What is Computer Science?

An Information Security Perspective

Daniel Page ⟨dan@phoo.org⟩ and Nigel P. Smart ⟨csnps@bristol.ac.uk⟩

git # 148def3 @ 2018-07-11

SAFETY IN NUMBERS: MODERN CRYPTOGRAPHY FROM ANCIENT ARITHMETIC

Think back to the k-place shift encryption scheme that was introduced in Chapter 7, and consider two **parties**, a **sender** called Alice and a **receiver** called Bob, using it to communicate with each other. We might describe how Alice and Bob behave using a diagram:

Alice	Eve	Bob
$C \leftarrow \textsc{Enc}_k(M)$		
	$\xrightarrow{C}$	
		$M \leftarrow \textsc{Enc}_k(C)$

This description is a simple cryptographic **protocol**: it describes the computational steps each party performs and the communication steps that occur between them. Note that we *could* have named the parties Angharad and Bryn but, somewhat bizarrely, an entire menagerie of standard names are used within protocol descriptions of this sort [1]. Unless we use Alice and Bob, cryptographers get confused as to what their roles are! In this case, the protocol that Alice and Bob engage in has three simple steps read from top-to-bottom:

1. Alice encrypts a plaintext, e.g., M = 'a', using the key $k = 3$ to produce the ciphertext message

   $$C = \text{'d'} = \textsc{Enc}_{k=3}(\text{'a'}).$$

2. Alice communicates C to Bob.

3. Bob decrypts the ciphertext, e.g., C = 'd', using the key $k = 3$ to recover the plaintext message

   $$M = \text{'a'} = \textsc{Dec}_{k=3}(\text{'d'}).$$

There are some important features of this protocol that demand further discussion:

- The diagram shows a third party: Eve is a passive **adversary** who hopes to learn M (or k) just by *observing* what Alice and Bob communicate to each other (in this case C). If Eve is allowed to *alter* the communication rather than just observe it, she turns into Mallory the active, malign adversary. We saw why this could be a problem at the start of Chapter 7: Thomas Phelippes was basically doing the same thing with messages communicated between Mary Stewart and Anthony Babington.

- With the encryption scheme we have, both M and C are single characters. Formally, we have **block cipher** [2] where each **block** is one character in size. What if Alice wants to send a *sequence* of characters (i.e., a string) to Bob? This is easy to accommodate: we simply apply the encryption scheme independently to each of the characters (i.e., the blocks) in the sequence. That is, take the i-th block of the plaintext and encrypt it to form the i-th block of the ciphertext

$$C_i = \textsc{Enc}_{k=3}(M_i)$$

and vice versa

$$M_i = \textsc{Dec}_{k=3}(C_i).$$

- Both Alice *and* Bob need to know and use the *same* shared key. We call this type of encryption scheme a **symmetric-key** [24] block cipher because the use of k is symmetric, i.e., the same for both Alice and Bob. In some circumstances this poses no problem at all, but in others it represents what we call the **key distribution problem**: given that Eve can see everything communicated between Alice and Bob, how can they decided on k in the first place?

Research (task #1)

The *way* we use a block cipher is, roughly speaking, called a **mode of operation** [3]. For example, although

$$C_i = \textsc{Enc}_k(M_i)$$

and

$$M_i = \textsc{Dec}_k(C_i).$$

describe the **Electronic Codebook (ECB)** mode, various alternatives also exist. Find out about at least one alternative, plus when and why it might be preferred. Try to write equations, similar to those above, that describe how the mode produces a block of ciphertext from the corresponding block of plaintext (and vice versa).

The focus of this Chapter relates to the final point above, and two potential solutions more specifically: **key agreement protocols** [15] and **public-key encryption schemes** [21]. Both are interesting from a historical point of view, but also underpin a widely-used application of modern cryptography: the **Secure Sockets Layer (SSL)** [25] system that permits modern e-commerce to exist in a usable form. If you open a web-browser and load a secure web-site (whose URL will often start with `https://` rather than `http://`), these technologies enable you to shop online without a *real* Eve finding out and using your credit card number!

Additionally, they both rely on a field of mathematics called **number theory** [17]. Many of the algorithms within this field have been studied for thousands of years; two of the most famous are Euclid's algorithm [8] for computing the greatest common divisor of two numbers, and the binary exponentiation algorithm [10] that we saw in Chapter 3. Both are often studied as examples in Computer Science because, aside from being useful, they have a number of attractive properties. In particular they

1. are short and fairly easy to understand (or at least explain),
2. allow simple arguments and proofs about their correctness, and
3. allow relatively simple analysis of their computational complexity.

Our aim is to demonstrate these properties and to show how these ancient algorithms support the modern cryptographic protocols on which we now rely.

1 Modular arithmetic: the theory

Chapter 7 described modular arithmetic as being like the "clock arithmetic" you need when reading the time from an analogue clock face such as Figure 1a. Based on someone saying "the time is now ten o'clock; I will meet you in four hours", our example question asked what time they mean? We reasoned that since

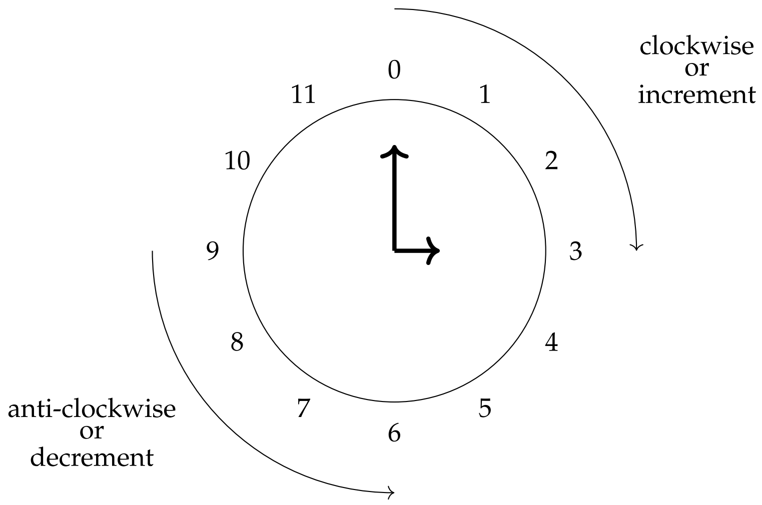
$$(10 + 4) = 14 \equiv 2 \pmod{12},$$

the answer is two o'clock. We started being more formal by saying that the $\equiv$ symbol means **equivalent to**, so writing

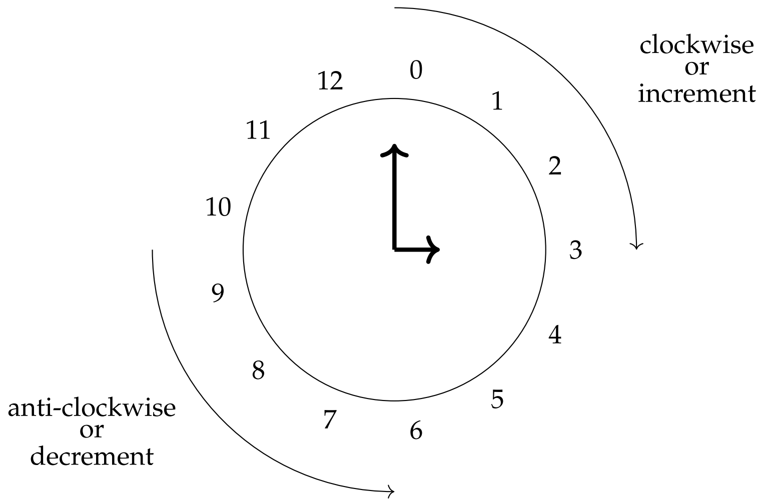
$$x \equiv y \pmod{N}$$

means x and y give the same remainder after division by the **modulus** N. This is the same as saying there exists some κ so that

$$x = y + \kappa \cdot N,$$



(a) *A 12-hour clock face.*



(b) *A 13-hour clock face.*

Figure 1: *Two analogue clock faces, one standard 12-hour version and a slightly odd 13-hour alternative.*

i.e., x equals y after some multiple of the modulus is added on. We can see this from our example: we can say

$$14 \equiv 2 \pmod{12}$$

precisely because for $\kappa = 1$

$$14 = 2 + 1 \cdot 12,$$

i.e., both 14 and 2 give the remainder 2 after division by 12.

