Cryptology Autumn 2022 Week 0
Modular Arithmetic

Dr Chloe Martindale

These notes are for the lecture course Cryptology at the University of Bristol. If you come across any typos, please email chloe.martindale@bristol.ac.uk.

1 Modular arithmetic

Modular arithmetic is a generalization of something with which you have been familiar with for most of your life: the arithmetic of a clock. *Arithmetic* should be thought of as the basic mathematical operations such as addition, subtraction, multiplication, and division. This is something with which you are very familiar with in the sets $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$, but there are many more ways of constructing sets of numbers on which there exist consistent arithmetic laws. The arithmetic of a clock is especially interesting because we can construct a consistent set of arithmetic laws on the *finite* set of hours.

Let us start by studying 'clock addition'. If you add 4 hours to 10 o'clock, then you get 2 o'clock (rather than 14 o'clock, since we will work here with the 12-hour clock). The way that we will write this is:

$$10 + 4 \equiv 2 \pmod{12}.$$

The $\equiv$ sign should be read as 'is equivalent to', and the notation $\pmod{12}$ tells us that we should reset when we get to the number 12, or if you like that our day is split into 12 hours.

'Clock subtraction' works in much the same way. If you subtract 6 hours from 1 o'clock, then you get 7 o'clock. The way that we will write this is:

$$1 - 6 \equiv 7 \pmod{12}.$$

'Clock multiplication' can be thought of just as repeated addition, so can as be defined in a natural way. For example,

$$5 \times 3 = 5 + 5 + 5 \equiv 3 \pmod{12}.$$

Division is a little more complicated, so we will return to that later.

A natural question that arises when studying clock arithmetic is: what if the day was not split into sets of 12 hours, but some other number, like 7? We can of course set up addition, subtraction, and multiplication $\pmod 7$ in just the same way as $\pmod{12}$. Formally, we define the notation $\equiv$ and $\pmod n$ as follows.

Definition 1. Let $n \in \mathbb{Z}_{>1}$ and let $a, b \in \mathbb{Z}$. We say that

$$a \equiv b \pmod n$$

if there exists $k \in \mathbb{Z}$ such that $a = b + kn$.

We refer to basic arithmetic $\pmod n$ as *modular arithmetic*. Suppose now that we want to compute $10 \times 11 \pmod{12}$. We would like to find $a \in \mathbb{Z}$ such that $1 \leq a \leq 12$ and $10 \times 11 \equiv a \pmod{12}$. One way to do this is to first compute $10 \times 11 = 110$, and then divide 110 by 12 and take a to be the remainder. Try to prove for yourself that this will give the right answer.

Finally, let us turn to division. Suppose that you want to divide 3 by 4 on our 7-hour clock. It turns out that the best way to think of this is as 3×4^{-1}– we already have a notion of multiplication (and of 3), so it remains to understand the notion of inverses:

Definition 2. Let $a \in \mathbb{Z}$ and $n \in \mathbb{Z}_{>1}$. If there exists $b \in \mathbb{Z}$ such that

$$ab \equiv 1 \pmod n$$

then we say that $b \pmod n$ is the *inverse* of $a \pmod n$.

Notice the 'if there exists' part of this definition. Consider $a = n = 12$. No matter how many multiples of 12 you take, you are always going to land back at the 12 o'clock position on the clock, or more formally, for every $b \in \mathbb{Z}$ we have that $12b \equiv 12 \pmod{12}$, so in particular 12 has no inverse mod 12. In fact, since $12 \equiv 0 \pmod{12}$, this isn't so surprising, since we are used to the idea of 0 having no inverse. There are however other numbers by which we cannot divide $\pmod{12}$. Consider $a = 6$ and $n = 12$. For every $b \in \mathbb{Z}$ we have that either $6b \equiv 6 \pmod{12}$ or $6b \equiv 0 \pmod{12}$. So $6 \pmod{12}$ also has no inverse. When does an integer mod n have an inverse?

To understand if the inverse exists, we first need to understand in which situations the inverse of $a \pmod b$ exist for any a and b. Let's look at a couple of examples.

Examples.

- The inverse of $4 \pmod 7$ is $2 \pmod 7$ because $4 \cdot 2 \pmod 7 \equiv 1 \pmod 7$.
- $4 \pmod 8$ has no inverse because for every $n \in \mathbb{Z}$ we know that

  $$4 \cdot n \pmod 8 \in \{0 \pmod 8, 4 \pmod 8\},$$

  so in particular there does not exist any $n \pmod 8$ such that $4 \cdot n \equiv 1 \pmod 8$.

- Exercise: generalize the above example. That is, show that if m and n are not coprime then m does not have an inverse mod n.

In fact, the above exercise is also true in the reverse. That is, $a \pmod b$ is invertible if and only if a and b are coprime. The exercise above gives the 'only if', but what about the 'if'? For this we need *Euclid's algorithm.*

Algorithm 1 Euclid's Algorithm

Require: a and $b \in \mathbb{Z}_{>0}$; without loss of generality suppose that $a \geq b$.
Ensure: $d = \gcd(a, b)$.

1: Set $r_0 = a$, $r_1 = b$, and $i = 1$.
2: **while** $r_i \neq 0$ **do**
3: $\quad i \leftarrow i + 1$.
4: $\quad$ Compute the unique m_i and $r_i \in \mathbb{Z}$ such that $0 \leq r_i < r_{i-1}$ and

$$r_{i-2} = m_i \cdot r_{i-1} + r_i.$$

return r_i

Euclid's algorithm has an important corollary, which we often just lazily refer to Euclid's algorithm itself, and that is that if $d = \gcd(a, b)$ then there exist $m, n \in \mathbb{Z}$ such that

$$am + bn = d.$$

This follows from Euclid's algorithm just by solving the series

$$\{r_{i-2} = m_i \cdot r_{i-1} + r_i\}_{2 \leq i \leq k}$$

of simultaneous equations occuring in Euclid's algorithm for $r_0 = a$, $r_1 = b$, and $r_k = d$.

Recall that we wanted to show that $a \pmod b$ is invertible if a and b are coprime. So, suppose that a and b are coprime. Then by Euclid's algorithm we know that there exist $x, y \in \mathbb{Z}$ such that

$$ax + by = 1.$$

In particular, reducing mod b gives us that $ax \equiv 1 \pmod b$, so $x \pmod b$ is an inverse of $a \pmod b$.

This algorithm to compute inverses mod b is polynomial-time, but if b has many small factors it is possible to do something much faster using the *Chinese Remainder Theorem.* We first state the simplest version of this theorem

Theorem 1 (Chinese Remainder Theorem (CRT) - vanilla edition)**.** *Given coprime $n, m \in \mathbb{Z}_{>1}$ and $a, b \in \mathbb{Z}$ there exists a unique $x \pmod{nm}$ such that both*

$$x \equiv a \pmod m \quad \textit{and} \quad x \equiv b \pmod n.$$

You may have seen this before in a basic number theory course or a group theory course for example: with some mathematical machinery it is quick to

prove. We won't prove uniqueness now but we will give a useful constructive proof of existence.

