(p-1)/2$ (provided that $x \not\equiv 0, \pm 1$). (And, using Euler's criterion, it is computationally easy to determine whether a residue modulo p is a quadratic residue or not.)

Example 181. Is there any advantage for RSA if p is a safe prime? Potential issues?

Solution. If p is a safe prime, then $\gcd(p-1, q-1) = 2$. Why?!

Hence, the key space is as large as possible.

On the other hand, we need to think about whether we are weakening the security in case we might severely limit the number of possible p's to choose from. [The prime number theorem tells us that this is not something we need to worry about. Can you spell out the details?]

Another issue is that generating random safe primes is considerably more work. On the other hand, Bob usually does not generate a public key frequently, so that this might not be much of an issue.

Further comments on RSA and ElGamal

Theorem 182. Determining the secret private key d in RSA is as difficult as factoring N.

Proof. Let us show how to factor $N=pq$ if we know e and d such that $d\equiv e^{-1} \pmod{(p-1)(q-1)}$.

- Write $ed-1=2^t m$, where t is chosen as large as possible such that 2^t divides $ed-1$.
  Since $ed-1\equiv 0 \pmod{(p-1)(q-1)}$ and 2^2 divides $(p-1)(q-1)$, we have $t\geqslant 2$.
- Pick a random invertible residue x. Observe that $x^{ed-1}\equiv 1 \pmod N$. In other words, $(x^m)^{2^t}\equiv 1$.
  Hence, the multiplicative order of x^m must divide 2^t.
- Suppose that x^m has different order modulo p than modulo q.

  Note. One can show that this works for at least half of the (invertible) residues x. As such, if we are unlucky, we just select another x.

  Since both orders must divide 2^t, we may suppose x^m has order 2^s modulo p, and larger order modulo q.
  Then, $x^{2^s m}\equiv 1 \pmod p$ but $x^{2^s m}\not\equiv 1 \pmod q$.
  Consequently, $\gcd(x^{2^s m}-1, N)=p$ so that we have found the factor p of N.

  Note. Of course, we don't know s (because we don't know p and q), but we can just go through all $s=0,1,2,\ldots,t-1$. One of these has to reveal the factor p. □

However. It is not known whether knowing d is actually necessary for Eve to decrypt a given ciphertext c. This remains an important open problem.

Advanced comment. Recall that we don't necessarily need to have $d\equiv e^{-1} \pmod{(p-1)(q-1)}$ but that it suffices to have the same with $(p-1)(q-1)$ replaced by $\operatorname{lcm}(p-1, q-1)$. This means that, in the above argument, we only get $t\geqslant 1$ but the rest of the argument still applies.

Example 183. (homework) Bob's public RSA key is $N=323$, $e=101$. Knowing $d=77$, factor N using the approach of the previous theorem.

Solution. Here, $de-1=7776=2^5\cdot 243$ so that $t=5$ and $m=243$.

- Let's pick $a=2$. $a^m=2^{243}\equiv 246 \pmod{323}$ must have order dividing 2^5.
  $\gcd(246^2-1, 323)=19$ (so we don't even need to check $\gcd(246^{2^s}-1, 323)$ for $s=2,3,4$)
  Hence, we have factored $N=17\cdot 19$.

Comment. Among the $\phi(323)=16\cdot 18=288$ invertible residues a, only 36 would not lead to a factorization. The remaining 252 residues all reveal the factor 19.

Another project idea. Run some numerical experiments to get a feeling for the number of residues that result in a factorization.

Definition 184. Bob's public key cryptosystem is **semantically secure** if Eve cannot do better than guessing in the following challenge:

- Bob determines a random public and private key. The public key is given to Eve.
- Eve selects two plaintexts m_1 and m_2.
- Alice flips a fair coin and, accordingly, using the public key encrypts m_1 or m_2 as c.
- Eve now needs to decide whether c is the encryption of m_1 or m_2.

For this definition to make precise mathematical sense, we need to assume that Eve's computing power is somehow limited (typically, she is limited to polynomial-time algorithms).

Comment. Also, many variations exist of what semantic security exactly is. All of these try to capture the idea that an attacker does not learn anything about m from knowing c. The one above is often referred to as IND-CPA (Indistinguishability under Chosen Plaintext Attack).

Important comment. Realize that semantic security is a very strong property to ask for! In particular, this is much stronger than what we usually think about in terms of security: you might call a cipher secure if it is "impossible" for an attacker to get m from c. Semantic security is requiring that an attacker gets so little information from c that she cannot even tell whether it came from (her own choices) m_1 or m_2.

Example 185. Is vanilla RSA semantically secure?

Solution. No. Eve can just encrypt both m_1 and m_2 herself, and compare with c. She then knows for sure which of the two was encrypted.

Comment. As mentioned before, in practice, RSA is never used in its vanilla (or "textbook") version (unless random plaintexts are encrypted). Instead, it is randomized (like ElGamal is by design) by padding the plaintext with random stuff.

Check out OAEP: https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding

The resulting RSA-OAEP has been proven semantically secure (under the "RSA assumption" that finding m from c is hard).

Example 186. Is ElGamal semantically secure?

Solution. Essentially, yes.

Recall that the public key is $(p, g, h) = (p, g, g^x)$.

The ciphertext is $(c_1, c_2) = (g^y, h^y m) = (g^y, g^{xy} m)$. Eve needs to decide whether the m in there is m_1 or m_2.

Equivalently, she needs to decide whether $r = c_2/m_1$ (or $r = c_2/m_2$) equals g^{xy} or not.

This is essentially the DDH problem.

Strictly speaking. Because of the issue with quadratic residues mentioned when we introduced the DDH problem, ElGamal is not semantically secure in the sense we defined things. However, if we wanted (this is more of a theoretical point), this issue could be fixed by not computing with all invertible residues modulo p, but only with quadratic residues. We could further select p to be a **safe prime**, meaning that $(p-1)/2$ is prime again, in which case all quadratic residues (except 1) have order $(p-1)/2$ (so that no similar games can be played using orders of elements).

Practical implications. Indeed, Diffie–Hellman and ElGamal in practice often use safe primes p. In that case, as we observed in Example 180, there are no elements of small order (besides 1 and -1). Since generating such primes can be a bit expensive, it is common to use preselected ones. For instance, RFC 3526 lists six such primes (together with a generator g) with $1536, 2048, \ldots, 8192$ bits.

https://www.ietf.org/rfc/rfc3526.txt

Important. It is perfectly fine that p and g are not random in Diffie–Hellman or ElGamal. However, it is absolutely crucial that x (and y) are random (generated using a cryptographically secure PRG).

Example 187. What is your feeling? Can we make RSA even more secure by allowing N to factor into more than 2, say, 3 primes?

Solution. That doesn't seem like a good idea. Namely, observe that the security of RSA relies on adversaries being unable to factor N. Allowing more factors of N (while keeping the size of N fixed) makes that task easier, because more factors means that the factors are necessarily smaller.

Example 188. RSA has proven to be secure so far. However, it is easy to implement RSA in such a way that it is insecure. One important but occasionally messed up part of RSA is that **p and q must be unpredictable**, and the only way to achieve that is to choose p, q completely randomly in some huge interval $[M_1, M_2]$.

- For instance, if $N = pq$ has m digits and we know the first (or last) $m/4$ digits of p, then we can efficiently factor N.

  An adversary might know many digits of p if, for instance, we make the mistake of generating the random prime p by considering candidates of the form $2^{1023} + k$ for small (random) values of k (2^{1023} was chosen so that the resulting number has 1024 bits).

- Also, we must use a cryptographically secure PRG to generate p and q.

  If using a "bad" PRG or choosing seeds with too little entropy, then (especially among a large number of public keys generated this way) it becomes likely that (different) public keys N and N' share a prime factor p. In that case, everybody can determine $p = \gcd(N, N')$ and break both public keys.

  Indeed. For instance, in a study of Lenstra et. al., millions of public keys were collected and compared. Among the RSA moduli, about 0.2% shared a common prime factor with another one. That's terrible: if (different) public keys N and N' share a prime factor p, then everybody can determine $p = \gcd(N, N')$ and break both public keys.

  http://eprint.iacr.org/2012/064.pdf

- In that direction, is the security of public key cryptosystems like RSA in any way compromised when used by tens of millions of users?