1.1 Rules for modular addition

Although things look more complicated, many of the rules of addition and subtraction you already know still apply to modular arithmetic:

$$\begin{array}{lcll} x + y & \equiv & y + x & \pmod{N} \\ x + (y + z) & \equiv & (x + y) + z & \pmod{N} \\ x + 0 & \equiv & x & \pmod{N} \end{array}$$

In the last rule, 0 is called an **additive identity**. More generally, an **identity function** [13] gives the same output as the input. For example,

$$f(x) = x$$

means f is an identify function because whatever x we give it as input, the output is x as well. Here, you can think of the "add 0 function" as an identity function as well, i.e.,

$$f(x) = x + 0 \equiv x \pmod{N}.$$

A useful question to ask is whether for a given x we can find y, the **additive inverse** of x, which produces

$$x + y \equiv 0 \pmod{N}.$$

That is, can we find a y which when added to x produces the additive identity as a result? One way to answer this question is to just search through all possible values of y. If $x = 3$ and $N = 12$ for example, we can look at the following possibilities:

$$\begin{array}{ccccll@{\qquad}ccccrl} 3 & + & 0 & \equiv & 3 & \pmod{12} & 3 & + & 6 & \equiv & 9 & \pmod{12} \\ 3 & + & 1 & \equiv & 4 & \pmod{12} & 3 & + & 7 & \equiv & 10 & \pmod{12} \\ 3 & + & 2 & \equiv & 5 & \pmod{12} & 3 & + & 8 & \equiv & 11 & \pmod{12} \\ 3 & + & 3 & \equiv & 6 & \pmod{12} & 3 & + & 9 & \equiv & 0 & \pmod{12} \\ 3 & + & 4 & \equiv & 7 & \pmod{12} & 3 & + & 10 & \equiv & 1 & \pmod{12} \\ 3 & + & 5 & \equiv & 8 & \pmod{12} & 3 & + & 11 & \equiv & 2 & \pmod{12} \end{array}$$

The one we want is $y = 9$ which produces $3 + 9 \equiv 0 \pmod{12}$. But can we *always* find an additive inverse, no matter what x and N are? Ignoring modular arithmetic, the intuitive answer would be to set $y = -x$ meaning that $x + y = x - x = 0$. This also works for modular arithmetic. Remember from Chapter 7 that $-x \pmod{N} = N - x$, so setting $y = -x$ gives us

$$x - y \equiv x + (-y) \equiv x + (N - x) \equiv N \equiv 0 \pmod{N}.$$

In our example above, this is demonstrated by

$$3 - 3 \equiv 3 + (-3) \equiv 3 + (12 - 3) \equiv 12 \equiv 0 \pmod{N},$$

which produces exactly what we wanted.

1.2 Rules for modular multiplication

In Chapter 7 we only used modular addition, but it should be no great surprise that we can consider other operations as well. We saw in Chapter 3 that multiplication is just repeated addition, so it follows that modular multiplication could be similar.

As another example, suppose you started a job at ten o'clock and worked in three shifts each of two hours. When you finished, the time would be given by

$$10 + (3 \cdot 2) = 16 \equiv 4 \pmod{12},$$

i.e., you would have finished at four o'clock. Of course this is just the same as

$$10 + (2 + 2 + 2) = 16 \equiv 4 \pmod{12}.$$

We can again rely on existing rules for multiplication:

$$\begin{array}{lcll} x \cdot y & \equiv & y \cdot x & \pmod{N} \\ x \cdot (y \cdot z) & \equiv & (x \cdot y) \cdot z & \pmod{N} \\ x \cdot 1 & \equiv & x & \pmod{N} \end{array}$$

This time 1 is called a **multiplicative identity**; the "multiply by 1 function" is another identity function. Since this resembles the additive identity case we saw previously, we can ask the same question: can we always find a y so that $x \cdot y \equiv 1 \pmod{N}$? The answer this time is no, not always. If $x = 3$ and $N = 12$, we show this by searching through all values of y again:

$$\begin{array}{ccccll@{\qquad}ccccll} 3 & \cdot & 0 & \equiv & 0 & \pmod{12} & 3 & \cdot & 6 & \equiv & 6 & \pmod{12} \\ 3 & \cdot & 1 & \equiv & 3 & \pmod{12} & 3 & \cdot & 7 & \equiv & 9 & \pmod{12} \\ 3 & \cdot & 2 & \equiv & 6 & \pmod{12} & 3 & \cdot & 8 & \equiv & 0 & \pmod{12} \\ 3 & \cdot & 3 & \equiv & 9 & \pmod{12} & 3 & \cdot & 9 & \equiv & 3 & \pmod{12} \\ 3 & \cdot & 4 & \equiv & 0 & \pmod{12} & 3 & \cdot & 10 & \equiv & 6 & \pmod{12} \\ 3 & \cdot & 5 & \equiv & 3 & \pmod{12} & 3 & \cdot & 11 & \equiv & 9 & \pmod{12} \end{array}$$

None of the y gives us the result we want. Why?! Remember that finding such a y is the same as finding a κ which satisfies

$$3 \cdot y = 1 + 12 \cdot \kappa$$

or put another way,

$$3 \cdot y - 12 \cdot \kappa = 1.$$

The left-hand side of this equation is divisible by three, i.e., we can write it as

$$3 \cdot (y - 4 \cdot \kappa) = 1$$

instead, whereas the right-hand side is not; this means that no such κ can exist. What about some other values of x, say $x = 7$? This time things work out

$$\begin{array}{ccccll@{\qquad}ccccll} 7 & \cdot & 0 & \equiv & 0 & \pmod{12} & 7 & \cdot & 6 & \equiv & 6 & \pmod{12} \\ 7 & \cdot & 1 & \equiv & 7 & \pmod{12} & 7 & \cdot & 7 & \equiv & 1 & \pmod{12} \\ 7 & \cdot & 2 & \equiv & 2 & \pmod{12} & 7 & \cdot & 8 & \equiv & 8 & \pmod{12} \\ 7 & \cdot & 3 & \equiv & 9 & \pmod{12} & 7 & \cdot & 9 & \equiv & 3 & \pmod{12} \\ 7 & \cdot & 4 & \equiv & 4 & \pmod{12} & 7 & \cdot & 10 & \equiv & 10 & \pmod{12} \\ 7 & \cdot & 5 & \equiv & 11 & \pmod{12} & 7 & \cdot & 11 & \equiv & 5 & \pmod{12} \end{array}$$

and we find that for $y = 7$, $x \cdot y \equiv 1 \pmod{12}$. The reason is that given the equation

$$7 \cdot y - 12 \cdot \kappa = 1$$

we can write, using $\kappa = 4$,

$$7 \cdot 7 - 12 \cdot 4 = 1.$$

We are allowed to call this y the **multiplicative inverse** of the corresponding x: if you take x and multiply it with its multiplicative inverse, you end up with the multiplicative identity.

1.3 The sets $\mathbb{Z}$, $\mathbb{Z}_N$ and $\mathbb{Z}_N^\star$

The set of **integers** [14], or "whole numbers", is written using a special symbol:

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, +1, +2, +3, \ldots\}.$$

Clearly this is an infinite set: the continuation dots at the right- and left-hand ends highlight the fact that we can write down infinitely many positive and negative integers. With modular arithmetic, we deal with a sub-set of the integers. By using a modulus N, we only deal with 0 through to $N - 1$ because anything equal to or larger than N or less than 0 "wraps around" (as we saw with the number line in Chapter 7). More formally, we deal with the set

$$\mathbb{Z}_N = \{0, 1, 2, 3, \ldots, N - 1\},$$

which for $N = 12$ would obviously be

$$\mathbb{Z}_{12} = \{0, 1, 2, 3, \ldots, 11\}.$$

When you see $\mathbb{Z}_N$, take it to mean the set of integers modulo N; the *only* numbers we can work with are 0 through to $N - 1$.

Given an N we already know that for some $y \in \mathbb{Z}_N$, but not all, we can find a multiplicative inverse; this is a bit of a pain unless we know beforehand which values of y and N will work. For example, for $N = 12$ we can write out a table and fill in the multiplicative inverses for each y:

$$\begin{array}{rcccll@{\qquad}rcccll}
0 & \cdot & ? & \equiv & 1 & \pmod{12} & 6 & \cdot & ? & \equiv & 1 & \pmod{12} \\
1 & \cdot & 1 & \equiv & 1 & \pmod{12} & 7 & \cdot & 7 & \equiv & 1 & \pmod{12} \\
2 & \cdot & ? & \equiv & 1 & \pmod{12} & 8 & \cdot & ? & \equiv & 1 & \pmod{12} \\
3 & \cdot & ? & \equiv & 1 & \pmod{12} & 9 & \cdot & ? & \equiv & 1 & \pmod{12} \\
4 & \cdot & ? & \equiv & 1 & \pmod{12} & 10 & \cdot & ? & \equiv & 1 & \pmod{12} \\
5 & \cdot & 5 & \equiv & 1 & \pmod{12} & 11 & \cdot & 11 & \equiv & 1 & \pmod{12}
\end{array}$$

The question marks in the table highlight entries that we cannot fill in, with the values $y \in \{1, 5, 7, 11\}$ being those with a multiplicative inverse.

Recall that if some p is a **prime number** [20], it can only be divided exactly by 1 and p; we say 1 and p are the only **divisors** of p. Strictly speaking 1 is a prime number based on this definition, but 1 is not usually very interesting so we tend to exclude it. Two facts explain why the result above is no accident:

1. 2 and 3 are the two **prime divisors** of 12, i.e., the divisors of 12 which are prime. For example, 6 is a divisor of 12 but not prime, 5 is not a divisor but is prime, whereas 3 is both a divisor of 12 and prime.
2. $\{1, 5, 7, 11\}$ is precisely the set of elements in $\mathbb{Z}_{12}$ which are not divisible by 2 or 3. For example, 4 is not in the set because it is divisible by 2, whereas 5 is divisible by neither 2 nor 3 so it is in the set.

