Lecture 3 – Modular Arithmetic and Diffie-Hellman

Chloe Martindale

2025/26

In previous lectures we have seen how to set up secure communication *given a shared secret value*. In the next lecture, we will see how to make this communication more efficient using block ciphers. In this lecture, we'll consider how to distribute this secret value using mathematics. In the modern world, you are attempting to communicate securely with many different parties: servers on the other side of the world, family in another country, companies, governments, hospitals, the list goes on. So how can we use mathematics to share a secret value cheaply, easily, and without ever meeting? This is the premise of *public key cryptography*. This lecture will build up the mathematical foundations necessary to understand some ways in which this is done in practice and introduce the Diffie Hellman key exchange protocol.

3.1 Modular arithmetic

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 3.1. 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[1]:

Definition 3.2. 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 when 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 $4 \cdot n \equiv 1 \pmod 8$.

[1] Technically we are introducing *multiplicative* inverses here (analogous to the additive inverse that we saw in Worksheet 1). As multiplicative inverses are clearly more interesting that additive inverses, we tend to drop the adjective.

- Exercise: generalise 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.*[2]

3.2 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: $i \leftarrow i + 1$.
4: Compute[a] the unique m_i and $r_i \in \mathbb{Z}$ such that $0 \leq r_i < r_{i-1}$ and

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

[a] This is called *division-with-remainder.*

5: **return** r_{i-1}

Euclid's algorithm returns the greatest common divisor (gcd) of the input numbers a and b; if $\gcd(a, b) = 1$, then b is invertible mod a. Moreover, from Euclid's algorithm, we can recover the inverse. This is stated and proven in the following corollary.

Corollary 3.1 (Euclid's corollary[3])**.** *Let a and b be integers. If $d = \gcd(a, b)$ then there exist $m, n \in \mathbb{Z}$ such that*

$$am + bn = d.$$

Proof. 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 $r_k = d$. □

Now we have a new method to compute the inverse of a mod b, as long as a and b are coprime: Use Euclid's algorithm to compute m and n such that $am + bn = 1$. Then, modulo b, we have

$$am \equiv 1 \pmod b,$$

or in other words m is the inverse of a mod b.

[2] Most likely not due to Euclid, but Euclid wrote about it.
[3] Often referred to just as Euclid's algorithm.

Example.
Let's see an example of how to use Euclid's algorithm to compute an inverse. Suppose that you want to compute the inverse of $11 \pmod{17}$.

Run Euclid's algorithm with $r_0 = 17$ and $r_1 = 11$:

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

Then reverse engineer the algorithm to get:

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

In other words,

$$2 \cdot 17 - 3 \cdot 11 = 1,$$

so in particular

$$-3 \cdot 11 \equiv 1 \pmod{17},$$

so the inverse of $11 \pmod{17}$ is $-3 \equiv 14 \pmod{17}$.

3.3 Sun-Tzu's Remainder Theorem

There are many surprising constructions and consequences of modular arithmetic, and we now present a seminal theorem in modular arithmetic which turns out to be a key tool in cryptanalysis.

Theorem 3.2 (Sun-Tzu's Remainder Theorem (SRT)[4]). *Given coprime $n, m \in \mathbb{Z}_{>1}$ and $a, b \in \mathbb{Z}$ there exist $c, d \in \mathbb{Z}$ such that*

$$cm + dn = 1 \tag{3.1}$$

and

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

is the only number $\pmod{mn}$ 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 check that the given construction is valid.

[4]Most textbooks refer to this as the 'Chinese Remainder Theorem'. It is most likely not due to Sun-Tzu, but Sun-Tzu wrote about it.

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

$$cm + dn = 1. \tag{3.2}$$

We claim that

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

will work. We first check mod n. Note that $cm = 1 - dn$ by (3.2). So

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

Similarly

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

□

Example
Now suppose that you are given the equations

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

and

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

and you want to find the $x \pmod{17 \cdot 11}$ that reduces mod 17 and 11 to these values. We already saw in our example of computing inverses using Euclid's algorithm that

$$2 \cdot 17 - 3 \cdot 11 = 1.$$

Now using SRT, 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

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

3.4 Groups

Sets of integers equipped with addition modulo n are examples of *groups*. Groups are central to the construction of public-key cryptography – we'll see how to define a secure key exchange based on a group with certain properties. This key exchange (the Diffie-Hellman key exchange) is fundamental in every widely used protocol on the internet today (TLS 1.3, Signal, etc). Here we just give the definition and some examples of groups to familiarise ourselves with the concept.