  As noted above, millions of people using "bad" PRGs for generating RSA public keys make it likely that this weakness can be practically exploited.

  Similarly, for Diffie–Hellman and ElGamal, it is common to use fixed primes p. While fine in principle, this may be an issue if used by millions of users faced against an adversary Eve with vast resources. See, for instance: https://threatpost.com/prime-diffie-hellman-weakness-may-be-key-to-breaking-crypto/

Example 189. (side-channel attacks) For instance, by measuring the time it takes to decrypt messages as $m = c^d \pmod N$ in RSA, Eve might be able to reconstruct the secret key d.

This **timing attack**, first developed by Paul Kocher (1997), is particularly unsettling because it illustrates that the security of a system can be compromised even if mathematically everything is sound. This sort of attack is called a **side-channel attack**. It attacks the implementation (software and/or hardware) rather than the cryptographic algorithm.

See Section 6.2.3 in our book for more details on how d can be obtained in this attack.

In a similar spririt, there exist power attacks (measuring power instead of time during decryption) or fault attacks (for instance, injecting errors during computations):

https://en.wikipedia.org/wiki/Side-channel_attack

How to prevent? Implement RSA in such a way that no inferences can be drawn from the time and power consumption.

Lesson. Do not implement crypto algorithms yourself!! Instead, use one of the well-tested open implementations.

It's kind of sad, isn't it? Don't come up with your own ciphers. Don't implement ciphers yourself...

But it is important to realize just how easy it is to implement these algorithms in such a way that security is compromised (even if the idea, intentions and algorithms are all sound and secure).

After advertising open implementations, let us end this discussion with a cautionary example in that regard.

Example 190. The following story made lots of headlines in 2016:

https://threatpost.com/socat-warns-weak-prime-number-could-mean-its-backdoored/116104/

After a year, it was noticed that, in the open-source tool Socat ("Netcat++"), the Diffie-Hellman key exchange was implemented using a hard-coded 1024 bit prime p (nothing wrong with that), which wasn't prime! Explain how this could be used as a backdoor.

Solution. The security of the Diffie-Hellman key exchange relies on the difficulty of taking discrete logarithms modulo p. If we can compute x in $h = g^x \pmod p$, then we can break the key exchange.

Now, if $p = p_1 p_2$, then we can use the CRT to find x by solving the two (much easier!) discrete logarithm problems

$$h = g^x \pmod{p_1}, \quad h = g^x \pmod{p_2}.$$

This is an example of a **NOBUS backdoor** ("nobody but us"), because the backdoor can only be used by the person who knows the (secret) factorization of p.

Comment. In the present case, the Socat "prime" p actually has the two small factors 271 and 13597, and $p/(271 \cdot 13597)$ is still not a prime (but nobody has been able to factor it). This might hint more at a foolish accident than a malicious act.

Important follow-up question. Of course, the issue has been fixed and the composite number has been replaced by the developers with a large prime. However, should we trust that it really is a prime?

We don't need to trust anyone because primality checking is simple! We can just run the Miller–Rabin test N times. If the number was composite, there is only a 4^{-N} chance of us not detecting it. (In OpenSSL, for instance, $N = 40$ and the chance for an error, 2^{-80}, is astronomically low.) Both Fermat and Miller–Rabin instantly detect the number here to be composite (for certain).

Comment. This illustrates both what's good and what's potentially problematic about open source projects. The potentially problematic part for crypto is that Eve might be among the people working on the project. The good part is that (hopefully!*) many experts are working on or looking into the code. Thus, hopefully, any malicious acts on Eve's part should be spotted soon (in fact, with proper code review, should never make it into any production version). Of course, this "hope" requires ongoing effort on the parts of everyone involved, and the willingness to fund such projects.

*However, sometimes very few people are involved in a project, despite it being used by millions of users. For instance, see: https://en.wikipedia.org/wiki/Heartbleed

Example 191. (short plaintext attack on RSA) Suppose a 56bit DES key (or any other short plaintext) is written as a number $m \approx 2^{56} \approx 10^{16.9}$ and encrypted as $c = m^e \pmod N$.

Eve makes two lists:

- $cx^{-e} \pmod N$ for $x = 1, 2, ..., 10^9$
- $y^e \pmod N$ for $y = 1, 2, ..., 10^9$

If there is a match between the lists, that is $cx^{-e} = y^e \pmod N$, then $c = (xy)^e \pmod N$ and Eve has learned that the plaintext is $m = xy$.

This attack will succeed if m is the product of two integers x, y (up to 10^9). This is the case for many integers m.

Another project idea. Quantify how many integers factor into two small factors.

How to prevent? To prevent this attack, the plaintext can be padded with random bits before being encrypted. Recall that we should actually never use vanilla RSA (unless with random plaintexts) and always use a securely padded version instead!

Example 192. For RSA, does double (or triple) encryption improve security?

(a) Say, if Bob asks people to send him messages first encrypted with a first public key (N, e_1) and then encrypted with a second public key (N, e_2).

(b) Or, what if Bob asks people to send him messages first encrypted with a first public key (N_1, e_1) and then encrypted with a second public key (N_2, e_2).

Solution.

(a) No, this does not result in any additional security.
After one encryption, $c_1 = m^{e_1} \pmod N$ and the final ciphertext is $c_2 = c_1^{e_2} \pmod N$. However, note that $c_2 = m^{e_1 e_2} \pmod N$, which is the same as encryption with the single public key $(N, e_1 e_2)$.

(b) This adds only a negligible bit of security and hence is a bad idea as well. The reason is that an attacker able to determine the secret key for (N_1, e_1) is likely just as able to determine the secret key for (N_2, e_2), meaning that the attack would only take twice as long (or two computers). That's only a tiny bit of security gained, somewhat comparable to increasing N from 1024 to 1025 bits. If heightened security is wanted, it is better to increase the size of N in the first place.
[Make sure you see how the situation here is different from the situation for 3DES.]

Example 193. (common modulus attack on RSA) Alice encrypts m using each of the RSA public keys $(N,\ e_1)$ and $(N,\ e_2)$ so that the ciphertexts are $c_1 = m^{e_1} \pmod N$ and $c_2 = m^{e_2} \pmod N$. Eve might be able to figure out m from c_1 and c_2!! How and when?

Solution. The crucial observation is that $c_1^x c_2^y \equiv m^{e_1 x} m^{e_2 y} = m^{e_1 x + e_2 y} \pmod N$. Eve can choose x and y.
She knows m if she can arrange x and y such that $e_1 x + e_2 y = 1$. This is possible if $\gcd(e_1, e_2) = 1$, in which case Eve would use the extended Euclidean algorithm to determine appropriate x and y.

A scenario. Bob's public RSA key is (N, e). However, when Alice requests this public key from Bob, her message gets intercepted by Eve who instead sends (N, e_2) back to Alice, where e_2 differs from e in only one bit. Alice uses (N, e_2) to encrypt her message and sends c_2 to Bob. Of course, Bob fails to decrypt Alice's message and so resends his public key to Alice (this time, Eve doesn't intervene). Alice now uses (N, e) to encrypt her message and sends c to Bob.
Since $e - e_2 = \pm 2^r$, we have $\gcd(e, e_2) = 1$ (why?!), so that Eve can determine m as explained above.

Comment on that scenario. From a practical point of view, we can argue that, if Eve can trick Alice into using a modified version of Bob's public key, then she might as well give a completely new public key (that Eve created) to Alice, in which case she can immediately decipher c_2. That's certainly true. However, that way, Eve's malicious intervention would be plainly visible as such.

Example 194. (chosen ciphertext attack on RSA) Show that RSA is not secure under a chosen ciphertext attack.

First of all, let us recall that in a chosen ciphertext attack, Eve has some access to a decryption device. In the present case, we mean the following: Eve is trying to determine m from c. Clearly, we cannot allow her to use the decryption device on c (because then she has m and nothing remains to be said). However, Eve is allowed to decrypt some other ciphertext c' of her choosing (hence, "chosen ciphertext").