Proof of existence (constructive). As $\gcd(n, m) = 1$, by Euclid's algorithm there exist $c, d \in \mathbb{Z}$ such that

$$cm + dn = 1. \tag{1}$$

We claim that

$$x = bcm + adn \pmod{mn}$$

will work. We first check mod n. Note that $cm = 1 - dn$ by (1). So

$$x = b(1 - dn) + adn \equiv b \pmod{n}.$$

Similarly

$$x = bcm + a(1 - cm) \equiv a \pmod{m}.$$

□

Let's see an example of the Chinese Remainder Theorem in action. Suppose that you are given the equations

$$x \equiv 4 \pmod{17}$$

and

$$x \equiv 3 \pmod{11}$$

and you want to find $x \pmod{17 \cdot 11}$. We first run Euclid's algorithm to find the c and d of the constructive proof above.

$$\begin{aligned} r_0 &= 17 \\ r_1 &= 11 \\ r_2 &= 17 - 1 \cdot 11 = 6 \\ r_3 &= 11 - 1 \cdot 6 = 5 \\ r_4 &= 6 - 1 \cdot 5 = 1. \end{aligned}$$

Then

$$1 = r_4 = r_2 - r_3 = (r_0 - r_1) - (r_1 - r_2) = r_0 - 2r_1 + r_2 = 2r_0 - 3r_1.$$

Now using the formula from the constructive proof of CRT, we get

$$x = 2 \cdot 17 \cdot 3 - 3 \cdot 11 \cdot 4 = 2 \cdot 3(17 - 2 \cdot 11) = -30.$$

To get a positive representative, we can just add $17 \cdot 11 = 187$, so

$$x \equiv 157 \pmod{17 \cdot 11}.$$

The Chinese Remainder Theorem becomes really powerful when we want to compute inverses mod b and b has *many* small factors (not just two). For this however we need the theorem in more generality.

Theorem 2 (Chinese Remainder Theorem (CRT) - chocolate edition)**.** *Given* $n_1, \ldots, n_k \in \mathbb{Z}_{>1}$ *that are pairwise coprime and* $a_1, \ldots, a_k \in \mathbb{Z}$ *there exists a unique* $x \pmod{n_1 \cdots n_k}$ *such that*

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ &\cdots \\ x &\equiv a_k \pmod{n_k}. \end{aligned}$$

To find such an x, we just apply Euclid's algorithm as before, but now interatively. The complexity of an algorithm computing anything (including inverses) mod b, when combined with CRT, is a function of the size of the factors of b rather then b itself, which can be engineered to be very small. As we will see later on, CRT also comes up in the context of trying to break cryptosystems.

Cryptology Autumn 2022 Week 2
Key exchange and message encryption

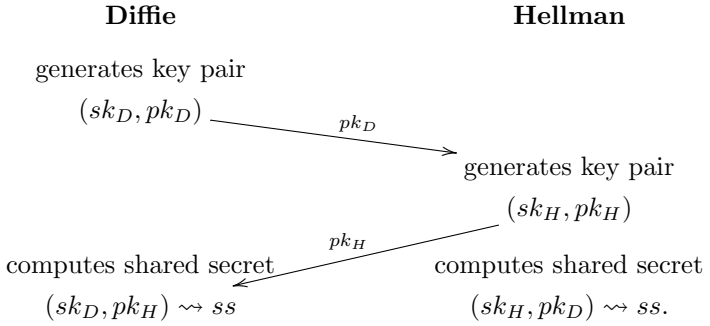
Dr Chloe Martindale

These notes are for the lecture course Cryptology at the University of Bristol. If you come across any typos, please email chloe.martindale@bristol.ac.uk.

1 Diffie-Hellman key exchange

Last week we saw the *one-time pad*, which is a secret known by multiple people which then can be used for cryptography. However, having a one-time pad with which we can work requires secure offline communication, which for most real-world scenarios is not practical and definitely not cost-effective. The cryptographic solution to this is to use a *key-exchange* algorithm, which does exactly what it says on the tin: It allows two (or more but we focus on two for now) parties who communicate over an open channel to compute a shared secret value, known only to them, which they can then use to encrypt communication between them.

The abstract idea of a key exchange is as follows: suppose Diffie and Hellman want compute a shared secret key (read: a shared value that noone else can compute).



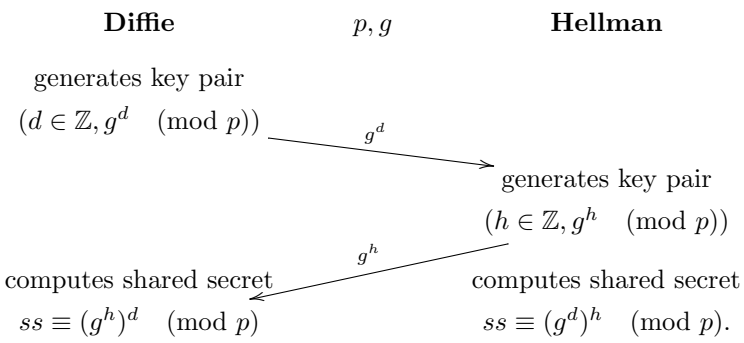
Here is a (overly simple) message encryption scheme built using such a key-exchange:

1. Alice and Bob compute ss via a key-exchange and encode it as a bit string.

2. Alice encodes her plaintext message m as a a bit string, computes the ciphertext $c = m \oplus s$, and sends c to Bob.

3. Bob decrypts the ciphertext via $m = c \oplus s$.

So, how do we instantiate such a key-exchange? The most basic version of the Diffie-Hellman key exchange uses exponentiation modulo a large prime p. A prime p and a nonzero element $g \pmod p$ is fixed in a public setup phase, and the key exchange is as follows:



In order for this to define a *crypto*system, we need more than just mathematical validity. We need any attack method to be much much slower than the algorithms used by the honest participants. 'Much slower' is something that we need to define in order to understand how to choose security parameters; for this we introduce a *security parameter* λ, which will be some smallish positive integer (often 128 in real-world scenarios), and which we use to discuss the amount of time an algorithm takes in terms of λ.

When we say that we want a computation to be 'fast' we mean 'the number of basic operations for said computation is polynomial in λ', i.e., you can abstractly compute an upper bound on the number of basic operations (e.g. addition) needed as a polynomial in λ. When we say that we want a computation to be 'slow' we 'the number of basic operations is exponential or subexponential in λ', meaning that the number of basic operations for said computation is lower bounded by $O(2^n)$ or $O(n^\alpha \log_2(n)^{1-\alpha})$ for some $\alpha \in (0, 1]$ respectively. In practise, if this is true for a reasonable α (for example, for RSA which we will see below, $\alpha = 1/3$), we can increase λ to a size for which polynomial time calculations are at most milliseconds and subexponential calculations would take years.

For Diffie-Hellman, if we choose p so that $\lambda \approx \log_2(p)$, then exponentiation mod p should be easy/fast/polynomial in λ (more on this later) and the *discrete logarithm problem*, or computing d^{th} or h^{th} roots mod p, should be hard/slow/subexponential in λ (more on this later too).

2 ElGamal encryption

A more sophisticated method to use the Diffie-Hellman key exchange to build an encrypted messaging protocol is called *ElGamal* encryption. ElGamal works as follows:

- **Setup**:
  1. Diffie chooses a prime p and a generator g of $\mathbb{Z}/p\mathbb{Z} - \{0\}$.
  2. Diffie chooses a random secret $d \in \{1, \ldots, p-1\}$ and computes $pk_D = g^d \pmod{p}$.
  3. Diffie sends his public key (p, g, pk_D) to Hellman.
- **Encryption**:
  1. Hellman chooses a random secret $h \in \{1, \ldots p-1\}$ and computes $pk_H = g^h \pmod{p}$.
  2. Hellman computes the shared secret $ss = pk_D^h \pmod{p}$.
  3. Hellman computes the encrypted message $enc_m = m \cdot ss$.
  4. Hellman sends the ciphertext (pk_H, enc_m) to Diffie.
- **Decryption**:
  1. Diffie computes the shared secret $ss = pk_H^d \pmod{p}$.
  2. Diffie computes the ciphertext $m = enc_m \cdot ss^{-1} = enc_m \cdot pk_H^{p-1-d} \pmod{p}$.