The special symbol $\mathbb{Z}_N^\star$ is used to represent the sub-set of $\mathbb{Z}_N$ for which we can find a multiplicative inverse; for our example this means

$$\mathbb{Z}_{12}^\star = \{1, 5, 7, 11\}.$$

Interestingly, the $y \in \mathbb{Z}_{12}^\star$ are integers we can safely divide by. Normally if we wrote $2/5$ you might say the result is 0.4, but keep in mind that we can *only* work with 0 through to $N - 1$: there are no fractional numbers. So actually, when we write $2/5 \pmod{12}$ this means

$$2/5 = 2 \cdot 1/5 = 2 \cdot 5^{-1} \equiv 2 \cdot 5 \equiv 10 \pmod{12}$$

which is "allowed" precisely because, as we saw, $1/5 \equiv 5^{-1} \equiv 5 \pmod{12}$, i.e., 5 has a multiplicative inverse modulo 12 which, coincidentally, is also 5.

Now look at a different modulus, namely $N = 13$. Something special happens, because if we follow the same reasoning as above we get

$$\begin{array}{lcl}
\mathbb{Z}_{13} & = & \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}, \\
\mathbb{Z}_{13}^\star & = & \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}.
\end{array}$$

$\mathbb{Z}_{13}$ and $\mathbb{Z}_{13}^\star$ look fairly similar: *all* non-zero $y \in \mathbb{Z}_{13}$ have a multiplicative inverse and appear in $\mathbb{Z}_{13}^\star$ as a result. We can see this by again writing out a table and filling in the multiplicative inverse for each y:

$$\begin{array}{rcccll@{\qquad}rcccll}
0 & \cdot & ? & \equiv & 1 & \pmod{13} & 6 & \cdot & 11 & \equiv & 1 & \pmod{13} \\
1 & \cdot & 1 & \equiv & 1 & \pmod{13} & 7 & \cdot & 2 & \equiv & 1 & \pmod{13} \\
2 & \cdot & 7 & \equiv & 1 & \pmod{13} & 8 & \cdot & 5 & \equiv & 1 & \pmod{13} \\
3 & \cdot & 9 & \equiv & 1 & \pmod{13} & 9 & \cdot & 3 & \equiv & 1 & \pmod{13} \\
4 & \cdot & 10 & \equiv & 1 & \pmod{13} & 10 & \cdot & 4 & \equiv & 1 & \pmod{13} \\
5 & \cdot & 8 & \equiv & 1 & \pmod{13} & 11 & \cdot & 6 & \equiv & 1 & \pmod{13}
\end{array}$$

There are no question marks (except for 0) this time, precisely because 13 is a prime number, and means we can divide by every non-zero y. You can think of this mirroring integer arithmetic, where dividing by zero is not allowed either. It also illustrates the fact that for $p = 12$, it was just a coincidence that $y^{-1} \equiv y \pmod{12}$ for all y with an inverse: clearly this is not true for $p = 13$, where we find $5^{-1} \equiv 8 \pmod{13}$ for instance.

1.4 Some interesting facts about $\mathbb{Z}_N^\star$

The set $\mathbb{Z}_N^\star$ is interesting because the number of elements it contains can be calculated, using just N, via the so-called **Euler** Φ (or "phi") function [9]. Specifically,

$$\Phi(N) = \prod_{i=0}^{l-1} p_i^{e_i - 1} \cdot (p_i - 1)$$

when the **prime factorisation** [19] of N is given by

$$N = \prod_{i=0}^{l-1} p_i^{e_i}$$

and where each p_i is a prime number. Basically we express N as the product of l prime numbers (or powers thereof). This is quite a nasty looking definition, but some examples should make things clearer:

- If we select $N = 12$, then the prime factorisation of N is

  $$N = 12 = 2 \cdot 2 \cdot 3 = 2^2 \cdot 3^1.$$

  We have expressed N as the product of $l = 2$ primes, namely 2 and 3. Matching this description with what we had above, this means $p_0 = 2$, $e_0 = 2$, $p_1 = 3$ and $e_1 = 1$ so

  $$N = \prod_{i=0}^{l-1} p_i^{e_i} = \prod_{i=0}^{2-1} p_i^{e_i} = p_0^{e_0} \cdot p_1^{e_1} = 2^2 \cdot 3^1 = 12.$$

  If we select $N = 13$ then things are even simpler, i.e.,

  $$N = 13 = 13^1,$$

  due to the fact that 13 *is* a prime number: the prime factorisation only includes 13 itself! That is, for $N = 13$ we just have $p_0 = 13$ and $e_0 = 1$ so

  $$N = \prod_{i=0}^{l-1} p_i^{e_i} = \prod_{i=0}^{1-1} p_i^{e_i} = p_0^{e_0} = 13^1 = 13.$$

- We can now compute

  $$\begin{array}{rclcccccc} \Phi(12) & = & 2^{2-1} & \cdot & (2-1) & \cdot & 3^{1-1} & \cdot & (3-1) \\ & = & 2^1 & \cdot & 1 & \cdot & 3^0 & \cdot & 2 \\ & = & 2 & & & \cdot & 2 & & \\ & = & 4 & & & & & & \end{array}$$

  and

  $$\begin{array}{rclcl} \Phi(13) & = & 13^{1-1} & \cdot & (13-1) \\ & = & 13^0 & \cdot & 12 \\ & = & 12 & & \end{array}$$

  Both these results make sense because looking back we know that

  $$\begin{array}{rcl} \mathbb{Z}_{12}^{\star} & = & \{1, 5, 7, 11\} \\ \mathbb{Z}_{13}^{\star} & = & \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\} \end{array}$$

  which match up: $\mathbb{Z}_{12}^{\star}$ contains $\Phi(12) = 4$ elements, and $\mathbb{Z}_{13}^{\star}$ contains $\Phi(13) = 12$ elements.

Another way to look at things is that $\Phi(N)$ represents the number of integers less than N which are also **coprime** [4] to N. Two integers a and b are coprime if their **Greatest Common Divisor (GCD)** is 1; some examples again make the idea clearer:

- If $a = 3$ and $b = 12$, the divisors of a are 1 and 3 while the divisors of b are 1, 2, 3 and 6. The greatest divisor they have in common is 3 so we can write

  $$\gcd(a, b) = \gcd(3, 12) = 3$$

  and say a and b are not coprime.

- If $a = 3$ and $b = 13$, the divisors of a are 1 and 3 while the divisors of b are 1 and 13. The greatest divisor they have in common is 1 so we can write

  $$\gcd(a, b) = \gcd(3, 13) = 1$$

  and say a and b are coprime.

So, if $N = 12$ how many integers y exist that are less than N and also coprime to N? If we list all the options (except 0)

$$\begin{array}{lclclcl} & & & & \gcd(12,6) & = & 6 \\ \gcd(12,1) & = & 1 & & \gcd(12,7) & = & 1 \\ \gcd(12,2) & = & 2 & & \gcd(12,8) & = & 4 \\ \gcd(12,3) & = & 3 & & \gcd(12,9) & = & 3 \\ \gcd(12,4) & = & 4 & & \gcd(12,10) & = & 2 \\ \gcd(12,5) & = & 1 & & \gcd(12,11) & = & 1 \end{array}$$

the answer is four, i.e., 1, 5, 7 and 11, so $\Phi(12) = 4$. Notice that the values of y which produce $\gcd(N, y) = 1$ are precisely those which make up the set $\mathbb{Z}_N^\star$. We need not go through the same task for $N = 13$: *all* positive integers less than 13 are coprime to 13 because it is prime, so $\Phi(13) = 12$.

Another interesting fact is that if you take any $x \in \mathbb{Z}_N^\star$ and raise it to the power $\Phi(N)$ modulo N, the result is 1. In short, for every $x \in \mathbb{Z}_N^\star$ we have

$$x^{\Phi(N)} \equiv 1 \pmod{N}.$$

For example, consider $x = 5$ which is an element of both $\mathbb{Z}_{12}^\star$ and $\mathbb{Z}_{13}^\star$. In these cases we have

$$\begin{array}{lclclclr} x^{\Phi(12)} & = & 5^4 & = & 625 & \equiv & 1 & \pmod{12}, \\ x^{\Phi(13)} & = & 5^{12} & = & 244140625 & \equiv & 1 & \pmod{13}. \end{array}$$

Implement (task #2)

Demonstrate this fact is also true for other elements of $\mathbb{Z}_{12}^\star$ and $\mathbb{Z}_{13}^\star$ by working out similar results for them.

2 Modular arithmetic: the practice

In Chapter 3, one of the goals was to convince ourselves that we could write down an algorithm for multiplication: the premise was that if we could do this, we were closer to writing a program to do the same thing on a computer.