You may rightfully say that this is a strange attacker who can decrypt messages except the one of particular interest. This model is not meant to be realistic—instead, it is important for theoretical security considerations: if our cryptosystem is secure against this (adaptive) version of chosen ciphertext attacks, then it is also secure against any other reasonable chosen ciphertext attacks.

Solution. RSA is not secure under a chosen ciphertext attack:

Suppose $c = m^e \pmod N$ is the ciphertext for m.

Then, Eve can ask for the decryption m' of $c' = 2^e c \pmod N$. Since $c' = (2m)^e \pmod N$, Eve obtains $m' \equiv 2m$, from which she readily determines $m = 2^{-1} m' \pmod N$.

Comment. On the other hand, RSA-OAEP is provably secure against chosen ciphertext attacks. Recall that, in this case, m is padded prior to encryption. As a result, $2m$ or, more generally am, is not going to be a valid plaintext.

Example 195. What we just exploited is that RSA is **multiplicatively homomorphic**.

Multiplicatively homomorphic means the following: suppose m_1 and m_2 are two plaintexts with ciphertexts c_1 and c_2. Then, (the residue) $m_1 m_2$ has ciphertext $c_1 c_2$.

[That is, multiplication of plaintexts translates to multiplication of ciphertexts, and vice versa. Mathematically, this means that the map $m \to c$ is a homomorphism (with respect to multiplication).]

Indeed, for RSA, $c_1 = m_1^e$ and $c_2 = m_2^e$, so that $c_1 c_2 = m_1^e m_2^e = (m_1 m_2)^e \pmod N$ is the ciphertext for $m_1 m_2$.

Why care? In our previous example, being multiplicatively homomorphic was a weakness of RSA (which is "cured" by RSA-OAEP). However, there are situations where homomorphic ciphers are of practical interest. With a homomorphic cipher, we can do calculations using just the ciphertexts without knowing the plaintexts (for instance, the ciphertexts could be encrypted (secret) votes, which could be publicly posted; then anyone could add up (in an additively homomorphic system) these votes into a ciphertext of the final vote count; the advantage being that we don't need to trust an authority for that count). The search for a fully **homomorphic encryption** scheme is a hot topic. For a nice initial read, you can find more at:

https://blog.cryptographyengineering.com/2012/01/02/very-casual-introduction-to-fully/

Example 196. (chosen ciphertext attack on ElGamal) Show that ElGamal is not secure under a chosen ciphertext attack.

Solution. Recall, again, that in a chosen ciphertext attack, Eve is trying to determine m from c and Eve has access to a decryption device, which she can use, except not to the ciphertext c in question.

Suppose $c = (c_1, c_2) = (g^y, g^{xy} m)$ is the ciphertext for m. Then $(c_1, 2c_2) = (g^y, g^{xy} 2m)$ is a ciphertext for $2m$. Hence, Eve can ask for the decryption of $c' = (c_1, 2c_2)$, which gives her $m' = 2m$, from which she determines $m = 2^{-1} m' \pmod p$.

In fact, again, the reason that ElGamal is not secure under a chosen ciphertext attack is that it is multiplicatively homomorphic.

Example 197. Show that ElGamal is multiplicatively homomorphic.

Solution. Let $(g^{y_1}, g^{xy_1} m_1)$ be a ciphertext for m_1, and $(g^{y_2}, g^{xy_2} m_2)$ a ciphertext for m_2.

The product (component-wise) of the ciphertexts is $(g^{y_1+y_2}, g^{x(y_1+y_2)} m_1 m_2)$, which is a ciphertext for $m_1 m_2$. So, again, the product of ciphertexts corresponds to the product of plaintexts.

A quick summary of some aspects of RSA and ElGamal.

- As long as appropriate key sizes are used, both RSA and ElGamal appear secure.

  About the same key size needed for both: at least 1024 bits. By now, better 2048 bits.

- The security of both RSA and ElGamal can be compromised by using a cryptographically insecure PRG to generate the secret pieces p, q (for RSA) or x (for ElGamal).

- It is important to have different ciphers, especially ones that rely on the difficulty of different mathematical problems.

  Comment. Factoring $N = pq$ and computing discrete logarithms modulo p are the two different problems for RSA and ElGamal, respectively. It is not known whether the ability to solve one of them would make it significantly easier to also solve the other one. However, historically, advances in factorization methods (like the number field sieve) have subsequently lead to similar advances in computing discrete logarithms. Both problems seem of comparable difficulty.

- Both are multiplicatively homomorphic, but RSA loses this property when padded.

Application: hash functions

A hash function H is a function, which takes an input x of arbitrary length, and produces an output $H(x)$ of fixed length, say, b bit.

Example 198. (error checking) When Alice sends a long message m to Bob over a potentially noisy channel, she also sends the hash $H(m)$. Bob, who receives m' (which, he hopes is m) and h, can check whether $H(m') = h$.

Comment. This only protects against accidental errors in m (much like the check digits in credit card numbers we discussed earlier). If Eve intercepts the message $(m, H(m))$, she can just replace it with $(m', H(m'))$ so that Bob receives the message m'.

Eve's job can be made much more difficult by sending m and $H(m)$ via two different channels. For instance, in software development, it is common to post hashes of files on websites (or announce them otherwise), separately from the actual downloads. For that use case, we should use a one-way hash function (see next example).

- The hash function $H(x)$ is called **one-way** if, given y, it is computationally infeasible to compute m such that $H(m) = y$. [Also called **preimage-resistant**.]

  Similarly, a hash function is called (weakly) **collision-resistant** if, given a message m, it is difficult to find a second message m' such that $H(m) = H(m')$. [Also called **second preimage-resistant**.]

- It is called **(strongly) collision-resistant** if it is computationally infeasible to find two messages m_1, m_2 such that $H(m_1) = H(m_2)$.

  Comment. Every hash function must have many collisions. On the other hand, the above requirement says that finding even one must be exceedingly difficult.

Example 199. (error checking, cont'd) Alice wants to send a message m to Bob. She wants to make sure that nobody can tamper with the message (maliciously or otherwise). How can she achieve that?

Solution. She can use a one-way hash function H, send m to Bob, and publish (or send via some second route) $y = H(m)$. Because H is one-way, Eve cannot find a value m' such that $H(m') = y$.

Some applications of hash functions include:

- **error-checking:** send m and $H(m)$ instead of just m
- **tamper-protection:** send m and $H(m)$ via different channels (H must be one-way!)

  If H is one-way, then Eve cannot find m' such that $H(m') = H(m)$, so she cannot tamper with m without it being detected.

- **password storage:** discussed later (there are some tricky bits)
- **digital signatures**: more later
- **blockchains**: used, for instance, for cryptocurrencies such as Bitcoin

Some popular hash functions:

	published	output bits	comment
CRC32	1975	32	not secure but common for checksums
MD5	1992	128	common; used to be secure (now broken)
SHA-1	1995	160	common; used to be secure (collision found in 2017)
SHA-2	2001	256/512	considered secure
SHA-3	2015	arbitrary	considered secure

- CRC is short for **Cyclic Redundancy Check**. It was designed for protection against common transmission errors, not as a cryptographic hash function (for instance, CRC is a linear function).
- SHA is short for **Secure Hash Algorithm** and (like DES and AES) is a federal standard selected by NIST. SHA-2 is a family of 6 functions, including SHA-256 and SHA-512 as well as truncations of these.

  SHA-3 is not meant to replace SHA-2 but to provide a different alternative (especially following successful attacks on MD5, SHA-1 and other hash functions, NIST initiated an open competition for SHA-3 in 2007). SHA-3 is based on Keccak (like AES is based on Rijndael; Joan Daemen involved in both). Although the ouput of SHA-3 can be of arbitrary length, the number of security bits is as for SHA-2.

  https://en.wikipedia.org/wiki/NIST_hash_function_competition
- MD is short for **Message Digest**. These hash functions are due to Ron Rivest (MIT), the "R" in RSA. Collision attacks on MD5 can now produce collisions within seconds. For a practical exploit, see: https://en.wikipedia.org/wiki/Flame_(malware)

  MD6 was submitted as a candidate for SHA-3, but later withdrawn.

Constructions of hash functions

Recall that a hash function H is a function, which takes an input x of arbitrary length, and produces an output $H(x)$ of fixed length, say, b bit.