Observations:

- Note that $ss^{-1} = pk_H^{p-1-d}$ as

$$ss \cdot pk_H^{p-1-d} = g^{dh} \cdot g^{h\cdot(p-1-d)} = (g^{p-1})^h = 1 \pmod{p}.$$

- If m is known, the shared secret ss can be recovered from the ciphertext, so use a new secret h for each message.

3 RSA

This section is about RSA, named after Rivest, Shamir, and Ademan. RSA was the first public key encryption (PKE) and signature system, and is still in wide use today. First we need a little bit of maths background following on from the introduction to modular arithmetic.

Definition 1. Let $n \in \mathbb{Z}_{>0}$. The *Euler φ-function* of n is

$$\varphi(n) := \#\{m \in \mathbb{Z} : 0 < m < n, \gcd(m, n) = 1\}.$$

Examples. 1. $\varphi(7) = \#\{1,2,3,4,5,6\} = 6$.

2. $\varphi(8) = \#\{1,3,5,7\} = 4$.

Exercise: prove that for $p \neq q$ prime,

$$\varphi(p) = p - 1$$

and

$$\varphi(pq) = (p-1)(q-1).$$

The basic RSA public key encryption system consists of 3 steps: a setup phase for key generation by the user (**KeyGen**), encryption of a message m by a second party (**Encrypt**), and decryption of the message m by the user (**Decrypt**). The algorithms for these steps are below. We have coloured the users secrets in red and the public values in green.

KeyGen

1. Pick primes $p \neq q$ of bit length λ.
2. Compute $n = p \cdot q$ and $\varphi(n) = (p-1)(q-1)$.
3. Pick e coprime to $\varphi(n)$.
4. Compute $d = e^{-1} \pmod{\varphi(n)}$.
5. Generate key pair
$$\text{pk}, \text{sk} = (e, n), (d, n).$$

Encrypt

1. Pick $m \in \mathbb{Z}_{[0,n-1]}$.
2. Compute $c \equiv m^e \pmod{n}$.
3. Send c.

Decrypt

1. Compute $c^d \equiv m \pmod{n}$.

In order for this to be a valid system, there are two steps that don't obviously mathematically check out: step 4 of **KeyGen** (does the inverse exist?) and step 1 of **Decrypt**.

Recall from the Modular Arithmetic notes for week 0 that $a \pmod{b}$ is invertible if and only if a and b are coprime, so step 4 of **KeyGen** is valid.

For the **Decrypt** step, note that if m *is* invertible mod n then Fermat's Little Theorem implies that $m^{\varphi(n)} \equiv 1 \pmod{n}$. Let $k \in \mathbb{Z}$ be such that $ed = 1 + k\varphi(n)$. Then

$$c^d \equiv (m^e)^d \equiv m^{1+k\varphi(n)} \equiv m \cdot (m^{\varphi(n)})^k \equiv m \cdot 1^k \equiv m \pmod{n},$$

so the decrypt step is valid (if m is invertible mod n, actually also if it's not but that requires some more steps).

For RSA to work as a cryptosystem:

- Step 2 of **KeyGen** needs to be fast, i.e., we need to be able to multiply fast.
- Step 4 of **KeyGen** needs to be fast, i.e., we need to be able to compute inverses mod $\varphi(n)$ fast.
- In Step 5 of **KeyGen**, an attacker shouldn't be able to compute d from (e, n). Because Step 4 is fast, if the attacker knows $\varphi(n)$ then they can compute d fast. So computing $\varphi(n)$ from n should be slow. As $\varphi(n)$ is easy to compute from p and q, factoring n should be slow.
- Step 2 of **Encrypt** needs to be fast, i.e., we need to be able to exponentiate mod n fast.
- An attacker should also not be able to recover m from c, so computing e^{th} roots mod n should be slow.

More on how we ensure these things in the next lecture.

Cryptology Autumn 2022 Week 3
Easy and Hard Problems in Cryptography

Dr Chloe Martindale

These notes are for the lecture course Cryptology at the University of Bristol. If you come across any typos, please email chloe.martindale@bristol.ac.uk.

1 Fast multiplication, exponentiation, and inversion

Last week we saw how to share a secret (Diffie-Hellman), how to build message encryption from Diffie-Hellman (ElGamal), and message encryption using RSA. We also saw that the security of Diffie-Hellman and ElGamal relies on 'the Discrete Logarithm Problem' and that the security of RSA relies on factoring being hard. However, to really have a valid cryptosystem, we also need to make sure that the computations the players have to do are easy, meaning computationally fast or polynomial-time in the bit length of the input. For example, for RSA to work as a cryptosystem:

- Step 2 of **KeyGen** needs to be fast, i.e., we need to be able to multiply fast.
- Step 4 of **KeyGen** needs to be fast, i.e., we need to be able to compute inverses mod $\varphi(n)$ fast.
- In Step 5 of **KeyGen**, an attacker shouldn't be able to compute d from (e, n). Because Step 4 is fast, if the attacker knows $\varphi(n)$ then they can compute d fast. So computing $\varphi(n)$ from n should be slow. As $\varphi(n)$ is easy to compute from p and q, factoring n should be slow.
- Step 2 of **Encrypt** needs to be fast, i.e., we need to be able to exponentiate mod n fast.
- An attacker should also not be able to recover m from c, so computing e^{th} roots mod n should be slow.

We will focus for now on the computations we want to be fast. To multiply fast, we use a method called 'double-and-add'. To see how this works let's

consider how we would multiply p by q in Step 2 of **KeyGen**. We first write q in binary as $(q_\lambda, \ldots, q_0)$, or in other words

$$q = \sum_{i=0}^{\lambda} q_i 2^i,$$

where $q_i \in \{0, 1\}$. In particular

$$pq = \sum_{i=0}^{\lambda} q_i \cdot (2^i p).$$

We can compute the $2^i p$ terms just by repeated doubling – each doubling is just one addition (which is a basic operation) so this is very efficient.

- **Double**: Compute

$$\begin{aligned} 2^0 \cdot p = p &\to \text{ 0 additions} \\ 2^1 \cdot p = p + p &\to \text{ 1 addition} \\ 2^2 \cdot p = 2p + 2p &\to \text{ 1 addition} \\ &\cdots \\ 2^\lambda \cdot p = 2^{\lambda-1} p + 2^{\lambda-1} p &\to \text{ 1 addition.} \end{aligned}$$

  The doubling step costs λ additions, so is polynomial in λ i.e. fast.

Now to get $p \cdot q$ we just have to add together the terms $2^i \cdot p$ together for which $q_i = 1$. So, let $q_{i_0}, \ldots, q_{i_k}$ be the non zero coefficients of the binary expansion of q.

- **Add**: Compute

$$p \cdot q = (2^{i_0} \cdot p) + \cdots + (2^{i_k} \cdot p).$$

  The adding step costs $k \leq \lambda$ additions.

In total, the double-and-add method costs at most 2λ basic operations, so is 'fast'.

To exponentiate mod n fast, we play the same game. To see how this works let's consider computing $m^e \pmod n$ as we do in **Encrypt**. This time we use the binary expansions of $e = (e_\lambda, \ldots, e_0)$, so in other words

$$e = \sum_{i=0}^{\lambda} e_i 2^i,$$

where $e_i \in \{0, 1\}$. In particular

$$m^e = m^{\sum_{i=0}^{\lambda} e_i 2^i} = \prod_{i=0}^{\lambda} (m^{2^i})^{e_i}.$$

We can compute the $m^{2^i} \pmod n$ terms just by repeated squaring – each squaring is at most one multiplication (maybe you get some cancellation so it could be less), each of which we just saw is at most 2λ basic operations.