You can easily imagine having a similar motivation with respect to modular arithmetic and cryptographic protocols based on it. The goal is to show that you can write algorithms to compute *all* the modular arithmetic operations we need using *only* integer arithmetic as a starting point. Put another way, all the theory that surrounds modular arithmetic is important, but if all you want to do is make some cryptographic protocol work on a computer then there is no magic involved: we can describe such computation entirely in terms of what you already know.

2.1 Addition and subtraction

Imagine we start off with $0 \leq x, y < N$, i.e., both x and y are in the set $\mathbb{Z}_N = \{0, 1, \ldots, N-1\}$. If we add x and y together to get $t = x + y$, we end up with a result that satisfies

$$0 \leq t \leq 2 \cdot (N-1).$$

The smallest result we can end up with is $t = 0$ which happens when x and y are as small as they can be, i.e., $x = y = 0$; the largest result we can end up with is $t = 2 \cdot (N-1)$ which happens when x and y are as large as then can be, i.e., $x = y = N - 1$. We would like to apply a modular reduction to t, i.e., compute $t' \equiv t \pmod{N}$. This is simple in that to get

$$0 \leq t' \leq N - 1,$$

we just subtract N from t if $t \geq N$. This is captured in Algorithm 2a, and demonstrated in action by some examples:

- Imagine $x = 8$, $y = 7$ and $N = 12$: we compute $t = 8 + 7 = 15$, and then $t' = 15 - 12 = 3$ since $t \geq 12$. Checking the modular reduction, we can see that

$$3 \equiv 15 \pmod{12}.$$

- Imagine $x = 1$, $y = 7$ and $N = 12$: we compute $t = 1 + 7 = 8$, and then $t' = 8$ since $t < 12$, i.e., no modular reduction is needed in this case because we know

$$8 \equiv 8 \pmod{12}.$$

```
algorithm Add-Mod($x, y, N$) begin
    $t \leftarrow x + y$
    if $t \geq N$ then
        $t' \leftarrow t - N$
    end
    else
        $t' \leftarrow t$
    end
    return $t'$
end
```

(a) $x + y \pmod{N}$.

```
algorithm Subtract-Mod($x, y, N$) begin
    $t \leftarrow x - y$
    if $t < 0$ then
        $t' \leftarrow t + N$
    end
    else
        $t' \leftarrow t$
    end
    return $t'$
end
```

(b) $x - y \pmod{N}$.

```
algorithm Multiply-Mod($x, y, N$) begin
    $t \leftarrow x \cdot y$
    $t' \leftarrow t - (N \cdot \lfloor \frac{t}{N} \rfloor)$
    return $t'$
end
```

(c) $x \cdot y \pmod{N}$.

```
algorithm Exponentiate-Mod($x, y, N$) begin
    $t \leftarrow 1$
    for $i$ from $|y| - 1$ downto $0$ do
        $t \leftarrow$ Multiply-Mod($t, t, N$)
        if $y_i = 1$ then
            $t \leftarrow$ Multiply-Mod($t, x, N$)
        end
    end
    return $t$
end
```

(d) $x^y \pmod{N}$.

```
algorithm Euclidean($a, b$) begin
    $r_0 \leftarrow a$
    $r_1 \leftarrow b$
    $i \leftarrow 2$
    forever do
        if $(r_{i-2} \bmod r_{i-1}) = 0$ then
            return $r_{i-1}$
        end
        $q_i \leftarrow \lfloor \frac{r_{i-2}}{r_{i-1}} \rfloor$
        $r_i \leftarrow r_{i-2} - r_{i-1} \cdot q_i$
        $i \leftarrow i + 1$
    end
end
```

(e) $\gcd(a, b)$.

```
algorithm Extended-Euclidean($a, b$) begin
    $r_0 \leftarrow a, s_0 \leftarrow 1, t_0 \leftarrow 0$
    $r_1 \leftarrow b, s_1 \leftarrow 0, t_1 \leftarrow 1$
    $i \leftarrow 2$
    forever do
        if $(r_{i-2} \bmod r_{i-1}) = 0$ then
            return $r_{i-1}, s_{i-1}, t_{i-1}$
        end
        $q_i \leftarrow \lfloor \frac{r_{i-2}}{r_{i-1}} \rfloor$
        $r_i \leftarrow r_{i-2} - r_{i-1} \cdot q_i$
        $s_i \leftarrow s_{i-2} - s_{i-1} \cdot q_i$
        $t_i \leftarrow t_{i-2} - t_{i-1} \cdot q_i$
        $i \leftarrow i + 1$
    end
end
```

(f) $\text{xgcd}(a, b)$.

Figure 2: *Some algorithms for modular arithmetic.*

So to cut a long story short, computing a modular addition relies only on integer addition and subtraction.

Things work more or less the same way for modular subtraction: if we subtract y from x to get $t = x - y$, we end up with a result that satisfies

$$-(N-1) \leq t \leq (N-1).$$

The smallest result we can end up with is $t = -(N-1)$ which happens when x is as small as it can be and y is as large as it can be, i.e., $x = 0$ and $y = N-1$; the largest result we can end up with is $t = N-1$ which happens when x is as large as it can be and y is as small as it can be, i.e., $x = N-1$ and $y = 0$. Again, we would like to apply a modular reduction to get

$$0 \leq t' \leq N-1$$

which is again simple: we just add N to t if $t < 0$. This is captured in Algorithm 2b, and demonstrated in action by some examples:

- Imagine $x = 8$, $y = 7$ and $N = 12$: we compute $t = 8 - 7 = 1$, and then $t' = 1$ since $t \geq 0$. No modular reduction is required in this case because we know

  $$1 \equiv 1 \pmod{12}.$$

- Imagine $x = 1$, $y = 7$ and $N = 12$: we compute $t = 1 - 7 = -6$, and then $t' = -6 + 12 = 6$ since $t < 0$. Checking the modular reduction, we can see that

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

Once more it is clear that to compute a modular subtraction, integer addition and subtraction are all we need.

2.2 Multiplication

Imagine we start off with $0 \leq x, y < N$, i.e., both x and y are in the set $\mathbb{Z}_N = \{0, 1, \ldots, N-1\}$. Following the cases for addition and subtraction above, if we now multiply x and y together to get $t = x \cdot y$, we end up with a result that satisfies

$$0 \leq t \leq (N-1)^2.$$

The smallest result we can end up with is $t = 0$ which happens when x and y are as small as they can be, i.e., $x = y = 0$; the largest result we can end up with is $t = (N-1)^2$ which happens when x and y are as large as then can be, i.e., $x = y = N-1$.

We would like to apply a modular reduction to t, but this time things are more tricky. For addition and subtraction, the t we compute is only ever wrong by at most N, i.e., we only need to subtract or add N *once* to get the right result; here we might need to subtract N *many* times. Consider an example: imagine we set $N = 12$ and select $x = 10$ and $y = 11$. If we compute $t = x + y = 10 + 11 = 21$, then we only need to subtract N once from t to get the result $t' = 9 \equiv 21 \pmod{12}$. If however we compute $t = x \cdot y = 10 \cdot 11 = 110$, then we need to subtract N *nine* times to get $t' = 2 \equiv 110 \pmod{12}$. So basically, instead of

```
if $t \geq N$ then
    $t' \leftarrow t - N$
end
else
    $t' \leftarrow t$
end
```

we need something that *repeatedly* subtracts N until we get the right result, i.e., something more like

```
$t' \leftarrow t$
while $t' \geq N$ do
    $t' \leftarrow t' - N$
end
```

Potentially this will be *very* inefficient however; the reason is more or less the same as when we talked about repeated addition in Chapter 3. Instead, we can calculate how many multiples of N we *would* need to subtract using this approach, and then simply do them all in one go. The idea is basically to compute

$$t' = t - (N \cdot \left\lfloor \frac{t}{N} \right\rfloor).$$

The term $\left\lfloor \frac{t}{N} \right\rfloor$ is the number of times N goes into t; this is basically the number of times we would go around the loop above. As a result $N \cdot \left\lfloor \frac{t}{N} \right\rfloor$ tells us what to subtract from t, which we can now do in one step. This is captured in Algorithm 2c, and demonstrated in action by some examples:

- Imagine $x = 8$, $y = 7$ and $N = 12$: we compute $t = 8 \cdot 7 = 56$, and then
$$t' = 56 - (12 \cdot 4) = 8$$
since $\left\lfloor \frac{56}{12} \right\rfloor = 4$. Checking the modular reduction, we can see that
$$8 \equiv 56 \pmod{12}.$$

- Imagine $x = 1$, $y = 7$ and $N = 12$: we compute $t = 1 \cdot 7 = 7$, and then
$$t' = 7 - (12 \cdot 0) = 7$$
since $\left\lfloor \frac{7}{12} \right\rfloor = 0$, i.e., no modular reduction is needed in this case because we know that
$$7 \equiv 7 \pmod{12}.$$

Things are more involved here than in the case of modular addition and subtraction, but to compute a modular multiplication we *still* only rely on integer multiplication, division and subtraction.