Example 200. (Merkle–Damgård construction) Similarly, a **compression function** $\tilde{H}$ takes input x of length $b+c$ bits, and produces output $\tilde{H}(x)$ of length b bits. From such a function, we can easily create a hash function H. How?

Importantly, it can be proved that, if $\tilde{H}$ is collision-resistant, then so is the hash function H.

Solution. Let x be an arbitrary input of any length. Let's write $x = x_1x_2x_3...x_n$, where each x_i is c bits (if necessary, pad the last block of x so that it can be broken into c bit pieces).
Set $h_1 = 0$ (or any other initial value), and define $h_{i+1} = \tilde{H}(h_i, x_i)$ for $i \geqslant 1$. Then, $H(x) = h_{n+1}$ (b bits).
[In $\tilde{H}(h_i, x_i)$, we mean that the b bits for h_i are concatenated with the c bits for x_i, for a total of $b+c$ bits.]
Comment. This construction is known as a Merkle–Damgård construction and is used in the design of many hash functions, including MD5 and SHA-1/2.
Careful padding. Some care needs to be applied to the padding. Just padding with zeroes would result in easy collisions (why?), which we would like to avoid. For more details:
https://en.wikipedia.org/wiki/Merkle-Damgård_construction

Example 201. Consider the compression function $\tilde{H}\colon \{3\,\text{bits}\} \to \{2\,\text{bits}\}$ defined by

x	000	001	010	011	100	101	110	111
$\tilde{H}(x)$	00	10	11	01	10	00	01	11

[This was not chosen randomly: the first output bit is the sum of the digits, and the second output bit is just the second input bit.]

(a) Find a collision of $\tilde{H}$.

(b) Let H be the hash function obtained from $\tilde{H}$ using the Merkle–Damgård construction (using initial value $h_1 = 0$). Compute $H(1101)$.

(c) Find a collision with $H(1101)$.

Solution.

(a) For instance, $\tilde{H}(001) = \tilde{H}(100)$.

(b) Here, $b = 2$ and $c = 1$, so that each x_i is 1 bit: $x_1x_2x_3x_4 = 1101$.
$h_1 = 00$
$h_2 = \tilde{H}(h_1, x_1) = \tilde{H}(001) = 10$
$h_3 = \tilde{H}(h_2, x_2) = \tilde{H}(101) = 00$
$h_4 = \tilde{H}(h_3, x_3) = \tilde{H}(000) = 00$
$h_5 = \tilde{H}(h_4, x_4) = \tilde{H}(001) = 10$
Hence, $H(1101) = h_5 = 10$.

(c) Our computation above shows that, for instance, $H(1) = 10 = H(1101)$.

The construction of good hash functions is linked to the construction of good ciphers. Below, we indicate how to use a block cipher to construct a hash function.

Why linked? The ciphertext produced by a good cipher should be indistinguishable from random bits. Similarly, the output of a cryptographic hash function should look random, because the presence of patterns would likely allow us to compute preimages or collisions.

However. The design goals for a hash function are somewhat different than for a cipher. It is therefore usually advisable to not crossbreed these constructions and, instead, to use a specially designed hash function like SHA-2 when a hash function is needed for cryptographic purposes.

First, however, a cautionary example.

Example 202. (careful!) Let E_k be encryption using a block cipher (like AES). Is the compression function $\tilde{H}$ defined by

$$\tilde{H}(x, k) = E_k(x)$$

one-way?

Solution. No, it is not one-way.

Indeed, given y, we can produce many different (x, k) such that $\tilde{H}(x, k) = y$ or, equivalently, $E_k(x) = y$. Namely, pick any k, and then choose $x = D_k(y)$.

Example 203. Let E_k be encryption using a block cipher (like AES). Then the compression function $\tilde{H}$ defined by

$$\tilde{H}(x,k) = E_k(x) \oplus x$$

is usually expected to be collision-resistant.

Let us only briefly think about whether $\tilde{H}$ might have the weaker property of being one-way (as opposed to the previous example). For that, given y, we try to find (x,k) such that $\tilde{H}(x,k) = y$ or, equivalently, $E_k(x) \oplus x = y$. This seems difficult.

Just getting a feeling. We could try to find such (x,k) with $x = 0$. In that case, we need to arrange k such that $E_k(0) = y$. For a block cipher like AES, this seems difficult. In fact, we are trying a known-plaintext attack on the cipher here: assuming that $m = 0$ and $c = y$, we are trying to determine the key k. A good cipher is designed to resist such an attack, so that this approach is infeasible.

Comment. Combined with the Merkle–Damgård construction, you can therefore use AES to construct a hash function with 128 bits output size. However, as indicated before, it is advisable to use a hash function designed specifically for the purpose of hashing.

For several other (more careful) constructions of hash functions from block ciphers, you can check out Chapter 9.4.1 in the *Handbook of Applied Cryptography* (Menezes, van Oorschot and Vanstone, 2001), freely available at: http://cacr.uwaterloo.ca/hac/

We have seen how hash functions can be constructed from block ciphers (though this is usually not advisable). Similarly, hash functions can be used to build PRGs (and hence, stream ciphers).

Example 204. A hash function $H(x)$, producing b bits of output, can be used to build a PRG as follows. Let x_0 be our b bit seed. Set $x_n = H(x_{n-1})$, for $n \geqslant 1$, and $y_n = x_n \pmod 2$. Then, the output of the PRG are the bits $y_1 y_2 y_3 \ldots$

Comment. As for the B-B-S PRG, if b is large, it might be OK to extract more than one bit from each x_n.

Comment. Technically speaking, we should extract a "hardcore bit" y_n from x_n.

Comment. It might be a little bit better to replace the simple rule $x_n = H(x_{n-1})$ with $x_n = H(x_0, x_{n-1})$. Otherwise, collisions would decrease the range during each iteration. However, if b is large, this should not be a practical issue. (Also, think about how this alleviates the issue in the next example.)

Comment. Of course, one might then use this PRG as a stream cipher (though this is probably not a great idea, since the design goals for hash functions and secure PRGs are not quite the same). Our book lists a similar construction in Section 8.7: starting with a seed $x_0 = k$, bytes x_n are created as follows $x_1 = H(x_0)$ and $x_n = L_8(H(x_0, x_{n-1}))$, where L_8 extracts the leftmost 8 bits. The output of the PRG is $x_1 x_2 x_3 \ldots$ However, can you see the flaw in this construction? (Hint: it repeats very soon!)

Example 205. Suppose, with the same setup as in the previous example, we let our PRG output $x_1 x_2 x_3 \ldots$, where each x_n is b bits. What is your verdict?

Solution. This PRG is not unpredictable (at all). After b bits have been output, x_1 is known and $x_2 = H(x_1)$ can be predicted perfectly. Likewise, for all the following output.

Comment. While completely unacceptable for cryptographic purposes, this might be a fine PRG for other purposes that do not need unpredictability.

Here is a compression function, which is provably strongly collision-resistant.

However, it is rather slow and so it is not practical for hashing larger data. On the other hand, its slowness could be beneficial for applications like password hashing.

Example 206. (the discrete log hash) Let p be a large safe prime, meaning that $q=(p-1)/2$ is also prime. Let g_1, g_2 be two primitive roots modulo p. Define the compression function $\tilde{H}$ as:

$$\tilde{H}\colon \{0,1,\ldots,q^2-1\} \to \{1,2,\ldots,p-1\}, \quad \tilde{H}(m_1+m_2 q) = g_1^{m_1} g_2^{m_2} \pmod{p}.$$

[Note that, although not working with inputs and outputs of certain size in bits, this is a compression function, because the input space is much larger than the output space.]

Show that finding a collision of $\tilde{H}$ is as difficult as determining the discrete logarithm x in $g_1^x = g_2 \pmod{p}$.

Solution. Suppose we have a collision: $g_1^{m_1} g_2^{m_2} \equiv g_1^{m_1'} g_2^{m_2'} \pmod{p}$

Hence, $g_1^{(m_1-m_1')+(m_2-m_2')x} \equiv 1 \pmod{p}$ or, equivalently, $(m_1-m_1')+(m_2-m_2')x \equiv 0 \pmod{p-1}$ (because g_1 is a primitive root and so has order $p-1$).