- **Square**: Compute

$$\begin{aligned} m^{2^0} &\equiv m \pmod{n} \to \text{ 0 squarings} \\ m^{2^1} &\equiv m^2 \pmod{n} \to \text{ 1 squaring} \\ m^{2^2} &\equiv (m^2)^2 \pmod{n} \to \text{ 1 squaring} \\ &\cdots \\ m^{2^\lambda} &\equiv (m^{2^{\lambda-1}})^2 \pmod{n} \to \text{ 1 squaring.} \end{aligned}$$

  The squaring step costs λ squarings, so is at most $2\lambda^2$ basic operations.

Note that the fact that we are doing the computations $\pmod{n}$ here is very important - for large λ you would quickly run into memory problems otherwise. All that remains now to get $m^e \pmod{n}$ is to multiply together the terms $m^{2^i} \pmod{n}$ together for which $e_i = 1$. So, let $e_{i_0}, \ldots, e_{i_k}$ be the non zero coefficients of the binary expansion of e.

- **Multiply**: Compute

$$m^e \pmod{n} \equiv m^{2^{i_0}} \times \cdots \times m^{2^{i_k}} \pmod{n}.$$

  The multiplication step then costs $k \leq \lambda$ multiplications, so $\leq 2\lambda^2$ basic operations.

From these calculations, we see that square-and-multiply can always be performed in $\leq 4\lambda^2$ basic operations, so is polynomial time, i.e. 'fast'. In practise you can do quite a bit better! But we won't go into that now.

If we look now at our list of computations that we want to be fast for RSA to work as a cryptosystem, we've tackled multiplication and exponentiation, and only inversion $\pmod{\varphi(n)}$ remains. We have actually already seen a polynomial time algorithm to compute this inverse, namely Euclid's algorithm, but since $\varphi(n)$ (unlike n) may have many small factors it is possible to do something much faster using the *Chinese Remainder Theorem*, covered also in the Modular Arithmetic notes from Week 1; we repeat the relevant parts here. We first state the simplest version of this theorem

Theorem 1 (Chinese Remainder Theorem (CRT) - vanilla edition)**.** *Given coprime $n, m \in \mathbb{Z}_{>1}$ and $a, b \in \mathbb{Z}$ there exists a unique $x \pmod{nm}$ such that both*

$$x \equiv a \pmod{m} \quad \textit{and} \quad x \equiv b \pmod{n}.$$

We won't prove uniqueness now but we will give a useful constructive proof of existence.

Proof of existence (constructive). As $\gcd(n, m) = 1$, by Euclid's algorithm there exist $c, d \in \mathbb{Z}$ such that

$$cm + dn = 1. \tag{1}$$

We claim that

$$x = bcm + adn \pmod{mn}$$

will work. We first check mod n. Note that $cm = 1 - dn$ by (1). So

$$x = b(1 - dn) + adn \equiv b \pmod{n}.$$

Similarly

$$x = bcm + a(1 - cm) \equiv a \pmod{m}.$$

□

Let's see an example of the Chinese Remainder Theorem in action. Suppose that you are given the equations

$$x \equiv 4 \pmod{17}$$

and

$$x \equiv 3 \pmod{11}$$

and you want to find $x \pmod{17 \cdot 11}$. We first run Euclid's algorithm to find the c and d of the constructive proof above.

$$\begin{aligned} r_0 &= 17 \\ r_1 &= 11 \\ r_2 &= 17 - 1 \cdot 11 = 6 \\ r_3 &= 11 - 1 \cdot 6 = 5 \\ r_4 &= 6 - 1 \cdot 5 = 1. \end{aligned}$$

Then

$$1 = r_4 = r_2 - r_3 = (r_0 - r_1) - (r_1 - r_2) = r_0 - 2r_1 + r_2 = 2r_0 - 3r_1.$$

Now using the formula from the constructive proof of CRT, we get

$$x = 2 \cdot 17 \cdot 3 - 3 \cdot 11 \cdot 4 = 2 \cdot 3(17 - 2 \cdot 11) = -30.$$

To get a positive representative, we can just add $17 \cdot 11 = 187$, so

$$x \equiv 157 \pmod{17 \cdot 11}.$$

We now introduced the Chinese Remainder Theorem as a way of speeding up the computation of the inverse of the public key $e \bmod \varphi(n)$, where $n = pq$ was the RSA modulus. We didn't however yet state the theorem in more generality, or even enough generality to see where the large speed up comes from.

Theorem 2 (Chinese Remainder Theorem (CRT) - chocolate edition)**.** *Given $n_1, \ldots, n_k \in \mathbb{Z}_{>1}$ that are pairwise coprime and $a_1, \ldots, a_k \in \mathbb{Z}$ there exists a unique $x \pmod{n_1 \cdots n_k}$ such that*

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ &\cdots \\ x &\equiv a_k \pmod{n_k}. \end{aligned}$$

To find such an x, we just apply Euclid's algorithm as before, but now interatively. The complexity of an algorithm computing anything (including inverses) mod $\varphi(n)$, when combined with CRT, is a function of the size of the factors of $\varphi(n)$ rather then $\varphi(n)$ itself, which can be engineered to be very small. As we will see later on, CRT also comes up in the context of trying to break cryptosystems.

2 The Discrete Logarithm Problem

Just as with RSA, the validity of ElGamal encryption and more generally the Diffie-Hellman key exchange relies on certain computations being easy (fast) or hard (slow). For Diffie-Hellman, exponentiation mod p should be easy (fast, polynomial time), which we have already seen it can be with square-and-multiply.

The fundamental problem that should be hard (slow, (sub) exponential time) for Diffie-Hellman is the *discrete logarithm problem*, that is, computing $d \in [0, p-1]$ given $g \pmod p$ and $g^d \pmod p$.[1] With the key exchange stated as in the Week 2 notes, there are instances where this might be very easy, for example if $g = p - 1 \equiv -1 \pmod p$ then the only values of g^d or g^h that can occur are -1 and 1 so finding a root is very easy. To avoid this, we want g^d to be able to take as many values as possible – it's time for a bit of maths.

2.1 Groups

Diffie–Hellman key exchange is an idea that works in any *group*. A group is a name for a set that satisfies some desirable properties - properties that turn out to be essential for Diffie–Hellman as well as other cryptosystems.

Definition 1. Let G be a set and $* : G \times G \to G$ a map that takes pairs of elements in G to a single element of G. We say that $(G, *)$ is a *group* or that G *defines a group under* $*$ if the following *group axioms* are satisfied:

(G1) There exists $e \in G$ such that for every $g \in G$, $e * g = g * e = g$. *(G has an identity).*

(G2) For every $g \in G$, there exists $h \in G$ such that $g * h = h * g = e$. *(every element has an inverse).*

(G3) For every $a, b, c \in G$, $(a * b) * c = a * (b * c)$. *($*$ is associative).*

We often just say 'G is a group' instead of '$(G, *)$ is a group' if the author considers it 'obvious' which operation $*$ should be.

Examples. 1. For any integer $n \geq 2$, the set $\{0 \pmod n, 1 \pmod n, \ldots, n-1 \pmod n\}$ is a group under $+ \pmod n$.

[1] Computing d given g^d if $g \in \mathbb{R}$ is something you've seen before: namely computing logarithms base g. However, the problem turns out be fundamentally different when instead of working in a continuous solution set like $\mathbb{R}$ we are working in a discrete solution set like $\mathbb{Z}/p\mathbb{Z}$–hence the name the *Discrete Logarithm Problem*.