2.3 Exponentiation

The case of modular exponentiation is (believe it or not) one of the easiest to resolve. What we end up with is Algorithm 2d, which should look at least a bit familiar: basically we have just taken the old algorithm based on Horner's Rule from Chapter 3 and replaced the normal, integer multiplications with modular multiplications instead. The idea is that if each multiplication applies a modular reduction, then of course the exponentiation algorithm as a whole will be sane.

Consider an example where $x = 3$ and $y = 6_{(10)} = 110_{(2)}$. We know that $3^6 = 729$ and that $729 \equiv 9 \pmod{12}$, so the question is whether the new algorithm will give us the same result. The steps it performs demonstrate that it does:

Step #1 Assign $t \leftarrow 1$.

Step #2 Assign $t \leftarrow t^2 \pmod{N}$, i.e., $t \leftarrow 1^2 \pmod{12} = 1$.

Step #3 Since $y_2 = 1$, assign $t \leftarrow t \cdot x \pmod{N}$, i.e., $t \leftarrow 1 \cdot 3 \pmod{12} = 3$.

Step #4 Assign $t \leftarrow t^2 \pmod{N}$, i.e., $t \leftarrow 3^2 \pmod{12} = 9$.

Step #5 Since $y_1 = 1$, assign $t \leftarrow t \cdot x \pmod{N}$, i.e., $t \leftarrow 9 \cdot 3 \pmod{12} = 3$.

Step #6 Assign $t \leftarrow t^2 \pmod{N}$, i.e., $t \leftarrow 3^2 \pmod{12} = 9$.

Step #7 Since $y_0 = 0$, skip the assignment $t \leftarrow t \cdot x \pmod{N}$.

Step #8 Return $t = 9$.

All we are relying on to do this is modular multiplication, and we already know that modular multiplication can be described using integer operations only.

2.4 Division (via inversion)

We have already seen that in modular arithmetic, dividing by some y only makes sense if y has a multiplicative inverse. This is only true if $y \in \mathbb{Z}_N^\star$ which, in turn, is only true if N and y are coprime, i.e., $\gcd(N, y) = 1$. When this all works out, we can write
$$x/y = x \cdot 1/y = x \cdot y^{-1} \pmod{N}$$
which implies that modular division of x by y boils down to finding the multiplicative inverse of y modulo N, and multiplying this by x modulo N.

2.4.1 Problem #1: computing $\gcd(N, y)$

The first challenge is to test whether y has a multiplicative inverse or not: for large values of N we cannot simply write out the whole set $\mathbb{Z}_N^\star$, we need to compute and test whether $\gcd(N, y) = 1$ or not.

One way would be to write out the prime factorisation of N and y and simply pick out the greatest divisor they have in common; if this turns out to be 1, we know N and y are coprime. We performed this exact task when we originally discussed coprimality, but imagine we now select the larger examples

$$\begin{array}{rcrcl} N & = & 1426668559730 & = & 2^1 \cdot 5^1 \cdot 157^1 \cdot 271^1 \cdot 743^1 \cdot 4513^1, \\ y & = & 810653094756 & = & 2^2 \cdot 3^2 \cdot 61^1 \cdot 157^1 \cdot 521^1 \cdot 4513^1. \end{array}$$

Using their prime factorisations (the right-hand part of each line), we can deduce that

$$\begin{array}{rcl} \gcd(N, y) & = & \gcd(1426668559730, 810653094756) \\ & = & 2 \cdot 157 \cdot 4513 \\ & = & 1417082 \end{array}$$

because both N and y have 2, 157 and 4513 as common divisors. However factoring is relatively hard work, particularly if N and y are large; if this *was* the best way to compute $\gcd(N, y)$, we would need a very fast computer or plenty of time on our hands! Luckily the **Euclidean algorithm** [8], an ancient algorithm due to Greek mathematician Euclid, can compute the same result rather more quickly.

Imagine we want to compute

$$\gcd(a, b)$$

assuming $a \geq b$ (swapping a and b if not). Understanding how the Euclidean algorithm works hinges on writing

$$a = b \cdot q + r$$

where q is called the quotient and r is called the remainder. This is the same as saying that if we divide a by b, we get a quotient (how many times b divides into a) and a remainder (what is left after such a division). An important fact is that $0 \leq r < b \leq a$ because if r *were* larger, b would divide into a one more time than q says it should. Taking this as our starting point, we can formulate two rules:

1. If you divide a by b and the remainder is 0, then a is a multiple of b (because b divides a exactly): this implies that b is a common divisor of a and b.

2. If you divide a by b and the remainder is not 0, then we can say that

   $$\gcd(a, b) = \gcd(b, r).$$

   Why is this? Imagine we write out the prime factorisation of a as follows:

   $$a = p_0^{e_0} \cdot p_1^{e_1} \cdots p_{l-1}^{e_{l-1}}.$$

   One of those terms will be $d = \gcd(a, b)$. Therefore you could imagine a written instead as

   $$a = d \cdot a'$$

   where a' represents all the "other" terms. We can do the same thing with b, because we know d must divide it as well, to get

   $$b = d \cdot b'.$$

   As a result

   $$a - b = (d \cdot a') - (d \cdot b') = d \cdot (a' - b')$$

   and hence d clearly divides $a - b$. Now going back to what we know from above, i.e.,

   $$r = a - b \cdot q,$$

   it follows that

   $$a - b \cdot q = (d \cdot a') - (d \cdot b' \cdot q) = d \cdot (a' - b' \cdot q).$$

   So basically

   $$r = d \cdot (a' - b' \cdot q)$$

   which means that d also divides r. To cut a long story short, $\gcd(b, r)$ will therefore give us the same result as $\gcd(a, b)$; because r is smaller than a and b, computing $\gcd(b, r)$ retains the restrictions we had to start with that said $a \geq b$.

Based on these two rules, the Euclidean algorithm computes the following sequence:

$$\begin{array}{rcccccc} r_0 & = & a & & & & \\ r_1 & = & b & & & & \\ r_2 & = & r_0 & - & r_1 & \cdot & q_2 \\ r_3 & = & r_1 & - & r_2 & \cdot & q_3 \\ & \vdots & & & & & \end{array}$$

Basically the sequence is describing repeated application of the second rule above: the i-th remainder r, which we call r_i, is computed via

$$r_i = r_{i-2} - r_{i-1} \cdot q_i$$

where the i-th quotient is $q_i = r_{i-2}/r_{i-1}$. The sequence will finish in some m-th step when r_{m-1} divides r_{m-2} exactly (meaning we compute $r_m = 0$), at which point the answer we want is r_{m-1}; this corresponds to the first rule above. There are plenty of ways to write this down as an algorithm; one simple approach is shown in Algorithm 2e.

If we invoke

$$\textsc{Euclidean}(21, 12),$$

i.e., try to compute $\gcd(a, b)$ with $a = 21$ and $b = 12$ as in the example above, the algorithm computes the sequence

$$\begin{array}{rcrcrcrcr} r_0 & = & 21 & & & & & & \\ r_1 & = & 12 & & & & & & \\ r_2 & = & 21 & - & 12 & \cdot & 1 & = & 9 \\ r_3 & = & 12 & - & 9 & \cdot & 1 & = & 3 \\ r_4 & = & 9 & - & 3 & \cdot & 3 & = & 0 \end{array}$$

meaning that $m = 4$ and hence $r_{m-1} = r_3 = 3$. The algorithm itself does this via the following steps:

Step #1 Assign $r_0 \leftarrow 21$.

Step #2 Assign $r_1 \leftarrow 12$.

Step #3 Assign $i \leftarrow 2$.

Step #4 Since $r_0 \bmod r_1 = 21 \bmod 12 \neq 0$, skip the return.

Step #5 Assign $r_2 \leftarrow r_0 \bmod r_1 \cdot \lfloor \frac{r_0}{r_1} \rfloor = 21 - 12 \cdot \lfloor \frac{21}{12} \rfloor = 21 - 12 \cdot 1 = 9$.

Step #6 Assign $i \leftarrow i + 1 = 3$.

Step #7 Since $r_1 \bmod r_2 = 12 \bmod 9 \neq 0$, skip the return.

Step #8 Assign $r_3 \leftarrow r_1 \bmod r_2 \cdot \lfloor \frac{r_1}{r_2} \rfloor = 12 - 9 \cdot \lfloor \frac{12}{9} \rfloor = 12 - 9 \cdot 1 = 3$.

Step #9 Assign $i \leftarrow i + 1 = 4$.

Step #10 Since $r_2 \bmod r_3 = 9 \bmod 3 = 0$, return $r_3 = 3$.