This final congruence can now be solved for x.

More precisely, if $d=\gcd(m_2-m_2', p-1)$, there are actually d solutions for x. Since we chose p to be safe, the only factors of $p-1$ are $1, 2, (p-1)/2, p-1$.

Since $|m_2-m_2'|<q$, the only possibilities are $d=1,2$ (unless $m_2=m_2'$; however, this cannot be the case since then also $m_1=m_1'$, so that we wouldn't have a collision in the first place).

Passwords

Let's say you design a system that users access using personal passwords. Somehow, you need to store the password information.

- The worst thing you can do is to actually store the passwords m.

  This is an absolutely atrocious choice, even if you take severe measures to protect (e.g. encrypt) the collection of passwords.

  Comment. Sadly, there are still systems out there doing that. An indication that this might* be happening is when systems require you to update passwords and then complain that your new password is too close to the original one. Any reasonably designed system should never learn about your actual password in the first place!

  *: On the other hand, think about how you could check for (certain kinds of) closeness of passwords without having to store the actual password.

- Better, but still terrible, is to instead store hashes $H(m)$ of the passwords m.

  Good. An attacker getting hold of the password file, only learns about the hash of a user's password. Assuming the hash function is one-way, it is infeasible for the attacker to determine the corresponding password (if the password was randomly chosen!).

  Still bad. However, passwords are (usually) not random. Hence, an attacker can go through a list of common passwords (dictionary attack), compute the hashes and compare with the hashes of users (similarly, a brute-force attack can simply go through all possible passwords).

  Even worse, it is immediately obvious if two users are using the same password (or, if the same user is using the same password for different services using the same hash function).

  Comment. So, storing password hashes is not OK unless all passwords are completely random.

- Better, a random value s is generated for each user, and then s and $H(m, s)$ are stored. The value s is referred to as **salt**.

  In other words, instead of storing the hash of the password m, we are storing the hash of the salted password, as well as the salt.

  Why? Two users using the same password would have different salt and hence different hashes stored. As a consequence, an attacker can (of course) still mount a dictionary or brute-force attack but only against a single user, not all users at once.

  Comment. Note how the concept of salt is similar to a nonce.

  Comment. To be future-proof, the hash+salt is often stored in a single field in a format like (hash-algo, salt, salted hash).

  Comment. There's also the concept of **pepper** (usually, sort of a secret salt). This provides extra security if the pepper is stored separately. [Sometimes pepper is used as a sort of small random salt, which is discarded; this only slows a brute-force attack down and should instead be addressed using the item below.]

  https://en.wikipedia.org/wiki/Pepper_(cryptography)

- Finally, we should not use the usual (fast!) hash functions like SHA-2.

  Why? One of the things that makes SHA-2 a good hash function in practice is its speed. However, that actually makes SHA-2 a poor choice in this context of password hashing. An attacker can compute billions of hashes per second, which makes a dictionary or brute-force attack very efficient.

  To make a dictonary or brute-force attack impractical, the hashing needs to be slowed down. See Example 207 for some scary numbers.

Hashing functions like SHA-2 are not secure password hashing algorithms.

Instead, options that are considered secure include: PBKDF2, bcrypt, scrypt, Argon2.

Comment. For instance, WPA2 uses PBKDF2 based on SHA-1 with 4096 (fairly small!) iterations.

Comment. Only increasing the number of iterations increases computation time but not memory usage. scrypt and Argon2 are designed to also consume an arbitrarily specified amount of memory.

For a nice discussion about password hashing:

https://security.stackexchange.com/questions/211/how-to-securely-hash-passwords

Example 207. (the power of brute-force) In April 2024, the Bitcoin network hashrate is about $600\text{E}{=}6 \cdot 10^{20}$ hashes per second. How long would it take to brute-force a (completely random!) 8 character password, using all 94 printable ASCII characters (excluding the space)?

Solution. There are $94^8 \approx 6.1 \cdot 10^{15}$ possible passwords. Hence, it would take about 0.000010 seconds!

Comment. Even using 10 random characters (almost no human password has that kind of entropy), there are $94^{10} \approx 5.4 \cdot 10^{19}$ possible passwords. It would take less than 0.090 seconds to go through all of these!

Comment. https://bitinfocharts.com/comparison/bitcoin-hashrate.html

Example 208. Your king's webserver contains the following code to check whether the king is accessing the server. [As is far too common, his password derives from his girlfriend's name and year of birth.]

```
def check_is_king(password):
    return password=="Ludmilla1310"
```

Obviously, anyone who might be able to see the code (including its binary version) learns about your king's password. With minimal change, how can this be fixed?

Solution. The password should be hashed. For instance, in Python, using SHA-2 (why is that actually not a good choice here?) with 256 output bits:

```
from hashlib import sha256
def check_is_king(password):
    phash = sha256(password).hexdigest()
    return phash == "9e4b4fe180e22bc6cdf01fe9711cf2558507e5c3ae1c3c1f6607a25741941c66"
```

Comment. 256 bits are 64 digits in hexadecimal.

Python comment. Of course, a real implementation would use `digest()` instead of `hexdigest()`.

[For Python 3, if operating with strings (instead of bytes), `sha256(password)` needs to be replaced with something like `sha256(password.encode('utf-8'))`.]

Why is SHA not good here? Too fast to discourage brute-force attacks.

Example 209. Suppose you don't like the idea of creating random salt.

(a) How about using the same salt for all your users?

(b) Is it a good idea to use the username as salt?

Solution.

(a) This is a terrible idea and defeats the purpose of a salt. (For instance, again an attacker can immediately see if users have the same password.)

Comment. Essentially, this is a form of pepper (if the value is kept secret, i.e. stored elsewhere).

(b) That is a reasonable idea. One reason against it is that, ideally, the salt should be unique (globally). However, this could be easily achieved by using the username combined with something identifying your service (like your hostname).

Comment. A possible practical reason against choosing the username for salt is that the username might change.

Example 210. You need to hash (salted) passwords for storage. Unfortunately, you only have SHA-2 available. What can you do?

Solution. Iterate many times! (In order to slow down the computation of the hash.) The naive way would be to simply set $h_0 = H(m)$ and $h_{n+1} = H(h_n)$. Then use as hash the value h_N for large N.

In current applications, it is typical to choose N on the order of 10^6 or higher (depending on how long is reasonable to have your user wait each time she logs in and needs her password hashed for verification).

Application: digital signatures

Goal: Using a private key, known only to her, Alice can attach her **digital signature** s to any message m. Anyone knowing her public key can then verify that it could only have been Alice, who signed the message.

- Consequently, in contrast to usual signatures, digital signatures must depend on the message to be signed so that they cannot be simply reproduced by an adversary.
- This should sound a lot like public-key cryptography!

Cryptographically speaking, a digitally signed message (m, s) from Alice to Bob achieves:

- **integrity**: the message has not been accidentally or intentionally modified
- **authenticity**: Bob can be confident the message came from Alice

  In fact, we gain even more: not only is Bob assured that the message is from Alice, but the evidence can be verified by anyone. We have "proof" that Alice signed the message. This is referred to as **non-repudiation**. We refer to a technical not a legal term here: if you are curious about legal aspects of digital signatures, see, e.g.:

  https://security.stackexchange.com/questions/1786/how-to-achieve-non-repudiation/6108

For comparison, sending a message with its hash $(m, H(m))$ only achieves integrity (with protection against intentional modification only if $H(m)$ can be sent via a separate secure channel).

Example 211. (authentication using digital signatures) Last time, we saw that using human generated and memorized passwords is problematic even if done "right" (Example 207). Among other things, digital signatures provide an alternative approach to authentication.

Authentication. If Alice wants to authenticate herself with a server, the server sends her a (random) message. Using her private key, Alice signs this message and sends it back to the server. The server then verifies her (digital) signature using Alice's public key.

Obvious advantage. The server (like everyone else) doesn't know Alice's secret, so it cannot be stolen from the server (of course, Alice still needs to protect her secret from it being stolen).

(RSA signatures) Let H be a collision-resistant hash function.

- Alice creates a public RSA key (N, e). Her (secret) private key is d.
- Her signature of m is $s = H(m)^d \pmod N$.
- To verify the signed message (m, s), Bob checks that $H(m) = s^e \pmod N$.