2. For any integer $n \geq 2$, the set $\{0 \pmod n, 2 \pmod n, \ldots, n-1 \pmod n\}$ is *not* a group under multiplication $\pmod n$. Reason: $0 \pmod n$ has no inverse (c.f. (G2)).

3. For any composite integer $n \geq 2$, the set $\{1 \pmod n, 2 \pmod n, \ldots, n-1 \pmod n\}$ is *not* a group under multiplication $\pmod n$. Reason: n is composite, so there exists $0 \neq a \pmod n$ such that $\gcd(a, n) \neq 1$, which we proved above was not invertible.

4. For any prime p, the set $\{1 \pmod p, \ldots, p-1 \pmod p\}$ is a group under multiplication $\pmod p$.

Sets of integers $\pmod p$ and $\pmod n$ will return again and again, so let us introduce some notation for this. From now on, we will write

$$\mathbb{Z}/n\mathbb{Z} = \{0 \pmod n, \ldots, n-1 \pmod n\}$$

and $(\mathbb{Z}/n\mathbb{Z})^*$ for the set of invertible elements of $\mathbb{Z}/n\mathbb{Z}$.

Note that for a prime p, that means that

$$(\mathbb{Z}/p\mathbb{Z})^* = \{1 \pmod p, \ldots, p-1 \pmod p\};$$

we saw in our examples above that $(Z/p\mathbb{Z})^*$ is a group under multiplication $\pmod p$. This group turns out to be very useful for us, partly because it is *cyclic* for any prime p. That is, there exists a $g \pmod p$ such that

$$(\mathbb{Z}/p\mathbb{Z})^* = \{g \pmod p, g^2 \pmod p, \ldots, g^{p-1} \pmod p\}.$$

For example, I claim that if $p = 7$ then $\mathbb{Z}/p\mathbb{Z} - \{0\}$ can be generated by $3 \pmod 7$. To see this we just repeatedely exponentiate $3 \pmod 7$:

$$\{3^1 \equiv 3 \pmod 7, 3^2 \equiv 2 \pmod 7, 3^3 \equiv 6 \pmod 7, 3^4 \equiv 4 \pmod 7,$$
$$3^5 \equiv 5 \pmod 7, 3^6 \equiv 1 \pmod 7\},$$

which is indeed all the elements of $(\mathbb{Z}/7\mathbb{Z})^*$.

In this example, you see that the last element in the list, $g^{p-1} \pmod p$, is $1 \pmod p$. In fact, this is not a coincidence: as p is prime and $g \not\equiv 0 \pmod p$ we have that $\gcd(g, p) = 1$, so $g \pmod p$ is invertible, so by Fermat's Little Theorem $g^{p-1} \equiv 1 \pmod p$ no matter what we chose for $g \pmod p$.

Now, the reason $(\mathbb{Z}/p\mathbb{Z})^*$ is a useful group for Diffie–Hellman key exchange is because it is possible to choose $g \pmod p \in (\mathbb{Z}/p\mathbb{Z})^*$ for which $g^d \pmod p$ takes $p-1$ different values: so that there is exactly one valid private key (d) for any given public key (g^d). In fact, the reason that we chose the letter 'g' when setting up the key exchange is because we typically choose a 'generator' for the cyclic group $(\mathbb{Z}/p\mathbb{Z})^*$,

In conclusion, for our Diffie-Hellman setup, we choose p prime and g a *generator* of the group $(\mathbb{Z}/p\mathbb{Z})^*$. Observe that this choice only avoids the most obvious reason for the discrete logarithm problem (computing d from $g^d \pmod p$) being easy; we'll get to other algorithms to compute discrete logarithms in due course.

2.2 CRT vs. the Discrete Logarithm Problem

Later in the course we will look at the some of best known classical (that is 'not quantum') algorithms to attack this problem, so computing $d \in [0, p-1]$ given $g^d \pmod p$. In some contexts though we already have the tools we need: the Chinese Remainder Theorem!

We define the *order* d of an element g of a group G with group operation $*$ and identity id as the minumum positive integer d such that $\underbrace{g * \cdots * g}_{d \text{ times}} = id$. Of course in our context this looks like the minumum positive integer d such that $g^d \equiv 1 \pmod p$. In particular, the generator g that we've been using in Diffie-Hellman and ElGamal has order $p-1$ (if you don't immediately see why, take some time to think about this).

Going back to the example with $p = 7$, you can hopefully now spot that $3^2 \equiv 2 \pmod 7$ is an element of order 3, and $3^3 \equiv 6 \pmod 7$ is an element of order 2. These are examples of a more general concept: if g generates $\mathbb{Z}/p\mathbb{Z} - \{0\}$ (so has order $p-1$) and ℓ divides $p-1$, then $g^{\frac{p-1}{\ell}} \pmod p$ has order ℓ (exercise: prove this); this gives an easy way of finding elements of a given order–this is going to be very useful in the following example.

Example Suppose that we want to solve the following Discrete Logarithm Problem: find $a \in \mathbb{Z}$ such that $2^a \equiv 17 \pmod{37}$, and you are given that 2 is a generator of the multiplicative group $(\mathbb{Z}/37\mathbb{Z})^*$. Then as a is in the exponent, it suffices to compute $a \pmod{36}$ (because of Fermat's Little Theorem). If we want to compute $a \pmod{36}$, by the Chinese Remainder Theorem it suffices to compute $a \pmod 4$ and $a \pmod 9$. This is something we can just do by brute force and observation, but there is a more effiicient way using the group theory above:

- To compute $a \pmod 4$, we first compute $a \pmod 2$. Write $a = a_0 + 2a_1$ where $a_0 \in \{0, 1\}$. By the above, we know that $2^{(p-1)/2} = 2^{18}$ is an element of order 2. Substituting for a, a_0, and a_1, we get the following equalities mod 37

  $$-1 \equiv 17^{18} \equiv (2^a)^{18} \equiv 2^{18a_0 + 36a_1} \equiv (2^{18})^{a_0} \cdot (2^{36})^{a_1} \equiv (-1)^{a_0},$$

  from which we can read off that $a_0 = 1$, so $a \equiv 1 \pmod 2$.

- Now we compute $a \pmod 4$. We know that $a \equiv 1 \pmod 2$, so there exist a_1, a_2 with $a_1 \in \{0, 1\}$ such that $a = 1 + 2a_1 + 4a_2$. By the above, we know that $2^{(p-1)/4} = 2^9$ is an element of order 4. Substituting for a, a_1, a_2, we get the following equalities mod 37

  $$31 \equiv 17^9 \equiv (2^{1+2a_1+4a_2})^9 \equiv 2^9 \cdot (2^{18})^{a_1} \cdot (2^{36})^{a_2} \equiv 6 \cdot (-1)^{a_1},$$

  from which we can read off that $a_1 = 1$, so $a \equiv 3 \pmod 4$.

- To compute $a \pmod 9$, we first compute $a \pmod 3$. Write $a = a_0 + 3a_1$ where $a_0 \in \{0, 1, 2\}$. By the above, we know that $2^{(p-1)/3} = 2^{12}$ is an

element of order 3. Substituting for a, a_0, a_1 we get the following equations mod 37

$$26 \equiv 17^{12} \equiv (2^a)^{12} \equiv 2^{12a_0+36a_1} \equiv (2^{12})^{a_0} \cdot (2^{36})^{a_0} \equiv 26^{a_0},$$

from which we can read off that $a_0 = 1$.