What about the example we started off with originally? The a and b are larger so we have a longer sequence, but by invoking

$$\textsc{Euclidean}(1426668559730, 810653094756)$$

we get

$$\begin{array}{rcrcrcrcr} r_0 & = & 1426668559730 & & & & & & \\ r_1 & = & 810653094756 & & & & & & \\ r_2 & = & 1426668559730 & - & 810653094756 & \cdot & 1 & = & 616015464974 \\ r_3 & = & 810653094756 & - & 616015464974 & \cdot & 1 & = & 194637629782 \\ r_4 & = & 616015464974 & - & 194637629782 & \cdot & 3 & = & 32102575628 \\ r_5 & = & 194637629782 & - & 32102575628 & \cdot & 6 & = & 2022176014 \\ r_6 & = & 32102575628 & - & 2022176014 & \cdot & 15 & = & 1769935418 \\ r_7 & = & 2022176014 & - & 1769935418 & \cdot & 1 & = & 252240596 \\ r_8 & = & 1769935418 & - & 252240596 & \cdot & 7 & = & 4251246 \\ r_9 & = & 252240596 & - & 4251246 & \cdot & 59 & = & 1417082 \\ r_{10} & = & 4251246 & - & 1417082 & \cdot & 3 & = & 0 \end{array}$$

via similar steps, meaning that $m = 10$ and hence $r_{m-1} = r_9 = 1417082$ which matches the original result.

2.4.2 Problem #2: computing $y^{-1} \pmod{N}$

Now we can tell whether or not y has a multiplicative inverse modulo N, the next challenge is to compute the inverse itself, i.e., $y^{-1} \pmod{N}$. One option is to use the fact that we know

$$y^{\Phi(N)} \equiv 1 \pmod{N}.$$

Based on this, it also makes sense that

$$\begin{array}{ccccl} & \vdots & & \vdots & \\ y^{\Phi(N)-2} & \equiv & y^{-2} & \equiv & 1/y^2 \pmod{N} \\ y^{\Phi(N)-1} & \equiv & y^{-1} & \equiv & 1/y \pmod{N} \\ y^{\Phi(N)+0} & \equiv & y^{0} & \equiv & 1 \pmod{N} \\ y^{\Phi(N)+1} & \equiv & y^{1} & \equiv & y \pmod{N} \\ y^{\Phi(N)+2} & \equiv & y^{2} & \equiv & y^2 \pmod{N} \\ & \vdots & & \vdots & \end{array}$$

Or, put another way, we can easily compute $y^{-1} \pmod{N}$ if we know $\Phi(N)$ because we already know how to do modular exponentiation. On the other hand, we already said that computing $\Phi(N)$ is relatively hard because we need to factor N.

An alternative option is to use use a variant of the Euclidean algorithm, called the Extended Euclidean Algorithm (EEA) [11]; sometimes this is called "XGCD" as in "eXtended". Recall that the Euclidean algorithm used

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

The idea of the extended Euclidean algorithm is to unwind the steps and write each r_i in terms of a and b. This produces

$$\begin{array}{rcl} r_0 & = & a \\ r_1 & = & b \\ r_2 & = & r_0 - r_1 \cdot q_2 \\ & = & a - b \cdot q_2 \\ r_3 & = & r_1 - r_2 \cdot q_3 \\ & = & b - (a - b \cdot q_2) \cdot q_3 \\ & = & -a \cdot q_3 + b \cdot (1 + q_2 \cdot q_1) \\ & \vdots & \end{array}$$

This quickly starts to look nasty, so we simplify it by saying that in the i-th step we have

$$r_i = a \cdot s_i + b \cdot t_i$$

where s_i and t_i collect together all the nasty looking parts in one place. The sequence again finishes in the m-th step. But if we keep track of s_i and t_i, as well as getting r_{m-1} we also get s_{m-1} and t_{m-1}. Algorithm 2f shows the updated algorithm that keeps track of the extra parts.

If we invoke the new algorithm via EXTENDED-EUCLIDEAN(21, 12), we get

$$\begin{array}{rcrcrcrcr} r_0 & = & 21 & & & & & & \\ s_0 & = & 1 & & & & & & \\ t_0 & = & 0 & & & & & & \\ r_1 & = & 12 & & & & & & \\ s_1 & = & 0 & & & & & & \\ t_1 & = & 1 & & & & & & \\ r_2 & = & 21 & - & 12 & \cdot & 1 & = & 9 \\ s_2 & = & 1 & - & 0 & \cdot & 1 & = & 1 \\ t_2 & = & 0 & - & 1 & \cdot & 1 & = & -1 \\ r_3 & = & 12 & - & 9 & \cdot & 1 & = & 3 \\ s_3 & = & 0 & - & 1 & \cdot & 1 & = & -1 \\ t_3 & = & 1 & - & -1 & \cdot & 1 & = & 2 \\ r_4 & = & 9 & - & 3 & \cdot & 3 & = & 0 \\ s_4 & = & 1 & - & -1 & \cdot & 3 & = & 4 \\ t_4 & = & -1 & - & 2 & \cdot & 3 & = & -7 \end{array}$$

meaning that $m = 4$. We can verify that this is sane because

$$\begin{array}{rcccccccccc} r_{m-1} & = & 3 & = & a & \cdot & s_{m-1} & + & b & \cdot & t_{m-1} \\ & & & = & 21 & \cdot & -1 & + & 12 & \cdot & 2 \\ & & & = & -21 & & & + & 24 & & \\ & & & = & 3 & & & & & & \end{array}$$

as expected. The point is that the extra parts the algorithm computes turn out to be very useful because given

$$r_{m-1} = \gcd(a, b) = 1$$

if a and b are coprime, we therefore know

$$1 = a \cdot s_{m-1} + b \cdot t_{m-1}.$$

As a result, we can compute the multiplicative inverse of x modulo N by setting $a = x$ and $b = N$. This means we get

$$1 = x \cdot s_{m-1} + N \cdot t_{m-1}$$

and, if we look at this modulo N,

$$1 \equiv x \cdot s_{m-1} \pmod{N}$$

since $N \cdot t_{m-1} \equiv 0 \pmod{N}$ (in fact any multiple of N is equivalent to 0 when reduced modulo N) or rather

$$x^{-1} \equiv s_{m-1} \pmod{N}$$

meaning that s_{m-1} is the multiplicative inverse we want.

3 From modular arithmetic to cryptographic protocols

As we saw in the introduction, a cryptographic protocol is a description of how the parties involved accomplish some task (e.g., sending messages between each other securely). The two tasks we focus on relate to the key distribution problem: can Alice and Bob communicate in a secure way *without* having to agree on a shared key before they start? We can describe two potential solutions using just the modular arithmetic studied so far:

1. key agreement protocols [15], and
2. public-key encryption schemes [21].

One of the motivations for solving this problem is that we, as Alice say, often want to communicate with a non-human Bob. Imagine for example if Bob were a web-site we want to buy something from; it is not feasible for us to agree a shared key with *every* possible web-site, so solutions like those above are critically important.

3.1 Diffie-Hellman key exchange

Suppose Alice wants to send a message to Bob, but they do not have a shared key; they *could* use the protocol we saw in the introduction, but first they need a secure way to agree on a shared key. The Diffie-Hellman key exchange [5] protocol, invented in 1976 by cryptographers Whitfield Diffie and Martin Hellman, offers one solution to this problem. A historically interesting fact is that the same protocol was invented independently, and around the same time, by Malcolm Williamson, a cryptographer at GCHQ [12] in the UK; this only became apparent when the intelligence agency declassified the work, making it public in 1997.

Before they start, Alice and Bob agree on two public values: a large prime number p, and some $g \in \mathbb{Z}_p$. The values are made available to *everyone*, e.g., by publishing them on a web-site. You can think of them as fixed settings which allow the protocol to work, a bit like the settings that tell a web-browser how to work. As a result, Alice and Bob can agree on a key as follows:

$$\begin{array}{ccc}
\text{Alice} & \text{Eve} & \text{Bob} \\ \hline
x \xleftarrow{\$} \mathbb{Z}_p & & y \xleftarrow{\$} \mathbb{Z}_p \\
a \leftarrow g^x \pmod{p} & & b \leftarrow g^y \pmod{p} \\
 & \xrightarrow{a} & \\
 & \xleftarrow{b} & \\
k \leftarrow b^x \pmod{p} & & k \leftarrow a^y \pmod{p} \\ \hline
\end{array}$$

The protocol Alice and Bob engage in has five simple steps:

1. Alice selects x, a random element in $\mathbb{Z}_p$; at the same time Bob selects y, a random element in $\mathbb{Z}_p$.
2. Alice computes $a = g^x \pmod{p}$; at the same time Bob computes $b = g^y \pmod{p}$.

3. Alice communicates a to Bob .
4. Bob communicates b to Alice.
5. Alice computes $k = b^x \pmod{p}$; at the same time Bob computes $k = a^y \pmod{p}$.