Example 212. We use the silly hash function $H(x) = x \pmod{10}$.
Alice's public RSA key is $(N, e) = (33, 3)$, her private key is $d = 7$.

(a) How does Alice sign the message $m = 12345$?

(b) How does Bob verify her message?

(c) Was the message $(m, s) = (314, 2)$ signed by Alice?

Armin Straub
straub@southalabama.edu

Solution.

(a) $H(m) = 5$. The signature therefore is $s = 5^7 \pmod{33}$ (note how computing that signature requires Alice's private key). Computing that, we find $s = 14$.

(b) Bob receives the signed message $(m, s) = (12345, 14)$.
He computes $H(m) = 5$ and then checks whether $H(m) \equiv s^3 \pmod{33}$ (for which he only needs Alice's public key). Indeed, $14^3 \equiv 5 \pmod{33}$, so the signature checked out.

(c) We compute $H(m) = 4$ and then need to check whether $H(m) \equiv s^3 \pmod{33}$. Since $2^3 \equiv 8 \not\equiv 4 \pmod{33}$, the signature does not check out. Alice didn't sign the message.

Just to make sure. What's a collision of our hash function? Why is it totally not one-way?

Example 213. Why should Alice sign the hash $H(m)$ and not the message m?

Solution. A practical reason is that signing $H(m)$ is simpler/faster. The message m could be long, in which case we would have to do something like chop it into blocks and sign each block (but then Eve could rearrange these, so we would have to do something more clever, like for block ciphers). In any case, we shouldn't just sign $m \pmod{N}$ because then Eve can just replace m with any m' as long as $m \equiv m' \pmod{N}$.

There is another issue though. Namely, Eve can do the following **no message attack**: she starts with any signature s, then computes $m = s^e \pmod{N}$. Everyone will then believe that (m, s) is a message signed by Alice. This does not work if H is a one-way function: Eve now needs to find m such that $H(m) = s^e \pmod{N}$, but she fails to find such m if H is one-way.

Example 214. Is it enough if the hash function for signing is one-way but not collision-resistant?

Solution. No, that is not enough. If there is a collision $H(m) = H(m')$, then Eve can ask Alice to sign m to get (m, s) and later replace m with m', because (m', s) is another valid signed message. (See also the comments after the discussion of birthday attacks.)

Comment. This question is of considerable practical relevance, since hash functions like MD5 and SHA-1 have been shown to not be collision-resistant (but are still considered essentially preimage-resistant, that is, one-way). In the case of MD5, this has been exploited in practice:

https://en.wikipedia.org/wiki/MD5#Collision_vulnerabilities

Example 215. Alice uses an RSA signature scheme and the (silly) hash function $H(x) = x_1 + x_2$, where $x_1 = x^{-1} \pmod{11}$ and $x_2 = x^{-1} \pmod{7}$ [with 0^{-1} interpreted as 0] to produce the signed message $(100, 13)$. Forge a second signed message.

Solution. Since we have no other information, in order to forge a signed message, we need to find another message with the same hash value as $m = 100$. From our experience with the Chinese remainder theorem, we realize that changing x by $7 \cdot 11$ does not change $H(x)$. Hence, a second signed message is $(177, 13)$.

Comment. The hash $H(m)$ for $m = 100$ is $H(100) = (100^{-1})_{\mathrm{mod}\,11} + (100^{-1})_{\mathrm{mod}\,7} = 1 + 4 = 5$.

Similar to what we did with RSA signatures, one can use ElGamal as the basis for digital signatures. A variation of that is the **DSA** (digital signature algorithm), another federal standard.

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

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

Not surprisingly, the hashes mandated for DSA are from the SHA family.

Birthday paradox and birthday attacks

Example 216. (birthday paradox) Among $n=35$ people (a typical class size; no leaplings), how likely is it that two have the same birthday?

Solution.

$$1-\left(1-\frac{1}{365}\right)\left(1-\frac{2}{365}\right)\left(1-\frac{3}{365}\right)\cdots\left(1-\frac{34}{365}\right)\approx 0.814$$

If the formula doesn't speak to you, see Section 8.4 in our book for more details or checkout:

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

Comment. For $n=50$, we get a 97.0% chance. For $n=70$, it is 99.9%.

Comment. In reality, birthdays are not distributed quite uniformly, which further increases these probabilities. Also note that, for simplicity, we excluded "leaplings", people born on February 29.

How is this relevant to crypto, and hashes in particular?

Think about people as messages and birthdays as the hash of a person. The birthday paradox is saying that "collisions" occur more frequently then one might expect.

Example 217. Suppose M is large ($M=365$ in the birthday problem) compared to n. The probability that, among n selections from M choices, there is no collision is

$$\left(1-\frac{1}{M}\right)\left(1-\frac{2}{M}\right)\cdots\left(1-\frac{n-1}{M}\right)\approx e^{-\frac{n^2}{2M}}.$$

Why? For small x, we have $e^x\approx 1+x$ (the tangent line of e^x at $x=0$). Hence, $\left(1-\frac{k}{M}\right)\approx e^{-k/M}$ and

$$\left(1-\frac{1}{M}\right)\left(1-\frac{2}{M}\right)\cdots\left(1-\frac{n-1}{M}\right)\approx e^{-1/M}e^{-2/M}\cdots e^{-(n-1)/M}=e^{-(1+2+\ldots+(n-1))/M}=e^{-\frac{n(n-1)}{2M}}$$

In the last step, we used that $1+2+\ldots+(n-1)=\frac{n(n-1)}{2}$. Finally, $e^{-\frac{n(n-1)}{2M}}\approx e^{-\frac{n^2}{2M}}$.

Decent approximation? For instance, for $M=365$ and $n=35$, we get 0.813 for the chance of a collision (instead of the true 0.814).

Important observation. If $n\approx\sqrt{M}$, then the probability of no collision is about $e^{-1/2}\approx 0.607$. In other words, a collision is quite likely!

In the context of hash functions, this means the following: if the output size is b bits, then there are $M=2^b$ many possible hashes. If we make a list of about $\sqrt{M}=2^{b/2}$ many hashes, we are likely to observe a collision.

For collision-resistance, the output size of a hash function needs to have twice the number of bits that would be necessary to prevent a brute-force attack.

Practical relevance. This is very important for every application of hashes which relies on collision-resistance, such as digital signatures.

For instance, think about Eve trying to trick Alice into signing a fraudulent contract m. Instead of m, she prepares a different contract m' that Alice would be happy to sign. Eve now creates many slight variations of m and m' (for instance, by varying irrelevant things like commas or spaces) with the hope of finding $\tilde{m}$ and $\tilde{m}'$ such that $H(\tilde{m})=H(\tilde{m}')$. (Why?!!)

Armin Straub
straub@southalabama.edu

Example 218. (chip based credit cards) Modern chip based credit cards use digital signatures to authenticate a payment.

How? The card carries a public key, which is signed by the bank, so that a merchant can verify the public key. The card then signs a challenge from the merchant for authentication. The private key used for that is not even known to the bank.

Note that all of this can be done offline, without needing to contact the bank during the transaction.

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

There's an interesting and curious story made possible by the fact that, around 2000, banks in France used 320 bit RSA (chosen in the 80s and then not fixed despite expert advice) for signing the card's public key:

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

Comment. For contrast, the magnetic stripe just contains the card information such as card number.

Comment. This also leads to interesting questions like: can we embed a private key in a chip (or code) in such a way that an adversary, with full access to the circuit (or code), still cannot extract the key?

https://en.wikipedia.org/wiki/Obfuscation#White_box_cryptography

A digital signature is like a hash, which can only be created by a single entity (using a private key) but which can be verified by anyone (using a public key).

As one might expect, a symmetric version of this idea is also common:

Example 219. (MAC) A **message authentication code**, also known as a **keyed hash**, uses a private key k to compute a hash for a message.

Like a hash, a MAC provides integrity. Further, like a digital signature, it provides authenticity because only parties knowing the private key are able to compute the correct hash.

