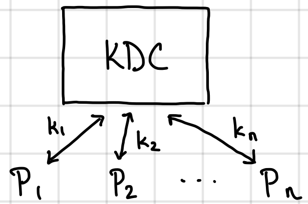
Thus far, we have <u>assumed</u> that parties have a <u>shared</u> key. Where does the shared key come from?

<u>Approach 1</u>: have a key-distribution center (KDC)



shared key between KDC and each party P_i

if P_i wants to talk to P_j:

- P_i sends nonce r_i (replay prevention) and identifier id_i to P_j
- P_j chooses nonce r_j and identifier id_j to P_i and KDC
- KDC samples k_{ij} and gives

often called a "ticket":
$$c_i \leftarrow \mathrm{Enc}(k_{i,\mathrm{Enc}}, k_{ij}), \quad t_i \leftarrow \mathrm{MAC}(k_{i,\mathrm{MAC}}, (r_i, r_j, id_i, id_j, c_i)) \quad \text{to } P_i$$

$$c_j \leftarrow \mathrm{Enc}(k_{j,\mathrm{Enc}}, k_{ij}), \quad t_j \leftarrow \mathrm{MAC}(k_{j,\mathrm{MAC}}, (r_i, r_j, id_i, id_j, c_j)) \quad \text{to } P_j$$

nonces needed to ensure "freshness" for session (no replays) and identifiers needed to bind session key k_{ij} to identities id_i, id_j

Basic design for Kerberos — only requires symmetric primitives

- <u>Drawback</u>: KDC must be fully trusted (knows everyone's keys) and is <u>single</u> point of failure (no session setup if KDC goes offline!)

<u>Public-key cryptography</u>: Session setup / key-exchange without a KDC

To develop this, we will need to introduce some abstract algebra / number theory.

<u>Definition</u>. A group consists of a set $\mathbb{G}$ together with an operation $*$ that satisfies the following properties:

- <u>Closure</u>: If $g_1, g_2 \in \mathbb{G}$, then $g_1 * g_2 \in \mathbb{G}$
- <u>Associativity</u>: For all $g_1, g_2, g_3 \in \mathbb{G}$, $g_1 * (g_2 * g_3) = (g_1 * g_2) * g_3$
- <u>Identity</u>: There exists an element $e \in \mathbb{G}$ such that $e * g = g = g * e$ for all $g \in \mathbb{G}$
- <u>Inverse</u>: For every element $g \in \mathbb{G}$, there exists an element $g^{-1} \in \mathbb{G}$ such that $g * g^{-1} = e = g^{-1} * g$

In addition, we say a group is commutative (or abelian) if the following property also holds:

- <u>Commutative</u>: For all $g_1, g_2 \in \mathbb{G}$, $g_1 * g_2 = g_2 * g_1$

<u>Notation</u>: Typically, we will use "$\cdot$" to denote the group operation (unless explictly specified otherwise). We will write g^x to denote $\underbrace{g \cdot g \cdot g \cdots g}_{x \text{ times}}$ (the usual exponential notation). We use "1" to denote the <u>multiplicative identity</u>

called "multiplicative" notation

<u>Examples of groups</u>: $(\mathbb{R}, +)$: real numbers under addition
$(\mathbb{Z}, +)$: integers under addition
$(\mathbb{Z}_p, +)$: integers modulo p under addition [sometimes written as $\mathbb{Z}/p\mathbb{Z}$]

here, p is <u>prime</u>

<u>The structure of $\mathbb{Z}_p^*$</u> (an important group for cryptography):

$$\mathbb{Z}_p^* = \{x \in \mathbb{Z}_p : \text{there exists } y \in \mathbb{Z}_p \text{ where } xy = 1 \pmod{p}\}$$

the set of elements with multiplicative inverses modulo p

What are the elements in $\mathbb{Z}_p^*$?

greatest common divisor

<u>Bezout's identity</u>: For all positive integers $x, y \in \mathbb{Z}$, there exists integers $a, b \in \mathbb{Z}$ such that $ax + by = \gcd(x,y)$.

<u>Corollary</u>: For prime p, $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$.

<u>Proof</u>. Take any $x \in \{1, 2, \ldots, p-1\}$. By Bezout's identity, $\gcd(x,p) = 1$ so there exists integers $a, b \in \mathbb{Z}$ where $1 = ax + bp$.
Modulo p, this is $ax = 1 \pmod p$ so $a = x^{-1} \pmod p$.