Notice that at the end of the protocol Alice and Bob end up with the *same* shared key k; this is because for Alice

$$\begin{array}{lcll} k & \equiv & b^x & \pmod{p} \\ & \equiv & (g^y)^x & \pmod{p} \\ & \equiv & g^{y \cdot x} & \pmod{p} \end{array}$$

while for Bob we find

$$\begin{array}{lcll} k & \equiv & a^y & \pmod{p} \\ & \equiv & (g^x)^y & \pmod{p} \\ & \equiv & g^{x \cdot y} & \pmod{p} \end{array}$$

A concrete example demonstrates that crucially, this happens even though Alice does not know y, Bob does not know x and Eve knows neither y nor x. Imagine we make the settings $p = 13$ and $g = 7$ available to everyone. The protocol can then proceed as follows:

1. Alice selects $x = 9$; at the same time Bob selects $y = 3$.
2. Alice computes $a = g^x \pmod{p} = 7^9 \pmod{13} = 8$; at the same time Bob computes $b = g^y \pmod{p} = 7^3 \pmod{13} = 5$.
3. Alice communicates $a = 8$ to Bob .
4. Bob communicates $b = 5$ to Alice.
5. Alice computes $k = b^x \pmod{p} = 5^9 \pmod{13} = 5$; at the same time Bob computes $k = a^y \pmod{p} = 8^3 \pmod{13} = 5$.

Research (task #3)

What if all you have is a block cipher? Clearly Alice and Bob cannot rely on the fact they know k before they start, but what if they each share a different key with some trusted third-party Trent (call them k_{Alice} and k_{Bob} say): can you write down a protocol whereby they end up with a shared k using Trent to help out?

3.2 RSA encryption

Now suppose Alice wants to send a message to Bob, without the need to agree on a shared key *at all*. The RSA public-key encryption [22], invented in 1978 by Ron Rivest, Adi Shamir and Leonard Adleman (who lend their initials to the name), represents an efficient and widely-used solution. In short, it allows us to avoiding the need for shared keys entirely, and hence side-step the key distribution problem. Again, cryptographers at GCHQ [12] produced similar results behind closed doors: Clifford Cocks had developed a similar scheme independently in 1973, but this was not declassified until 1997.

Before Alice can encrypt a message, Bob has to do some work. He picks two large prime numbers p and q and uses them to first compute $N = p \cdot q$, then

$$\Phi(N) = (p-1) \cdot (q-1).$$

Notice that only Bob can do this because only he knows p and q, the prime factors of N. He then selects a small number e which is coprime to $\Phi(N)$, and solves

$$e \cdot d \equiv 1 \pmod{\Phi(N)},$$

that is, he computes d, the multiplicative inverse of e modulo $\Phi(N)$. Remember, this means there exists a κ (known only by Bob in this case) such that

$$e \cdot d = 1 + \kappa \cdot \Phi(N).$$

The pair (N, e) is the **public-key** for Bob, while (N, d) is the **private-key**. The idea is that Bob can make his public-key available to *everyone* (e.g., by publishing it on his web-site), and that by downloading it doing so *anyone* can encrypt messages for him. Only Bob knows the corresponding private-key, so only he will be able to decrypt said messages.

As a result, Alice can send $M \in \mathbb{Z}_N^\star$ to Bob as follows:

Alice	Eve	Bob
$C \leftarrow M^e \pmod{N}$		
	$\xrightarrow{C}$	
		$M \leftarrow C^d \pmod{N}$

As in the introduction, the protocol that Alice and Bob engage in has three simple steps:

1. Alice encrypts a plaintext M using the public-key (N, e) to produce the ciphertext message
$$C = M^e \pmod{N}.$$
2. Alice communicates C to Bob.
3. Bob decrypts the ciphertext C using the private-key (N, d) to recover the plaintext message
$$M = C^d \pmod{N}.$$

But why does this work? Well, it follows from a fact we saw earlier: for any $x \in \mathbb{Z}_N^\star$ we have that
$$x^{\Phi(N)} \equiv 1 \pmod{N}.$$

This means
$$\begin{array}{rcll} C^d & \equiv & (M^e)^d & \pmod{N} \\ & \equiv & M^{e \cdot d} & \pmod{N} \\ & \equiv & M^{1+\kappa \cdot \Phi(N)} & \pmod{N} \\ & \equiv & M \cdot M^{\kappa \cdot \Phi(N)} & \pmod{N} \\ & \equiv & M \cdot (M^{\Phi(N)})^\kappa & \pmod{N} \\ & \equiv & M \cdot 1^\kappa & \pmod{N} \\ & \equiv & M \cdot 1 & \pmod{N} \\ & \equiv & M & \pmod{N} \end{array}$$

and so we get the message M as a result! The crucial difference between this and what we saw in the introduction is that Alice and Bob no longer use a shared key: *everyone* knows the public-key owned by Bob that was used for encryption, but *only* Bob knows the corresponding private-key used for decryption. We can demonstrate this using another concrete example. Imagine Bob selects
$$\begin{array}{rcr} p & = & 5 \\ q & = & 11 \end{array}$$

meaning $N = 55$ and $\Phi(N) = (p-1) \cdot (q-1) = 40$. He also selects $e = 7$, noting that $\gcd(e, \Phi(N)) = 1$, and computes $d = 23$, again noting that $e \cdot d = 161 \equiv 1 \pmod{\Phi(N)}$. The protocol can then proceed as follows:

1. Alice encrypts a plaintext $M = 46$ using the public-key $(N, e) = (55, 7)$ to produce the ciphertext message
$$C = M^e \pmod{N} = 46^7 \pmod{55} = 51.$$
2. Alice communicates C to Bob.
3. Bob decrypts the ciphertext $C = 51$ using the private-key $(N, d) = (55, 23)$ to recover the plaintext message
$$M = C^d \pmod{N} = 51^{23} \pmod{55} = 46.$$

Research (task #4)

The ElGamal [7] public-key encryption scheme is an alternative to RSA; it also makes use of modular arithmetic. Three public values, namely g, p and q, are known to everyone:

- Alice chooses a random private-key x from the set $\{1, 2, \ldots q-1\}$ and computes

  $$h = g^x \pmod{p}$$

  which is her public-key made available to everyone.

- Bob wants to send a message m to Alice: he first chooses a random k from the set $\{1, 2, \ldots q-1\}$ and computes

  $$\begin{array}{lcll} c_1 & = & g^k & \pmod{p} \\ c_2 & = & h^k \cdot m & \pmod{p} \end{array}$$

  The pair (c_1, c_2) is the ciphertext sent to Alice.

How can Alice decrypt (c_1, c_2) to recover the plaintext m? Work out the steps required, and show that the overall scheme works using an example (e.g., using the public values $g = 105$, $p = 107$ and $q = 53$).

3.3 Functional versus secure

In describing the protocols above, we showed they do what they should: they satisfy the functional requirements. But we forgot about (or more like ignored) Eve: how can we be sure the protocols are secure? We need to be able to reason about what Eve can do as well: it is not good enough to say in the case of RSA "Eve does not see M communicated between Alice and Bob so the protocol is secure". A standard approach is to model all the things that Eve must do in order that a protocol is insecure, and relate them to a "hard" Mathematical problem: if we can prove things about the problem, those proofs mean something in terms of Eve. For example, if we can reason about *how* hard the problem is (e.g., how many steps an algorithm will take to solve it), we can reason about how easily Eve will be able to attack the protocol.

Both of the protocols we introduced rely on a common building block, namely the computation of

$$z = x^y \pmod{N}$$

for various x, y and N. This allows us to consider (at least) two problems:

- If N is prime, say $N = p$, then given N, x and z computing y is believed to be hard (for large values of N). This is called the **Discrete Logarithm Problem (DLP)** [6]; we write down an instance of this problem as $\mathsf{DLP}(z, x, N)$.

- If N is a product of two primes, say $N = p \cdot q$, then given N, y and z computing x is believed to be hard without also knowing p and q. This is called the **RSA problem** [23]; we write down an instance of this problem as $\mathsf{RSA}(z, y, N)$.

In both cases, a problem instance is simply a challenge to Eve. When you see $\mathsf{DLP}(z, x, N)$ this should be read as "find y such that $z \equiv x^y \pmod{N}$"; when you see $\mathsf{RSA}(z, y, N)$ this should be read as"find an x such that $z \equiv x^y \pmod{N}$".

In general, a function f which is easy to compute but which is hard to invert or "reverse" (meaning f^{-1} is hard to compute) is called a **one-way function** [18]. Our examples fall exactly into this category: it is easy to compute $z = x^y \pmod{N}$, but (depending on the N we choose) hard to solve either $\mathsf{DLP}(z, x, N)$ or $\mathsf{RSA}(z, y, N)$. The question is, what do we mean by "hard"? Usually we try to answer this by using the same ideas as in Chapter 3. By saying that computing f should be easy, we mean that for a problem of size n (for example the number of bits in N), we might have an algorithm which does not take too many steps to compute f. So maybe f is $O(n)$ or $O(n^2)$ or even $O(n^3)$. In contrast, computing f^{-1} should be much harder; for example we might select f so that the *best known* algorithm to compute f^{-1} takes $O(2^n)$ steps.

So how does this relate to what Eve might do? Consider a model where Eve observes the communication between Alice and Bob (in addition to getting any publicly available values). How might she attack the protocol?

1. In the Diffie-Hellman example, Eve (like everyone else) gets access to the settings p and g and sees a and b communicated from Alice to Bob and vice versa.