Comment. On the other hand, a MAC does not offer non-repudiation because several parties know the private key (whether non-repudiation is desirable or undesirable depends on the application). Hence, it cannot be proven to a third party who among those computed the MAC (and, in any case, such a discussion would make it necessary to reveal the private key, which is usually unacceptable).

From hash to MAC. If you have a cryptographic hash function H, you can simply produce a MAC $M_k(x)$ (usually referred to as a HMAC) as follows:

$$M_k(x) = H(k, x)$$

This seems to work fine for instance for SHA-3. On the other hand, this does not appear quite as secure for certain other common hash functions. Instead, it is common to use $M_k(x) = H(k, H(k, x))$. For more details, see:

https://en.wikipedia.org/wiki/Hash-based_message_authentication_code

From ciphers to MAC. Similarly, we can also use ciphers to create a MAC. See, for instance:

https://en.wikipedia.org/wiki/CBC-MAC

Elliptic curve cryptography

The idea of Diffie–Hellman (used, for instance, in DH key exchange, ElGamal or DSA) can be carried over to algebraic structures different from multiplication modulo p.

Recall that the key idea is, starting from individual secrets x, y, to share g^x, g^y modulo p in order to arrive at the joint secret $g^{xy} \pmod p$. That's using multiplication modulo p.

One important example of other such algebraic structures, for which the analog of the discrete logarithm problem is believed to be difficult, are elliptic curves.

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

Comment. The main reason (apart from, say, diversification) is that this leads to a significant saving in key size and speed. Whereas, in practice, about 2048bit primes are needed for Diffie–Hellman, comparable security using elliptic curves is believed to only require about 256bits.

For a beautiful introduction by Dan Boneh, check out the presentation:

https://www.youtube.com/watch?v=4M8_Oo7lpiA

Points on elliptic curves

An **elliptic curve** is a (nice) cubic curve that can (typically) be written in the form

$$y^2 = x^3 + ax + b.$$

A point (x, y) is on the elliptic curve if it satisfies this equation. Each elliptic curve also contains the special point O ("the point at ∞"). [O will act as the neutral element when "adding points".]

Advanced comment. Sometimes it is useful (or necessary) to consider elliptic curves defined by more general cubic equations such as $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ (however, in most cases, a linear change of variables can transform this equation into the simpler form $y^2 = x^3 + ax + b$ mentioned above).

Example 220. Determine some points (x, y) on the elliptic curve E, described by

$$y^2 = x^3 - x + 9.$$

Solution. We can try some small values for x (say, $x = 0$, $x = 1$, $x = 2$, ...) and see what y needs to be in order to get a point on the elliptic curve. For instance, for $x = 1$, we get $x^3 - x + 9 = 9$ which implies that $(1, \pm 3)$ are points on the elliptic curve.

Doing so, we find the integral points $(0, \pm 3), (\pm 1, \pm 3)$.

On the other hand, for $x = 2$, we get $x^3 - x + 9 = 15$ which implies that $(2, \pm\sqrt{15})$ are points on the elliptic curve. Depending on the application, we are often not interested in such irrational points.

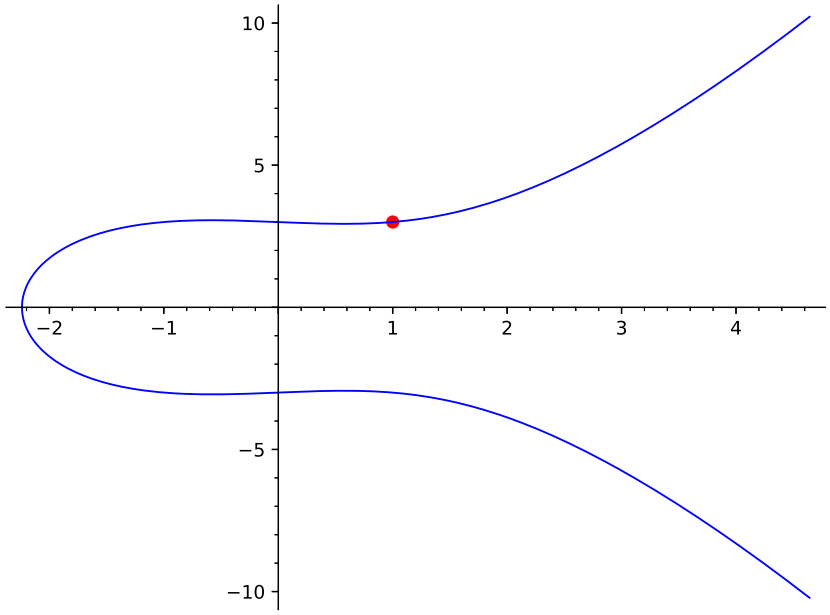
Much less obvious rational points include $(35, 207)$ or $\left(\frac{1}{36}, \frac{647}{216}\right)$ (see Example 222).

Comment. In general, it is a very difficult problem to determine all rational points on an elliptic curve, and lots of challenges remain open in that arena.

Example 221. Plot the elliptic curve E, described by $y^2 = x^3 - x + 9$ and mark the point $(1,3)$.

Solution. We let Sage do the work for us:

```
>>> E = EllipticCurve([-1,9])
>>> E.plot() + E(1,3).plot(pointsize=50, rgbcolor=(1,0,0))
```



Adding points on elliptic curves

Note. Simply adding the coordinates of two points P and Q on an elliptic curve will (almost always) not result in a third point on the elliptic curve. However, we will define a more fancy "addition" of points, which we will denote $P \boxplus Q$, such that the $P \boxplus Q$ is on the elliptic curve as well.

Given a point $P = (x, y)$ on E, we define $-P = (x, -y)$ which is another point on E.

Let us introduce an operation $\boxplus$ in the following geometric fashion: given two points P, Q, the line through these two points intersects the curve in a third point R.

We then define $P \boxplus Q = -R$.

We remark that $P \boxplus (-P)$ is the point O "at ∞". That's the neutral (zero) element for $\boxplus$.

How does one define $P \boxplus P$? (Tangent line!)

Comment. Are you able to explain why, if P and Q have rational coordinates, the same is true for R?

Remarkably, the "addition" $P \boxplus Q$ is associative. (This is not obvious from the definition.)

Using $\boxplus$, we can construct new points: for instance, $(0,3) \boxplus (1,-3) = (35, 207)$ as we will verify in the next example using Sage.

Easier to verify (but not producing anything new) is $(0,3) \boxplus (1,3) = (-1,-3)$.

Example 222. Consider again the elliptic curve E, described by $y^2 = x^3 - x + 9$.

(a) Determine $(0,3) \boxplus (1,3)$.

(b) Determine $(0,3) \boxplus (1,-3)$.

(c) Determine $4(0,3)$, which is short for $(0,3) \boxplus (0,3) \boxplus (0,3) \boxplus (0,3)$.

Solution. We let Sage do the work for us:

```
>>> E = EllipticCurve([-1,9])
>>> E(0,3) + E(1,3)
```

$(-1 : -3 : 1)$

```
>>> E(0,3) + E(1,-3)
```

$(35 : 207 : 1)$

```
>>> 4*E(0,3)
```

$$\left(-\frac{1677023}{60279696} : \frac{1406201395535}{468011559744} : 1\right)$$

We conclude that $(0,3) \boxplus (1,3) = (-1,-3)$ and $(0,3) \boxplus (1,-3) = (35,207)$ (one of the points mentioned in Example 220), while

$$4(0,3) = \left(-\frac{1677023}{60279696}, \frac{1406201395535}{468011559744}\right).$$

Comment. Note how Sage represents the point (x, y) as $(x : y : 1)$. These are **projective coordinates** which make it easier to incorporate the special point O which is represented by $(0 : 1 : 0)$.

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

The following computation demonstrates that adding O doesn't do anything:

```
>>> E(0)
```

$(0 : 1 : 0)$

```
>>> E(0,3) + E(0)
```

$(0 : 3 : 1)$

Comment. Note that, starting with a single point such as $(0, -3)$, we can generate other points such as $2(0,-3) = \left(\frac{1}{36}, \frac{647}{216}\right)$ (one of the points mentioned in Example 220). If the initial point is rational then so are the points generated from it.