- Now we compute $a \pmod 9$. We know that $a \equiv 1 \pmod 3$, so there exist a_1, a_2 with $a_1 \in \{0, 1, 2\}$ such that $a = 1 + 3a_1 + 9a_2$. By the above, we have that $2^{(p-1)/9} = 2^4$ is an element of order 9. Substituting for a, a_1, a_2, we get the following equations mod 37

$$12 \equiv 17^4 \equiv (2^a)^4 \equiv 2^{4+12a_1+36a_2} \equiv 2^4 \cdot (2^{12})^{a_1} \cdot (2^{36})^{a_2} \equiv 16 \cdot 26^{a_1},$$

from which we can read off that $a_1 = 2$, so $a \equiv 7 \pmod 9$.

- Now we know that $a \equiv 3 \pmod 4$ and $a \equiv 7 \pmod 9$, which by CRT we know uniquely defines $a \pmod{36}$. We can compute this via Euclid's algorithm as we've done before, giving $a \equiv 7 \pmod{36}$.

The above method is in no way specific to the numbers we have chosen (37, 4, 9, etc): it is in fact an example of an algorithm that works in general. This trick of using the Chinese Remainder Theorem to attack the Discrete Logarithm Problem is due to Pohlig and Hellman and is therefore referred to as the Pohlig-Hellman algorithm.

3 Pohlig-Hellman

The Pohlig-Hellman algorithm (read: CRT) is a method to solve a discrete logarithm problem: given a prime p and $g \in \mathbb{Z}/p\mathbb{Z} - \{0\}$ of order $p-1$, and given $g^a \pmod p$, find a.

Now as, by Fermat's Little Theorem, for any $k \in \mathbb{Z}$ we have that $g^{a+(p-1)k} \equiv g^a \pmod p$, it suffices to find $a \pmod{p-1}$. So just as in the example above, we factorise $p-1$ into prime powers as

$$p - 1 = q_1^{e_1} \cdots q_r^{e_r},$$

where the q_i are prime. Then by the Chinese Remainder Theorem, to compute $a \pmod{p-1}$ it suffices to compute $a \pmod{q_1^{e_1}}, \ldots, a \pmod{q_r^{e_r}}$. The algorithm is as follows:

1. Factorize $p-1$ into prime powers as

$$p - 1 = q_1^{e_1} \cdots q_r^{e_r},$$

where the q_i are prime.

2. For each $i = 1, \ldots, r$,

(i) Write $a = a_0 + a_1 q_i + a_2 q_i^2 + \cdots$, with $a_j \in [0, q_i - 1]$.

(ii) Compute a_0, ie., $a \pmod{q_i}$: Note that

$$(g^a)^{\frac{p-1}{q_i}} \equiv (g^{\frac{p-1}{q_i}})^a \equiv (g^{\frac{p-1}{q_i}})^{a_0} \cdot (g^{p-1})^{(\cdots)} \equiv (g^{\frac{p-1}{q_i}})^{a_0} \pmod{p},$$

so in particular

$$(g^a)^{\frac{p-1}{q_i}} \equiv (g^{\frac{p-1}{q_i}})^{a_0} \pmod{p},$$

and the values in red are all things we can compute. Just checking the q_i options for a_0 gives us a_0, and hence $a \pmod{q_i}$.

(iii) For $k = 1, \ldots, e_i - 1$:
Given $a_0, \ldots a_{k-1}$, ie., $a \pmod{q_i^k}$, compute a_k, i.e., compute $a \pmod{q_i^{k+1}}$:
Note that

$$(g^a)^{\frac{p-1}{q_i^{k+1}}} \equiv (g^{\frac{p-1}{q_i^{k+1}}})^a \equiv (g^{\frac{p-1}{q_i^{k+1}}})^{a_0 + q_i a_1 + \cdots + q_i^{k-1} a_{k-1}} \cdot (g^{\frac{p-1}{q_i}})^{a_k} \cdot (g^{p-1})^{(\cdots)} \pmod{p},$$

so in particular

$$(g^a)^{\frac{p-1}{q_i^{k+1}}} \equiv (g^{\frac{p-1}{q_i^{k+1}}})^{a_0 + q_i a_1 + \cdots + q_i^{k-1} a_{k-1}} \cdot (g^{\frac{p-1}{q_i}})^{a_k} \pmod{p},$$

and the values in red are all things we can compute. Just checking the q_i options for a_k gives us a_k, and hence $a \pmod{q_i^{k+1}}$.

3. Using Euclid's algorithm, compute $a \pmod{p-1}$ from the values $a \pmod{q_1^{e_1}}, \ldots, a \pmod{q_r^{e_r}}$.

The *complexity* of this algorithm will depend on ℓ, where ℓ is the largest prime dividing $(p-1)$, or more precisely the number of basic operations for this algorithm will be a polynomial in ℓ.

Of course this attack can therefore be thwarted by choosing a prime p such that there is at least one large prime dividing $p - 1$. But, this is a bit of a problem for the ideas we had for efficient computations mod p: remember we were also using the Chinese Remainder Theorem to make our computations more efficient for encryption.

So, we need a bit more choice in how to set up our cryptosystems: Instead of just using exponentiation mod p we can use exponetiation in some more general contexts. More on this later in the course.

Cryptology Autumn 2022 Week 7
Digital Signatures

Dr Chloe Martindale

These notes are for the lecture course Cryptology at the University of Bristol. If you come across any typos, please email chloe.martindale@bristol.ac.uk.

1 RSA signatures

Now that we have a basic understanding of RSA, we consider briefly how to use the same mathematical ideas to create a digital signature. On an abstract level, a digital signature has the following basic setup:

Assume that you, the signer, have already generated a key pair (sk, pk) and published your public key pk as your identity and there is a message m (already in the form of a bit string) that you wish you sign. It is then a basic two-step process:

1. **Sign**: You use a signing function $(sk, m) \rightsquigarrow sig$ and send the signed message together with your identity (sig, pk) to the verifier.
2. **Verify**: The verifier uses a verifier function $(sig, pk) \rightsquigarrow m$ to check the signature matches your identity.

Note that unlike a PKE system, the important functionality here is that nobody can impersonate the signer: so nobody should be able to compute sig or sk given pk and m. RSA signatures fill in these wiggly arrows as follows:

Setup:

1. Signer: Generate an RSA key pair $sk, pk = (d, n), (e, n)$ and publish your identity (e, n).
2. Verifier/anyone: Generate a message $m \pmod n$ to be signed.

Sign:

1. Compute $sig \equiv m^d \pmod n$.
2. Send $(sig, (e, n))$ to the verifier.

Verify:

1. Check that $m \equiv sig^e \pmod{n}$.

As before, this is a mathematically consistent scheme because of Fermat's Little Theorem:

$$sig^e \equiv (m^d)^e \equiv m^{de} \equiv m^{1+k\varphi(n)} \equiv m \cdot (m^{\varphi(n)})^k \equiv m \pmod{n}.$$

Also, the only way to send the signed message corresponding to a given public key (e, n) is to know the secret (d, n), so this scheme is as secure as RSA.

2 ElGamal signatures

We have seen one digital signature scheme with RSA. We now consider a different digital signature scheme, that is a scheme with the same functionality is RSA signatures but with different mathematical techniques. Recall the abstract notion of a digital signature scheme from the above section on RSA signatures.

The algorithm is in three parts, as before, setup, sign, and verify. Remember that unlike with message encryption, the message in this protocol (m) is public information, such as a contract to be signed.