Assuming Eve wants to recover the shared key k, one approach would be to solve the problem instance DLP(a, g, p); this yields x and means Eve can compute

$$b^x = k$$

just like Bob did. Alternatively, Eve could solve the problem instance DLP(b, g, p) to get y and compute

$$a^y = k$$

just like Alice did. But both problem instances are as hard as each other, so either way we might argue that if solving them is hard then Diffie-Hellman key exchange is secure.

2. In the RSA example, Eve (like everyone else) gets access to the public-key for Bob, i.e., (N, e) and sees C communicated by Alice to Bob.

   Assuming Eve wants to recover the message M, one can imagine two approaches:

   (a) Factor N to recover p and q, then compute $\Phi(N)$. By doing this, d can be computed and Eve can decrypt C just like Bob did.

   (b) Solve the problem instance RSA(C, e, N) which gives the result M.

   As a result we might argue that RSA is secure if factoring is hard, *and* solving instances of the RSA problem is hard.

This seems great: if Eve wants to do anything we might regard as "bad", she has to work out how to solve what we are assuming are hard mathematical problems. If the problems really are hard, then we can say the protocols are secure.

But what if Eve does something other than what we expect? What if she acts outside our current model, which assumes she can only observe the communication between Alice and Bob? For example, imagine Eve has cut the network cable that links Alice and Bob and placed her own computer in between them; this is known as a **man-in-the-middle** attack [16]. Now Eve becomes the active adversary Mallory, and can do all sorts of other things. The question is, are the two protocols still secure?

1. For the Diffie-Hellman example, imagine the protocol changes because of how Mallory behaves; Alice and Bob behave in exactly the same way. The result is as follows:

$$
\begin{array}{ccccc}
\text{Alice} & & \text{Mallory} & & \text{Bob} \\
\hline
 & & x' \xleftarrow{\$} \mathbb{Z}_p & & \\
x \xleftarrow{\$} \mathbb{Z}_p & & y' \xleftarrow{\$} \mathbb{Z}_p & & y \xleftarrow{\$} \mathbb{Z}_p \\
 & & a' = g^{x'} \pmod{p} & & \\
a \leftarrow g^x \pmod{p} & & b' = g^{y'} \pmod{p} & & b \leftarrow g^y \pmod{p} \\
 & \xrightarrow{a} & & \xrightarrow{a'} & \\
 & \xleftarrow{b'} & & \xleftarrow{b} & \\
 & & k_1 = a^{y'} \pmod{p} & & \\
k_1 \leftarrow b'^x \pmod{p} & & k_2 = b^{x'} \pmod{p} & & k_2 \leftarrow a'^y \pmod{p} \\
\hline
\end{array}
$$

Since

$$
\begin{array}{rcll}
k_1 & = & b'^x & \pmod{p} \\
 & = & (g^{y'})^x & \pmod{p} \\
 & = & g^{y' \cdot x} & \pmod{p} \\
 & = & (g^x)^{y'} & \pmod{p} \\
 & = & a^{y'} & \pmod{p}
\end{array}
$$

and

$$
\begin{array}{rcll}
k_2 & = & a'^y & \pmod{p} \\
 & = & (g^{x'})^y & \pmod{p} \\
 & = & g^{x' \cdot y} & \pmod{p} \\
 & = & (g^y)^{x'} & \pmod{p} \\
 & = & b^{x'} & \pmod{p}
\end{array}
$$

Mallory agrees one key with Alice and one with Bob: Alice *thinks* she is communicating with Bob and vice versa, but actually they are communicating *through* Mallory. When Alice sends a message to Bob encrypted using k_1, Mallory can simply decrypt the message and read it, then re-encrypt it using k_2 and send it on: neither Alice nor Bob are any the wiser because from their point of view, nothing has gone wrong.

2. For the RSA example, imagine the protocol changes to the following:

Alice		Mallory		Bob
$C \leftarrow M^e \pmod{N}$				
	$\xrightarrow{C}$			
		$C' \leftarrow 2^e \cdot C \pmod{N}$		
			$\xrightarrow{C'}$	
				$M' \leftarrow C'^d \pmod{N}$

Since

$$\begin{array}{rcll} C' & \equiv & 2^e \cdot C & \pmod{N} \\ & & 2^e \cdot M^e & \pmod{N} \\ & & (2 \cdot M)^e & \pmod{N}, \end{array}$$

when Bob decrypts C', he gets an M' equal to $2 \cdot M$. That seems quite bad: if M was a message saying "Alice owes Bob £100" then Alice might actually end up owing twice as much!

Both cases are meant to illustrate that Eve (now Mallory) has avoided the hard mathematical problems: in order to do "something bad", there was no need to solve an instance of the RSA problem for example. What does this mean: is the protocol insecure? Or is the security model wrong? The combined difficulty of

1. understanding what a protocol should do,
2. understanding and modelling what any adversaries can do, and
3. designing a protocol that is efficient while also matching all functional requirements

make *good* cryptography very challenging. The fact that people are constantly developing new capabilities for Eve (e.g., new ways to factor numbers or solve the RSA problem) adds even more of a challenge; what is secure today may not be secure in ten or twenty years time, and equally the types of protocol parties want to engage in might change. But these challenges also make cryptography a fascinating subject: it must, by definition, constantly evolve to keep pace with both functionality and attack landscapes.

References

[1] *Wikipedia: Alice and Bob.* http://en.wikipedia.org/wiki/Alice_and_Bob (see p. 3).

[2] *Wikipedia: Block cipher.* http://en.wikipedia.org/wiki/Block_cipher (see p. 3).

[3] *Wikipedia: Block cipher modes of operation.* http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation (see p. 4).

[4] *Wikipedia: Coprime.* http://en.wikipedia.org/wiki/Coprime (see p. 9).

[5] *Wikipedia: Diffie-Hellman.* http://en.wikipedia.org/wiki/Diffie-Hellman (see p. 17).

[6] *Wikipedia: Discrete logarithm.* http://en.wikipedia.org/wiki/Discrete_logarithm (see p. 20).

[7] *Wikipedia: ElGamal encryption.* http://en.wikipedia.org/wiki/ElGamal_encryption (see p. 20).

[8] *Wikipedia: Euclidean algorithm.* http://en.wikipedia.org/wiki/Euclidean_algorithm (see pp. 4, 14).

[9] *Wikipedia: Euler's totient function.* http://en.wikipedia.org/wiki/Euler's_totient_function (see p. 8).

[10] *Wikipedia: Exponentiation by squaring.* http://en.wikipedia.org/wiki/Exponentiation_by_squaring (see p. 4).

[11] *Wikipedia: Extended Euclidean algorithm.* http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm (see p. 16).

[12] *Wikipedia: GCHQ.* http://en.wikipedia.org/wiki/Government_Communications_Headquarters (see pp. 17, 18).

[13] *Wikipedia: Identity function*. http://en.wikipedia.org/wiki/Identity_function (see p. 6).

[14] *Wikipedia: Integer*. http://en.wikipedia.org/wiki/Integer (see p. 7).

[15] *Wikipedia: Key agreement protocol*. http://en.wikipedia.org/wiki/Key-agreement_protocol (see pp. 4, 17).

[16] *Wikipedia: Man-in-the-middle attack*. http://en.wikipedia.org/wiki/Man-in-the-middle_attack (see p. 21).

[17] *Wikipedia: Number theory*. http://en.wikipedia.org/wiki/Number_theory (see p. 4).

[18] *Wikipedia: One-way function*. http://en.wikipedia.org/wiki/One-way_function (see p. 20).

[19] *Wikipedia: Prime factor*. http://en.wikipedia.org/wiki/Prime_factor (see p. 9).

[20] *Wikipedia: Prime number*. http://en.wikipedia.org/wiki/Prime_number (see p. 8).

[21] *Wikipedia: Public key cryptography*. http://en.wikipedia.org/wiki/Public-key_cryptography (see pp. 4, 17).

[22] *Wikipedia: RSA*. http://en.wikipedia.org/wiki/RSA (see p. 18).

[23] *Wikipedia: RSA Problem*. http://en.wikipedia.org/wiki/RSA_problem (see p. 20).

[24] *Wikipedia: Symmetric key cryptography*. http://en.wikipedia.org/wiki/Symmetric_key_algorithm (see p. 4).

[25] *Wikipedia: Transport Layer Security (TLS)*. http://en.wikipedia.org/wiki/Transport_Layer_Security (see p. 4).

I have a question/comment/complaint for you. Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way.

However, keep in mind we are far from perfect; mistakes are of course possible, although hopefully rare. Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Can I use this material for something ? We are distributing these PDFs under the Creative Commons Attribution-ShareAlike License (CC BY-SA)

Is there a printed version of this material I can buy? Yes: Springer have published selected Chapters in

`http://www.springer.com/computer/book/978-3-319-04041-7`

while *also* allowing us to keep electronic versions online. Any royalties from this published version are donated to the UK-based Computing At School (CAS) group, whose ongoing work can be followed at

`http://www.computingatschool.org.uk/`

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.