Advanced comment. If you want to dig deeper, you can try to translate the geometric description of the addition $P \boxplus Q$ into algebra by deriving equations for the coordinates of $P \boxplus Q = (x_r, y_r)$ in terms of the coordinates of $P = (x_p, y_p)$ and $Q = (x_q, y_q)$. For instance, for the elliptic curve $y^2 = x^3 + ax + b$, one finds that

$$\begin{aligned} x_r &= \lambda^2 - x_p - x_q, \\ y_r &= \lambda(x_p - x_r) - y_p, \end{aligned}$$

where $\lambda = (y_q - y_p)/(x_q - x_p)$ is the slope of the line connecting P and Q. If P and Q are the same point, then this line becomes the tangent line and the slope becomes $\lambda = (3x_p^2 + a)/(2y_p)$ instead. For more details:

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

From these formulas, can you reproduce the computations we did in Sage?

Elliptic curves modulo primes

For cryptographic purposes, elliptic curves are usually considered modulo a (large) prime p.

Example 223. Let us consider $y^2 = x^3 - x + 9$ (the elliptic curve from the previous examples) modulo 7. List all points on that curve.

Solution. Note that, because we are working modulo 7, there are only 7 possible values for each of x and y. Hence, we can just go through all $7^2 = 49$ possible points (x, y) to find all points on the curve.

Or, better, we go through all possibilities for x (such as $x = 2$) and determine the corresponding possible values for y (if $x = 2$, then $y^2 = 2^3 - 2 + 9 = 15 \equiv 1 \pmod 7$ which has solutions $y \equiv \pm 1 \pmod 7$).

Doing so, we find 9 points: $O, (0, \pm 3), (\pm 1, \pm 3), (2, \pm 1)$.

[Recall that O is the special point "at ∞" which serves as the neutral element with respect to $\boxplus$.]

Comment. A theorem of Hasse–Weil says that the number of points on an elliptic curve modulo p is always close to p (this is indeed what we expect because, for each of the p choices for x, we get an equation of the form $y^2 \equiv a \pmod p$ which has 2 solutions if a is a nonzero quadratic residue [and for a random a the odds are about 50% that it is quadratic]). Moreover, we can compute the exact number of points very efficiently.

By taking everything modulo 7, we still have the previously introduced addition rule $\boxplus$.

For instance. $(0, 3) \boxplus (1, -3) = (35, 207) \equiv (0, -3) \pmod 7$

Here is how we can use Sage to list all points, as well as add any two of them:

```
>>> E7 = EllipticCurve(GF(7), [-1,9])
>>> E7.points()
  [(0: 1: 0), (0: 3: 1), (0: 4: 1), (1: 3: 1), (1: 4: 1), (2: 1: 1), (2: 6: 1), (6: 3: 1), (6: 4: 1)]
>>> E7(0,3) + E7(1,-3)
  (0: 4: 1)
>>> E7(1,-3) + E7(0,-3)
  (6: 3: 1)
```

Multiples of a point are simply denoted with nP. For instance, $3P = P \boxplus P \boxplus P$.

We then have a version of the **discrete logarithm** problem for elliptic curves:

(discrete logarithm) Given P, xP on an elliptic curve, determine x.
(computational Diffie–Hellman) Given P, xP, yP on an elliptic curve, determine $(xy)P$.

Comment. Interestingly, it appears that the computational Diffie–Hellman problem (CDH) is more difficult for elliptic curves modulo p than for regular multiplication modulo p. Indeed, suppose that p is an n-digit prime. Then the best known algorithms for regular CDH modulo p has runtime $2^{O(\sqrt[3]{n})}$, whereas the best algorithm for the elliptic curve CDH modulo p has runtime $\sqrt{p} \approx 2^{n/2} = 2^{O(n)}$.

As a consequence, it is believed that a smaller prime p can be used to achieve the same level of security when using elliptic curve Diffie–Hellman (ECDH). In practice 256bit primes are used, which is believed to provide security comparable to 2048bit (or, maybe, even 3072bit) regular Diffie–Hellman (DH); this makes ECDH about ten times faster in practice than DH.

Comment. On the other hand, due to that reduced bit size, quantum computing attacks on elliptic curve cryptography, if they become available, would be more feasible compared to attacks on ElGamal/RSA.

It is not an easy task to "randomly generate" cryptographically secure elliptic curves plus suitable base point. That is a reason why pre-selected elliptic curves are of practical importance.

The following are a few examples of specific elliptic curves that are widely used in practice.

http://blog.bjrn.se/2015/07/lets-construct-elliptic-curve.html

Example 224. For signing transactions, Bitcoin uses the elliptic curve

$$y^2 = x^3 + 7 \pmod p, \quad p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1,$$

together with the base point $P = (P_x, P_y)$ such that, in hexadecimal,

$$P_x = 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798.$$

(The y-coordinate P_y can be "lifted" from this; see the Sage code below.)

Where is this coming from? This is one of several vetted choices of elliptic curves compiled by the Standards for Efficient Cryptography Group (SECG), an industry consortium, in SEC 2: http://www.secg.org/

The particular curve above is secp256k1 in that document. While the equation of the curve and the prime p are clearly chosen to be "nice" (and so that the curve has nice properties; for instance its order is again a prime q; consequently, all regular points have order q themselves and, thus, generate all other points), it is much more mysterious how the point P was chosen:

https://crypto.stackexchange.com/questions/60420/

On the other hand, the choice of point is believed to not make much of a difference; in particular, it is not hard to see that the discrete logarithm problem is equally difficulty for all points.

Here is how to compute with that elliptic curve in Sage:

```
>>> p = 2^256 - 2^32 - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1
>>> E = EllipticCurve(GF(p), [0,7])
>>> P = E.lift_x(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798)
>>> P
(55066263022277343669578718895168534326250603453777594175500187360389116729240: 3267051\
00207588169780830851305070431844712733806592432759389043357573374824 24: 1)
>>> E.order()
115792089237316195423570985008687907852837564279074904382605163141518161494337
>>> is_prime(E.order())
1
>>> P.order()
115792089237316195423570985008687907852837564279074904382605163141518161494337
>>> 100*P
(10730358229073309792484219397246502205314821177519437367153951831350019463975 2: 103795\
96610878271744680668402374216846236544927263979079559154460683600744663 8: 1)
```

Example 225. A few years ago, more than 90% of webservers used one specific, NIST specified, elliptic curve referred to as P-256:

$$y^2 = x^3 - 3x + 41058363725152142129326129780047268409114441015993725554835256314039467401291,$$

taken modulo $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ (the fact that $p \approx 2^{256}$ makes the computations on the elliptic curve much faster in practice). The initial point $P = (x, y)$ on the curve has huge coordinates as well.

Using this single curve is sometimes considered to be problematic, especially following the concerns that the NSA may have implemented a backdoor into Dual_EC_DRBG, which was a previous NIST standard (2006–2014).

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

Example 226. A popular alternative is the curve Curve25519. In addition to some desirable theoretical advantages, its parameters are small ("nothing-up-my-sleeve numbers") and therefore not of similarly mysterious origin as the ones for P-256:

$$y^2 = x^3 + 486662x^2 + x, \quad p = 2^{255} - 19, \quad x = 9.$$

[Instead of points with (x, y) coordinates, one can actually work with just the x-coordinates for an additional speed-up.]

https://en.wikipedia.org/wiki/Curve25519

```
>>> E = EllipticCurve(GF(2^255-19), [0,486662,0,1,0])
>>> E
```

$y^2 = x^3 + 486662\,x^2 + x$

```
>>> E.order()
```

57896044618658097711785492504343953926856930875039260848015607506283634007912

```
>>> log(E.order(),2).n()
```

255.000000000000

```
>>> P = E.lift_x(9)
>>> P
```

(9: 43114425171068552920764898935933967039370386198203806730763910166200978582548: 1)

```
>>> 100*P
```

(44032819295671302737126221960004779200206561247519912509082330344845040669336: 8626006\
39244757237163427806001665957575078127166632317389150490196167274334 4: 1)

```
>>> P.order()
```

7237005577332262213973186563042994240857116359379907606001950938285454250989

```
>>> log(P.order(),2).n()
```

252.000000000000

```
>>> E.order() / P.order()
```

8

```
>>> 5*(20*P) == 20*(5*P)
```

1