Coefficients a, b in Bezout's identity can be efficiently computed using the extended Euclidean algorithm:

<u>Euclidean algorithm</u>: algorithm for computing $\gcd(a,b)$ for positive integers $a > b$:
relies on fact that $\gcd(a,b) = \gcd(b, a \pmod b)$:
to see this: take any $a > b$
↳ we can write $a = b \cdot q + r$ where $q \geq 1$ is the quotient and $0 \leq r < b$ is the remainder
↳ d divides a and b $\iff$ d divides b and r
↳ $\gcd(a,b) = \gcd(b,r) = \gcd(b, a \pmod b)$
gives an explicit algorithm for computing gcd: repeatedly divide:

$\gcd(60, 27)$:

$60 = 27(2) + 6$ $[q = 2, r = 6]$ $\rightsquigarrow \gcd(60,27) = \gcd(27,6)$
$27 = 6(4) + 3$ $[q = 4, r = 3]$ $\rightsquigarrow \gcd(27,6) = \gcd(6,3)$
$6 = 3(2) + 0$ $[q = 2, r = 0]$ $\rightsquigarrow \gcd(6,3) = \gcd(3,0) = 3$

"rewind" to recover coefficients in Bezout's identity:

extended Euclidean algorithm
$$\left\{\begin{aligned} 60 &= 27(2) + 6 \\ 27 &= 6(4) + 3 \\ 6 &= 3(2) + 0 \end{aligned}\right. \longrightarrow 3 = 27 - 6 \cdot 4$$

$6 = 60 - 27(2)$

$27 - (60 - 27(2))4$
$= 27(9) + 60(-4)$

coefficients

<u>Iterations needed</u>: $O(\log a)$ — ie, bit-length of the input [worst case inputs: Fibonacci numbers]

<u>Implication</u>: Euclidean algorithm can be used to compute modular inverses (faster algorithms also exist)

cyclic groups are commutative

defined to be the identity element

Definition. A group $\mathbb{G}$ is cyclic if there exists a generator g such that $\mathbb{G} = \{g^0, g^1, \dots, g^{|\mathbb{G}|-1}\}$.

Definition. For an element $g \in \mathbb{G}$, we write $\langle g \rangle = \{g^0, g^1, \dots, g^{|\mathbb{G}|-1}\}$ to denote the set generated by g (which need not be the entire set. The cardinality of $\langle g \rangle$ is the order of g (i.e., the size of the "subgroup" generated by g)

↳ means that $g^{\mathrm{ord}(g)} = 1$

Example. Consider $\mathbb{Z}_7^* = \{1,2,3,4,5,6\}$. In this case,

$\langle 2 \rangle = \{1,2,4\}$ [2 is not a generator of $\mathbb{Z}_7^*$] $\mathrm{ord}(2) = 3$

$\langle 3 \rangle = \{1,3,2,6,4,5\}$ [3 is a generator of $\mathbb{Z}_7^*$] $\mathrm{ord}(3) = 6$

Lagrange's Theorem. For a group $\mathbb{G}$, and any element $g \in \mathbb{G}$, $\mathrm{ord}(g) \mid |\mathbb{G}|$ (the order of g is a divisor of $|\mathbb{G}|$).

↳ For $\mathbb{Z}_p^*$, this means that $\mathrm{ord}(g) \mid p-1$ for all $g \in \mathbb{G}$

Corollary (Fermat's Theorem): For all $x \in \mathbb{Z}_p^*$, $x^{p-1} = 1 \pmod p$

Proof. $|\mathbb{Z}_p^*| = |\{1,2,\dots,p-1\}| = p-1$

for integer k

By Lagrange's Theorem, $\mathrm{ord}(x) \mid p-1$ so we can write $p-1 = k \cdot \mathrm{ord}(x)$ and so $x^{p-1} = (x^{\mathrm{ord}(x)})^k = 1^k = 1 \pmod p$

Implication: Suppose $x \in \mathbb{Z}_p^*$ and we want to compute $x^y \in \mathbb{Z}_p^*$ for some large integer $y \gg p$

↳ We can compute this as

$$x^y = x^{y \bmod (p-1)} \pmod p$$

since $x^{p-1} = 1 \pmod p$

↳ Specifically, the exponents operate modulo the order of the group

↳ **Equivalently:** group $\langle g \rangle$ generated by g is isomorphic to the group $(\mathbb{Z}_q, +)$ where $q = \mathrm{ord}(g)$

$$\langle g \rangle \cong (\mathbb{Z}_q, +)$$

$$g^x \mapsto x$$

Notation: g^x denotes $\overbrace{g \cdot g \cdot \cdots \cdot g}^{x \text{ times}}$

g^{-x} denotes $(g^x)^{-1}$ [inverse of group element g^x]

$g^{x^{-1}}$ denotes $g^{(x^{-1})}$ where x^{-1} computed mod $\mathrm{ord}(g)$ — need to make sure this inverse exists!

Computing on group elements: In cryptography, the groups we typically work with will be large (e.g., 2^{256} or 2^{1024})

- Size of group element (# bits): $\sim \log |\mathbb{G}|$ bits (256 bits / 2048 bits)
- Group operations in $\mathbb{Z}_p^*$: $\log p$ bits per group element

addition of mod p elements: $O(\log p)$

multiplication of mod p values: naively $O(\log^2 p)$

Karatsuba $O(\log^{1.71} p)$

Schönhage-Strassen (GMP library): $O(\log p \log\log p \log\log\log p)$

best algorithm $O(\log p \log\log p)$ [2019]

↳ not yet practical ($> 2^{4096}$ bits to be faster...)

exponentiation: using repeated squaring: $g, g^2, g^4, g^8, \dots, g^{\lceil \log p \rceil}$, can implement using $O(\log p)$ multiplications [$O(\log^3 p)$ with naive multiplication]

↳ time/space trade-offs with more precomputed values

division (inversion): typically $O(\log^2 p)$ using Euclidean algorithm (can be improved)