Setup

1. Choose a prime p and an element $g \in \mathbb{Z}/p\mathbb{Z} - \{0\}$ that generates $\mathbb{Z}/p\mathbb{Z} - \{0\}$ as a multiplicative group. (Remember, that means that
$$\mathbb{Z}/p\mathbb{Z} - \{0\} = \{g \pmod{p}, g^2 \pmod{p}, \ldots, g^{p-1} \pmod{p}\}.)$$
2. The signer Alice generates a (Diffie-Hellman-style) key pair $(sk, pk) = (a, g^a \pmod{p})$, where $a \in [0, p-1]$ is an integer, and publishes pk as her identity.
3. The verifier (or anyone) generates a message $m \pmod{p-1}$ to be signed.

Sign

1. Pick a random integer nonce (*number that you use once*) $k \in [0, p-1]$.
2. Compute $r = g^k \pmod{p}$.
3. Compute $sig \equiv k^{-1}(m - ar) \pmod{p-1}$.
4. Publish signed message (r, sig).

Verify

1. The verifier checks that $g^m \equiv pk^r \cdot r^{sig} \pmod{p}$.

Observations

- Note that the verification step works out just by unrolling all the notation:

$$pk^r \cdot r^{sig} = g^{ar} \cdot g^{k \cdot k^{-1}(m-ar)} = g^m.$$

- Observe that, unlike any of the messages in the other protocols we've seen, we defined our message $m \pmod{p-1}$, not mod p. This is because the message appears in the *exponent* in this protocol. Think about when two messages m and m' are equivalent in the verification: that is when $g^m \equiv g^{m'} \pmod{p}$, which is exactly when m and m' differ by a multiple of $p-1$ (because g has order $p-1$; we'll recall what order means in this context just below). That is, we only need to know the integer m modulo $p-1$.

- Observe that if an attacker knows the nonce k, they can recover the secret key a and consequently could imitate the signer, breaking the protocol. Remember that r, sig, p, and m are public values, so rearranging the equation defining sig in step 3 of 'Sign' will give the secret key a.

- You may be wondering why we use k only once, and not twice (or more times). The reason is that if you use k more than once then the attacker can actually recover k and hence also the secret key a by the previous point. To illustrate this, suppose that you sign messages m_1 and m_2 using the same nonce k. This will give signatures (r, sig_1) and (r, sig_2), where

$$sig_1 \equiv k^{-1}(m_1 - ar) \pmod{p-1}$$

and

$$sig_2 \equiv k^{-1}(m_2 - ar) \pmod{p-1}.$$

Solving these simultaneous equations then gives

$$k \equiv \frac{m_1 - m_2}{sig_1 - sig_2} \pmod{p-1},$$

so never reuse your nonce!

Cryptology Autumn 2022 Week 8
Algorithms for the Discrete Logarithm Problem

Dr Chloe Martindale

These notes are for the lecture course Cryptology at the University of Bristol. If you come across any typos, please email chloe.martindale@bristol.ac.uk.

1 Baby-Step-Giant-Step

We have already seen that if you want to find discrete logarithms in $\mathbb{F}_p^*$ and $p-1$, the size of the multiplicative group $\mathbb{F}_p^*$, has only small factors, you can do this very effectively using Pohlig-Hellman.

However, if you choose a finite field $\mathbb{F}_p$ such that there is a large prime ℓ dividing $p-1$, then you can also find an element $g \in \mathbb{F}_p$ of order ℓ where Pohlig-Hellman won't help you. So the question is: can we do better than brute-force? Below we will see two algorithms, Baby-Step-Giant-Step and Pollard-ρ, that are essentially clever methods for brute-forcing.

As always, we want to break the discrete logarithm problem, so suppose you have $g \in \mathbb{F}_p^*$ of order ℓ (not necessarily prime). Then, given g and g^a, find a. Remember that changing a by adding a multiple of ℓ amounts to multiplying g^a by $g^\ell = 1$, so it suffices to compute $a \pmod{\ell}$.

The algorithm is as follows:

1. For i from 0 to $\lfloor\sqrt{\ell}\rfloor$, compute and save $b_i = g^i$.
2. For j from 0 to $\lfloor\sqrt{\ell}+1\rfloor$, compute $c_j = g^a \cdot g^{-\lfloor\sqrt{\ell}\rfloor \cdot j}$; break if there exists an i such that $c_j = b_i$.
3. Return $a = i + \lfloor\sqrt{\ell}\rfloor \cdot j$.

Example Compute a such that $3^a \equiv 37 \pmod{101}$. The order of 3 in $\mathbb{F}_{101}^*$ is 100, so $\ell = 100$ and $\sqrt{\ell} = 10$. From step 1, the 'baby step', we get a table of values

i	0	1	2	3	4	5	6	7	8	9	10
$b_i = 3^i \pmod{101}$	1	3	9	27	81	41	22	66	97	89	65

Note that computing this table costs $10 = \sqrt{\ell}$ multiplications.

For step 2, the 'giant step', $c_0 = g^a$ is easy. For $c_1 = 3^a \cdot 3^{-10}$, we have to compute one inversion and exponentiate to get the 10th power, but we save the values c_1 and 3^{-10}. For $c_2 = 3^a \cdot 3^{-20}$, we first observe that $c_2 = c_1 \cdot 3^{-10}$, and we saved the values c_1 and 3^{-10}, so this just costs one multiplication, as will the computation of every c_j after this by a similar argument. So we get

$$\begin{aligned} c_0 &= 37, \\ c_1 &= 13, \\ c_2 &= 81, \end{aligned}$$

at which point we stop because $c_2 = b_4$, or in order words

$$3^a \cdot 3^{-20} \equiv 3^4 \pmod{101},$$

so $a = 24$.

As is illustrated in the example, the baby-step-giant-step algorithm costs at most $2\sqrt{\ell}$ multiplications plus some set up costs for the inversion and exponentiation to the $\lfloor\sqrt{\ell}\rfloor^{\text{th}}$ power (for example using square-and-multiply. As ℓ grows, these setup costs become negligable, so we say that 'the complexity of baby-step-giant-step is about $O(\sqrt{\ell})$' – any constants multiplying the $\sqrt{\ell}$ or added to it disappear in the big O.

This gives a square root speed up on just brute forcing – if your ℓ has 2000 bits then $\sqrt{\ell}$ only has 1000 bits – but at the expense of a pretty serious memory assumption: baby-step-giant-step also requires $O(\sqrt{\ell})$ storage. The next algorithm solves that problem.

2 Pollard's ρ method

Again, we want to solve the discrete logarithm problem: given g and $g^a \in \mathbb{F}_p^*$, find a.

The aim of Pollard's ρ method is to output integers $b, c, b', c' \in \{1, \ldots, \ell\}$ such that $c \neq c'$ and

$$g^b (g^a)^c = g^{b'} (g^a)^{c'}. \tag{1}$$

Why does this solve our problem? Well, suppose that the order of g in $\mathbb{F}_p^*$ is ℓ. Then, as before, adding ℓ to the exponent is equivalent to multiplying by $g^\ell = 1$, so taking logarithms of our equation (1) gives us:

$$b + ac \equiv b' + ac' \pmod{\ell}.$$

Rearranging this equation then gives us the secret a:

$$a \equiv \frac{b - b'}{c' - c} \pmod{\ell},$$

thus solving the discrete logarithm problem.

So, how exactly do we find such b, c, b', and c'? To do this, we define a *graph* G with vertices $G_i \in \mathbb{F}_p$ such that for each i there exists b_i and c_i such that $G_i = g^{b_i} g^{c_i}$. We define G_i, b_i, and c_i iteratively, and once we've found $i \neq j$ with $G_i = G_j$, we have candidates for b, c, b', c' satisfying (1), namely $b = b_i$, $c = c_i$, $b' = b_j$, $c' = c_j$.