Definition 3.3. 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

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

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

Another word for this is to say that g *generates* $(\mathbb{Z}/p\mathbb{Z})^*$.

Definition 3.4. Let $(G, *)$ be a group. We say that $g \in G$ *generates* G if

$$G = \{g, g * g, g \underbrace{* \cdots *}_{|G| \text{ times}} g\}.$$

We then call g a *generator*.

For example, if $p = 5$ it turns out that we can choose $g = 2$:

$$(\mathbb{Z}/5\mathbb{Z})^* = \{2 \pmod 5, 4 \equiv 2^2 \pmod 5, 3 \equiv 2^3 \pmod 5, 1 \equiv 2^4 \pmod 5\}.$$

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: This is from *Fermat's Little Theorem*:

Theorem 3.3 (Fermat's Little Theorem)**.** *Let p be a prime. For every $a \in (\mathbb{Z}/p\mathbb{Z})^*$,*

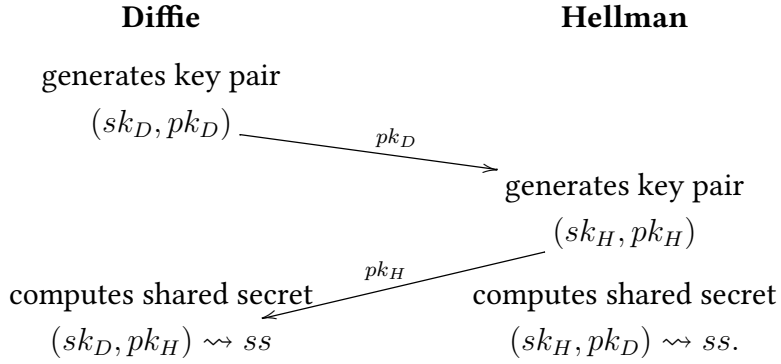
$$a^{p-1} \equiv 1 \pmod p.$$

We'll work with this more next week.

3.5 Diffie-Hellman key exchange

In Lecture 1 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.

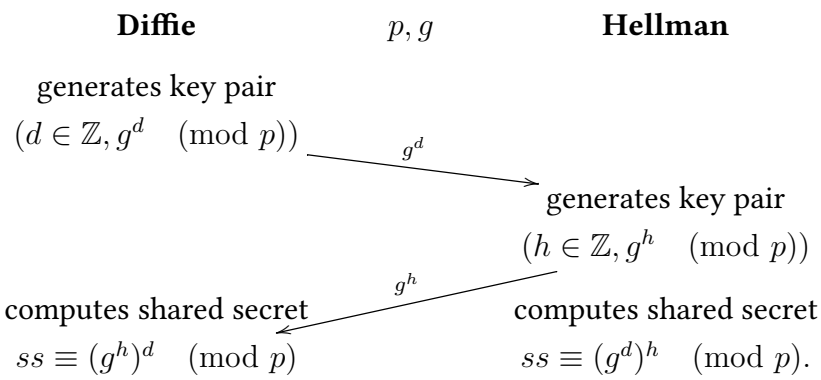
The abstract idea of a key exchange is as follows: suppose Diffie and Hellman want compute a shared secret key (read: a shared value that no-one else can compute).



Here is a(n 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 non-zero 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' or *polynomial-time* 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 $O(2^\lambda)$ or $O(\lambda^\alpha \log_2(\lambda)^{1-\alpha})$ for some $\alpha \in (0, 1)$ respectively[5]; these are referred to *exponential* and *subexponential* algorithms 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).

3.6 The Discrete Logarithm Problem

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 will later see is possible with square-and-multiply.

[5] A reminder on Big-Oh notation: $x = O(f(x))$ for some function f means that there exists constants $N, c > 0$ so that $x \leq cf(x)$ for all $x \geq N$. In the context of algorithmic run-time, if the function f is a polynomial in x, then we say that the algorithm runs in polynomial time.

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 $g^d \pmod p$.[6]

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.

For the Diffie-Hellman key exchange, our key space is $(\mathbb{Z}/p\mathbb{Z})^*$, where p is a prime, which we saw last week is defined by

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

we saw also that $(\mathbb{Z}/p\mathbb{Z})^*$ is a cyclic group under multiplication $\pmod p$. That is, there exists a $g \pmod p$ such that

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

The reason that it is important for Diffie–Hellman that $(\mathbb{Z}/p\mathbb{Z})^*$ is cyclic 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 $(\mathbb{Z}/p\mathbb{Z})^*$ (or a subgroup of that, but for simplicity we ignore that for now).

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.

[6] 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*.