The iterative sequence that turns out to be the most efficient to do this is:

$$G_0 = g,\ b_0 = 1,\ c_0 = 0,$$

and

$$(G_{i+1}, b_{i+1}, c_{i+1}) = \begin{cases} (G_i \cdot g, b_i + 1, c_i) & G_i \equiv 0 \pmod 3, \\ (G_i \cdot g^a, b_i, c_i + 1) & G_i \equiv 1 \pmod 3, \\ (G_i^2, 2b_i, 2c_i) & G_i \equiv 2 \pmod 3. \end{cases}$$

You might think the $\pmod 3$ looks a bit random, but this is basically just a way of making sure the choice of how to iterate changes around a bit.

Example. Compute a such that $3^a \equiv 7 \pmod{17}$. The order of 3 in $\mathbb{F}_{17}$ is 16, so $\ell = 16$. The algorithm above outputs a list:

$$\begin{aligned} G_0, b_0, c_0 &= 3, 1, 0 \\ G_1, b_1, c_1 &= 9, 2, 0 \\ G_2, b_2, c_2 &= 10, 3, 0 \\ G_3, b_3, c_3 &= 2, 3, 1 \\ G_4, b_4, c_4 &= 4, 6, 2 \\ G_5, b_5, c_5 &= 11, 6, 3 \\ G_6, b_6, c_6 &= 2, 12, 6, \end{aligned}$$

at which point we terminate because $G_6 = G_3$. In particular, this means that

$$g^{b_6} \cdot (g^a)^{c_6} = g^{b_3} \cdot (g^a)^{c_3},$$

which plugging in the values gives

$$3^3 \cdot (3^a) \equiv 3^{12} \cdot (3^a)^6,$$

giving $a \equiv 9 \cdot (-5)^{-1} \equiv 11 \pmod{16}$.

Pollard's ρ algorithm terminates after about $\sqrt{\frac{\pi}{2}\ell}$ steps, so also costs $O(\sqrt{\ell})$ multiplications, but requires only constant storage, which we typically notate by $O(1)$, so is better than baby-step-giant-step for memory reasons and is the algorithm typically used in practise.

Both baby-step-giant-step and Pollard ρ are what we refer to as 'generic' algorithms: they're not using anything particular about the structure of the group $\mathbb{F}_q^*$ or the choice of g for example–both are essentially just same methods for brute forcing.

In our context, that is in finite fields, there is another algorithm that beats both of these generic algorithms.

3 Index calculus

Again, we want to solve the discrete logarithm problem: given g and $g^a \in \mathbb{F}_q$, find $a = \log_g(g^a)$. This is the last algorithm we will see to attack this problem (and also the most efficient in this setting). We will first look at an example and then work out how to write down a general algorithm from that example.

Example Suppose you are given that 17 has order 106 in $\mathbb{F}_{107}^*$, and that $17^a \equiv 91 \pmod{107}$, and you want to compute a, i.e., you want to compute $\log_{17}(91)$. With the index calculus algorithm, the first thing you do is choose a *factor base* $\mathcal{F}$, which can contain any (and as many) primes (as) you like; here we will choose

$$\mathcal{F} = \{2, 3, 5\}.$$

We then compute $\log_{17}(n)$ for every $n \in \mathcal{F}$ in the following way: compute and factorise $17^i \pmod{107}$ for increasing i until you have found $3 = |\mathcal{F}|$ equations for the $\log_{17}(n)$. In this example, we get

$$17^2 \equiv 3 \cdot 5^2 \pmod{107},$$

which taking logs gives

$$2 \equiv \log_{17}(3) + 2\log_{17}(5) \pmod{106}, \tag{2}$$

then $17^3, \ldots, 17^8$ all have factors which are not in $\mathcal{F}$, but

$$17^9 \equiv 2^2 \cdot 5 \pmod{107},$$

which taking logs gives

$$9 \equiv 2\log_{17}(2) + \log_{17}(5) \pmod{106}, \tag{3}$$

and finally

$$17^{11} \equiv 2 \pmod{107},$$

which taking logs gives

$$11 \equiv \log_{17}(2) \pmod{106}. \tag{4}$$

Solving the three simultaneous equations (2), (3), and (4) gives

$$\begin{aligned} \log_{17}(2) &\equiv 11 \pmod{106}, \\ \log_{17}(3) &\equiv 28 \pmod{106}, \\ \log_{17}(5) &\equiv 93 \pmod{106}. \end{aligned}$$

So now we've found these values, what do we do with them? We want to be able to write the discrete log we're actually interested in, namely $\log_{17}(91)$, in terms of the discrete logs we now know, namely $\log_{17}(n)$ for $n \in \mathcal{F}$, and of course $\log_{17}(17^j)$ $(= j)$ for small values of j. We can play the same game as

above: try multiplying 91 with 17^j for small values of j and factorizing until we find a number with only factors from our factor base. Doing this we see that $17^0 \cdot 91, \ldots, 17^4 \cdot 91$ yields nothing but

$$17^5 \cdot 91 \equiv 2^2 \cdot 5^2 \pmod{107},$$

which taking logs gives

$$5 + \log_{17}(91) \equiv 2\log_{17}(2) + 2\log_{17}(5) \pmod{106},$$

and plugging in the values above this gives us that

$$a = \log_{17}(91) = 97.$$

So, let's summarize our method into a more general algorithm. Suppose you are given g and $g^a \in \mathbb{F}_p$ and you want to compute a. Then

1. Choose your factor base $\mathcal{F} = \{p_1, \ldots, p_n\}$.
2. Compute, for each $i = 1, \ldots, n$, $\log_g(p_i)$:
   (a) For increasing $j \geq 1$, factorize g^j. Break when you have found n values of j for which all the factors of g^j are in $\mathcal{F}$.
   (b) Take logs of the n equations for values g^j with all factors in $\mathcal{F}$ to get n simultaneous equations for $\log_g(p_1), \ldots, \log_g(p_n)$.
   (c) Solve your n simultaneous equations to get $\log_g(p_1), \ldots, \log_g(p_n)$.
3. For increasing $j \geq 0$, factoring $g^j \cdot g^a$. Break when all factors of j are in $\mathcal{F}$.
4. Take logs of the equation from the previous step, and solve for a.

This algorithm is by far the most efficient known algorithm for this setting of the discrete logarithm problem. To write down the complexity, we introduce a notation:

$$L_N(\alpha, c) = e^{c \log N^\alpha \log\log N^{1-\alpha}},$$

where $\alpha \in [0, 1]$

This may look strange, but consider what happens when $\alpha = 0$ or 1: $L_N(0, c) = \log(N)^c$ or $L_N(1, c) = N^c$, corresponding to polynomial time or exponential time. So the closer α is to 0, the closer an algorithm is to being polynomial time, and the closer it is to 1, the closer an algorithm is to being exponential time.

The most optimized version of the index calculus algorithm (containing many many details not covered here) has complexity $L_p(1/3, c)$, where the constant c depends very heavily on the conditions, so that's closer to the polynomial time end than the exponential end, but the different is still (asymptotically) big enough for powers of large primes that it is possible to make use of finite

fields in cryptography by scaling up the numbers. In particular, scaling up p to at least 3000 bits for 128-bit security, meaning that it should take about 2^{128} bit operations to break the protocol. Compare this to our Pollard ρ algorithm which takes about $\sqrt{p}$ bit operations to break the protocol–so we would need p to be about 256 bits, and you see how much different the index calculus makes.

There are other examples of groups in which the index calculus is less effective or not at all which are currently the most commonly used in practise, namely elliptic curve groups, where the size of your group can indeed be only about 256 bits instead of 3000. But that is beyond the scope of this course!